

Chapter 3 : STACKS AND QUEUES

A decorative L-shaped line in a dark blue color. It consists of a vertical line segment on the left and a horizontal line segment extending to the right, meeting at a right angle. The vertical line starts at the same level as the top of the text and extends downwards. The horizontal line starts at the same level as the top of the text and extends to the right.

3.1 STACKS

- The *stacks* and *queues* are special cases of the more general data type, *ordered list*.
- A *stack* is an ordered list in which insertions and deletions are made at one end called the *top*.

$$S = (a_0, a_1, \dots, a_{n-1})$$

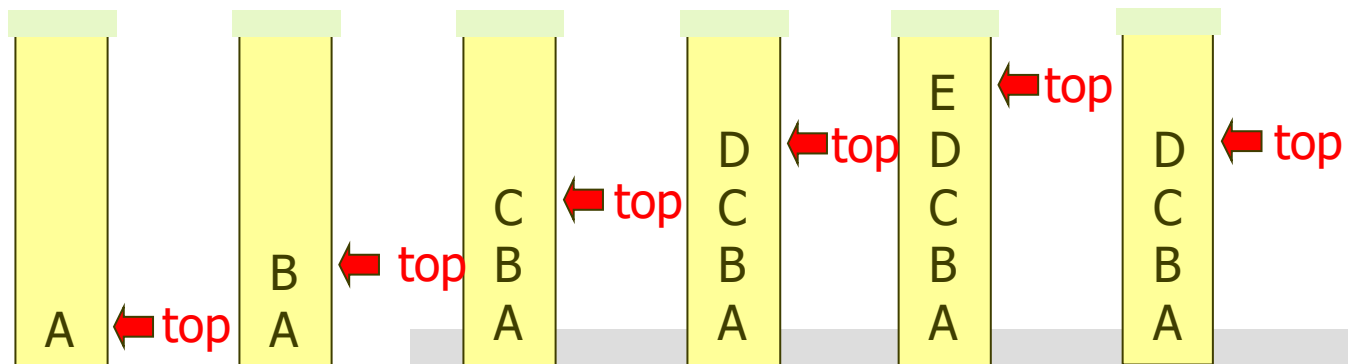
↑

bottom

↑

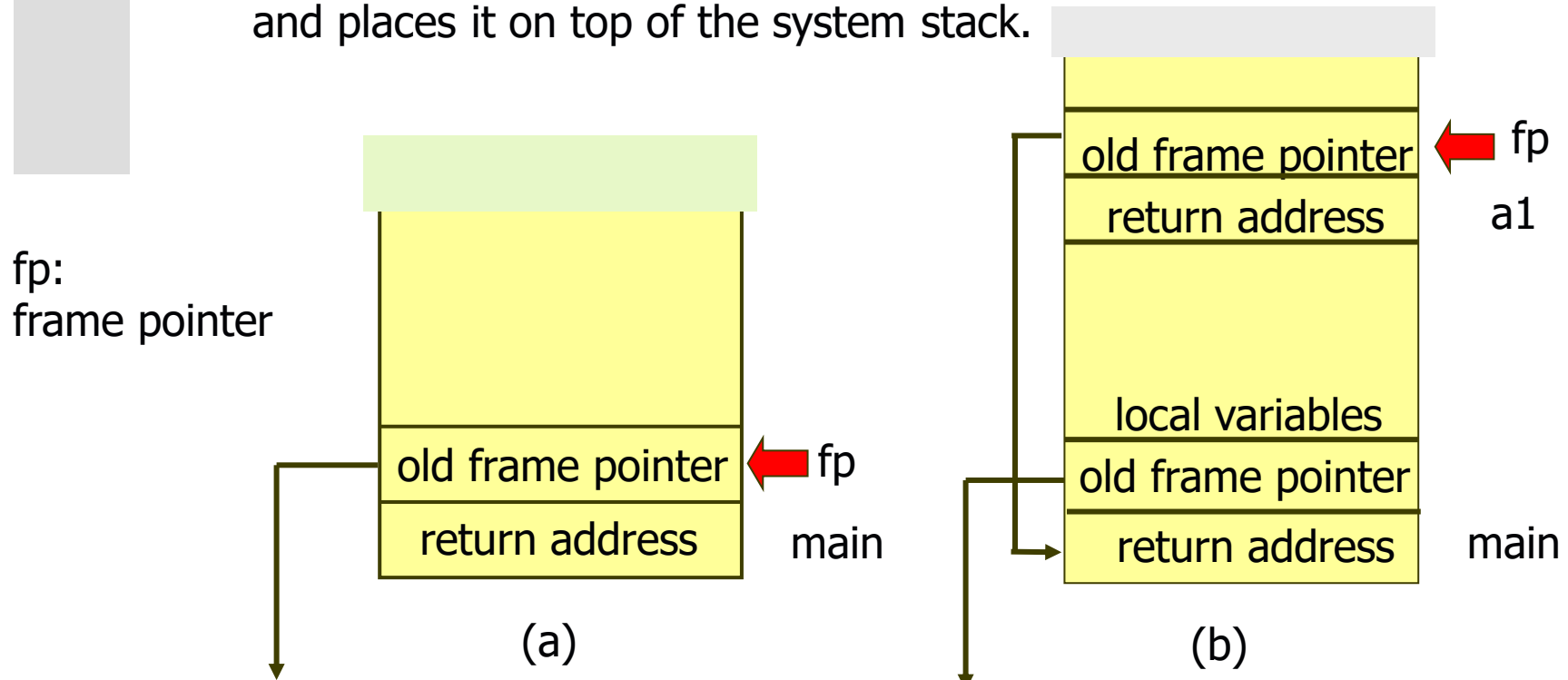
top

Last-In-First-Out (LIFO) list.



■ Example 3.1 [System stack] :

To process function calls, whenever a function is invoked, the program creates a structure, referred to as *activation record* or *stack frame* and places it on top of the system stack.



■ [ADT 3.1]: Abstract data type Stack

ADT *Stack* is

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

functions : for all $stack \in Stack$, $item \in element$, $maxStackSize \in \text{positive integer}$

Stack CreateS($maxStackSize$) ::=

create an empty stack whose maximum size is $maxStackSize$

Boolean IsFull($stack$, $maxStackSize$) ::=

if (number of elements in $stack == maxStackSize$) **return** *TRUE*

else return *FALSE*

Stack Push($stack$, $item$) ::=

if (IsFull($stack$)) *stackFull*

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

Boolean IsEmpty($stack$) ::=

if ($stack == \text{CreateS}(maxStackSize)$) **return** *TRUE*

else return *FALSE*

Element Pop($stack$) ::=

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

else remove and return the $item$ at the top of the stack.

■ <Implementation of *Stack*>

Using a one-dimensional array.

Stack CreateS(*maxStackSize*) ::=

```
#define MAX_STACK_SIZE 100  /* maximum stack size */
```

```
typedef struct {
```

```
    int key;
```

```
    /* other fields */
```

```
} element;
```

```
element stack[MAX_STACK_SIZE];
```

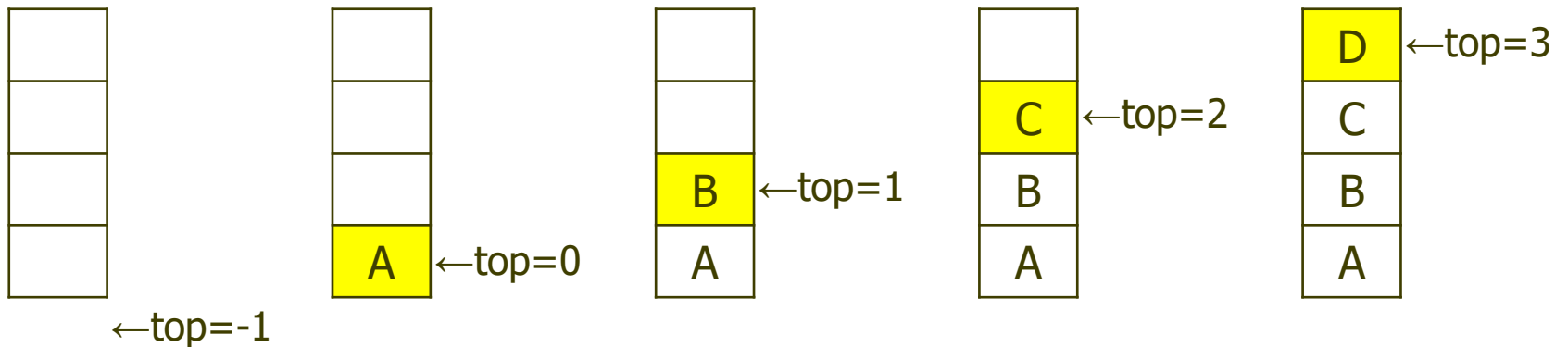
```
int top = -1;
```

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

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

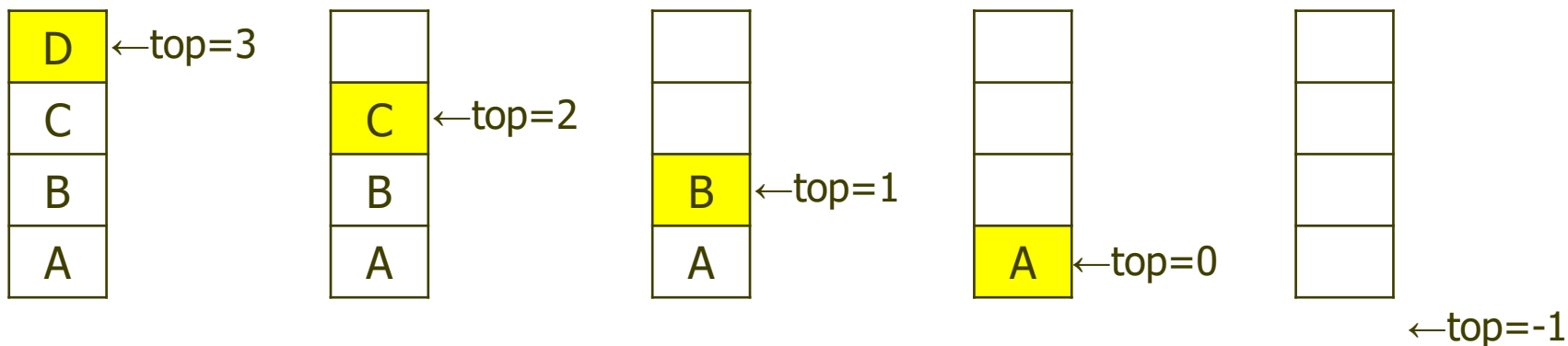
■ [Program 3.1] (Add to a stack)

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



■ [Program 3.2] (Delete from a stack)

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



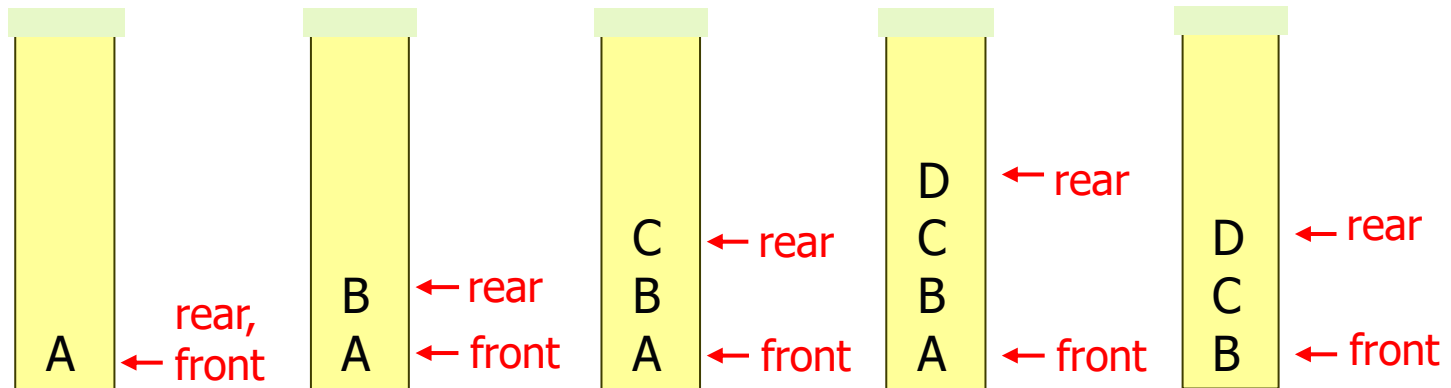
3.3 QUEUES

- A *queue* is an ordered list in which all insertions take place at one end called the *rear* and all deletions take place at the opposite end called the *front*.

$$Q = (a_0, a_1, \dots, a_{n-1})$$

 ↑ ↑
front rear

First-In-First-Out (FIFO) list.



■ [ADT 3.2]: Abstract data type Queue

ADT *Queue* is

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

functions : for all $queue \in Queue$, $item \in element$, $maxQueueSize \in \text{positive integer}$

Queue CreateQ($maxQueueSize$) ::=

create an empty queue whose maximum size is $maxQueueSize$

Boolean IsFullQ($queue$, $maxQueueSize$) ::=

if (number of elements in $queue == maxQueueSize$) **return** *TRUE*

else return *FALSE*

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

if (IsFullQ($queue$)) *queueFull*

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

Boolean IsEmptyQ($queue$) ::=

if ($queue == \text{CreateQ}(maxQueueSize)$) **return** *TRUE*

else return *FALSE*

Element DeleteQ($queue$) ::=

if (IsEmptyQ($queue$)) **return**

else remove and return the $item$ at front of $queue$.

■ <Implementation of *Queue*>

The simplest scheme employs a one-dimensional array and two variables, *front* and *rear*.

The *front* points the position in front of the first element and the *rear* points the position of the last element.

Thus, we can use a simple condition $front == rear$ to check if the queue is empty.

Queue CreateQ(maxQueueSize) ::=

```
#define MAX_QUEUE_SIZE 100 /* maximum queue size */
```

```
typedef struct {
```

```
    int key;
```

```
    /* other fields */
```

```
    } element;
```

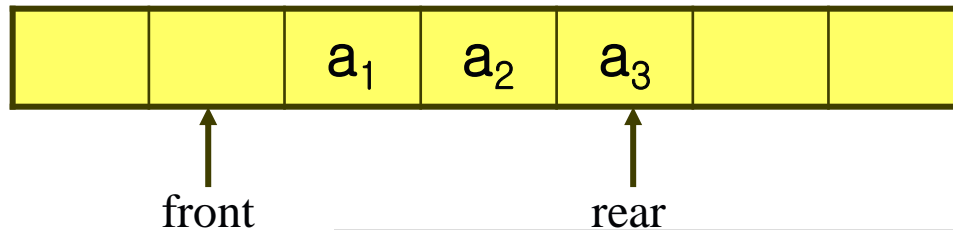
```
element queue[MAX_QUEUE_SIZE];
```

```
int rear = -1;
```

```
int front = -1;
```

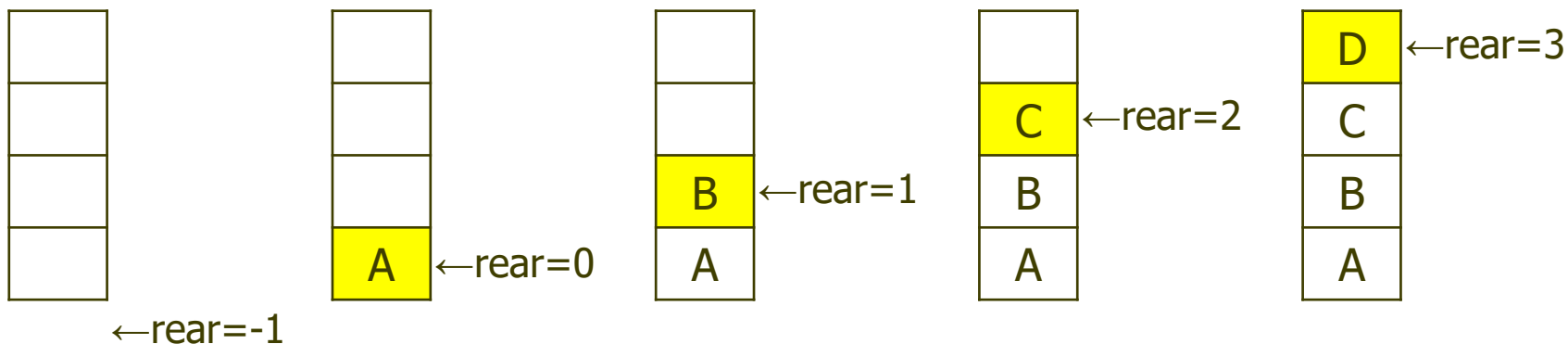
Boolean IsEmptyQ(queue) ::= front == rear;

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



■ [Program 3.5] Add to a queue

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

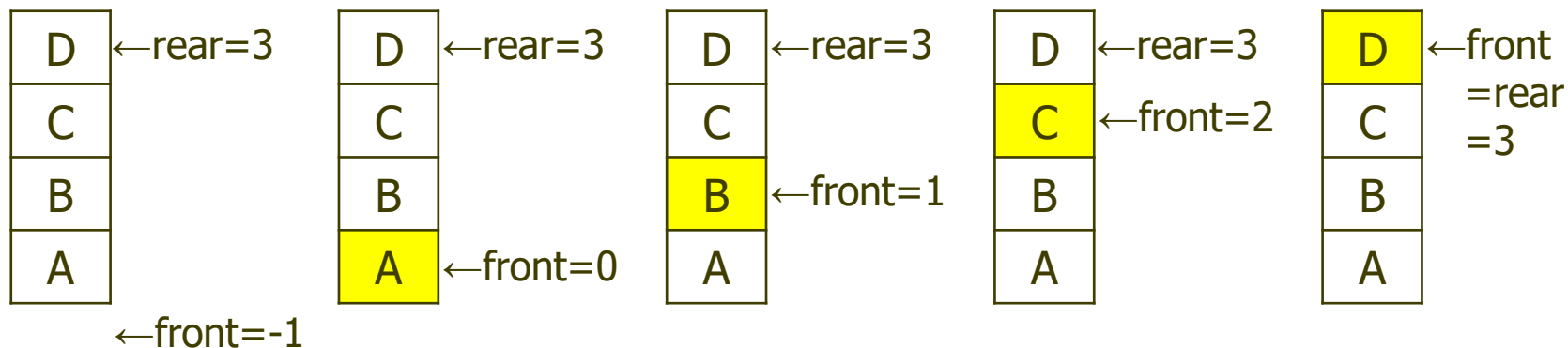


■ [Program 3.6] Delete from a queue

```

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

```



■ Example 3.2 [Job scheduling] :

A job queue by an operating system.

If the operating system does not use priorities, then the jobs are processed in the order they enter the system.

■ [Figure 3.5] : Insertion and deletion from a sequential queue

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job1 is added
-1	1	J1	J2			Job2 is added
-1	2	J1	J2	J3		Job3 is added
0	2		J2	J3		Job1 is deleted
1	2			J3		Job2 is deleted

■ Handling of *queueFull*:

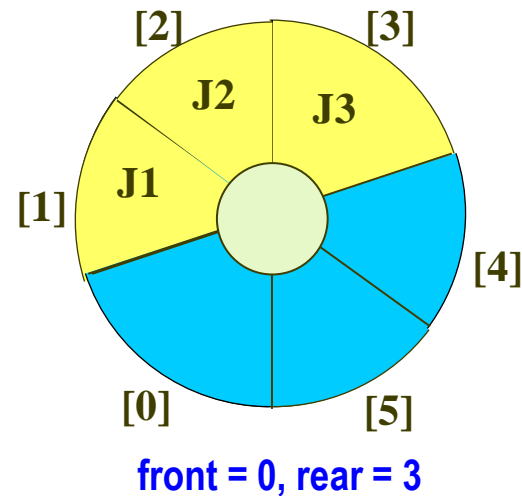
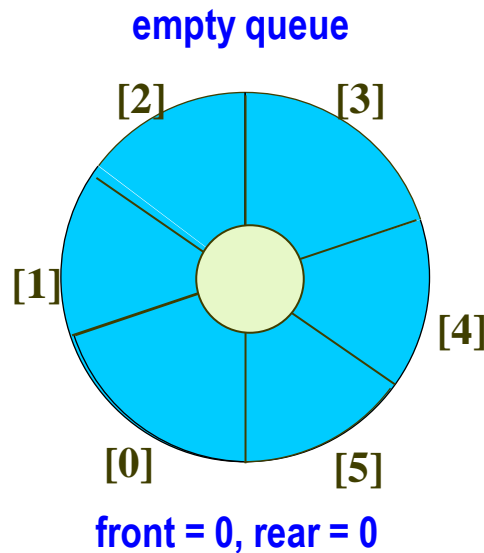
- As jobs enter and leave the system, the queue gradually shifts to the right.
- Eventually the rear index equals $\text{MAX_QUEUE_SIZE} - 1$, suggesting that the queue is full.
- In this case, *queueFull* should move the entire queue to the left so that the first element is again at *queue*[0] and *front* is at -1.
- The *rear* is also recalculated.
- Shifting an array is very time-consuming.

- **A more efficient implementation :**

By regarding the array *queue*[MAX_QUEUE_SIZE] as **circular**.

The *front* index always points one position counterclockwise from the first element in the queue.

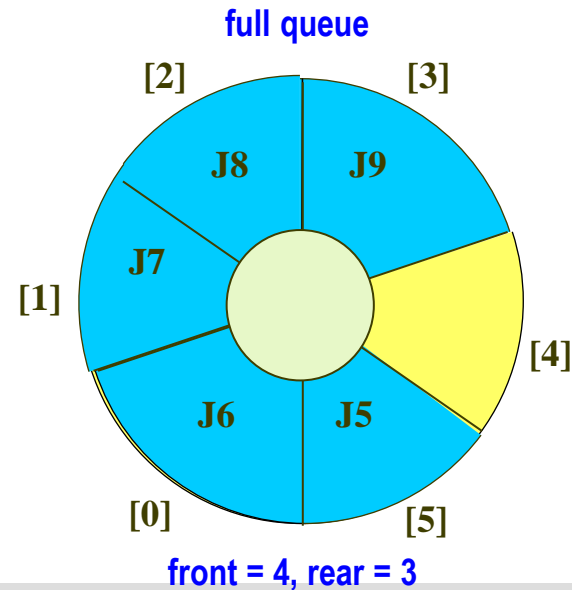
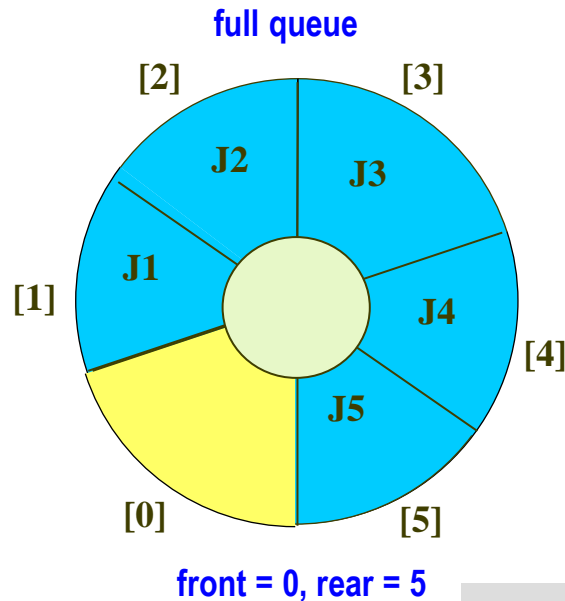
The *rear* index points to the current end of the queue.



- The queue is empty if $front == rear$.

- **When is the queue full?**

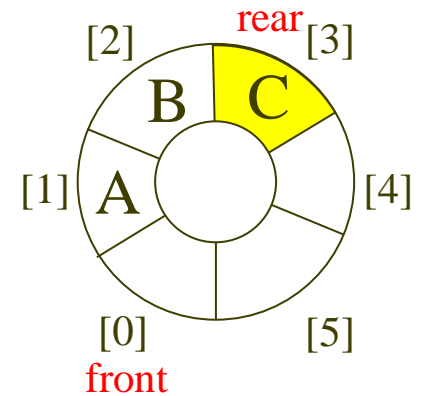
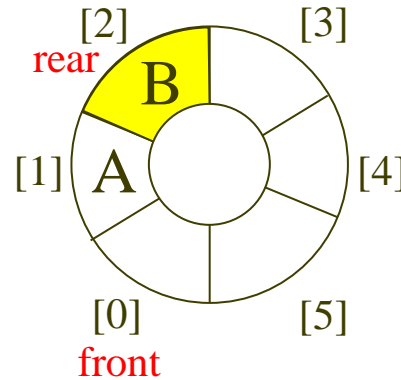
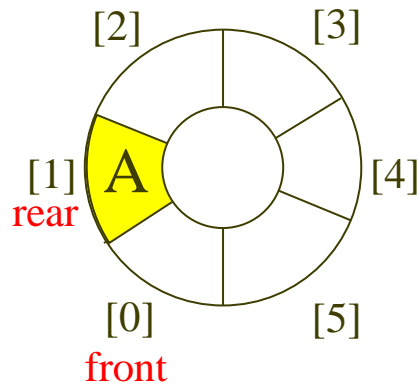
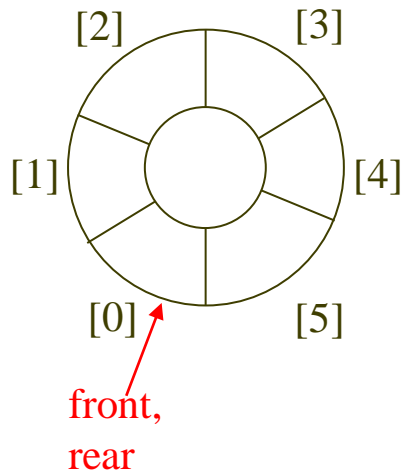
To distinguish between an empty and a full queue, we adapt the convention that a circular queue of size MAX_QUEUE_SIZE will be permitted to hold at most MAX_QUEUE_SIZE-1 elements.



- To implement *addq* and *deleteq* for a circular queue, we must assure that a circular rotation occurs.
- Using the modulus operator :
$$\text{rear} = (\text{rear} + 1) \% \text{MAX_QUEUE_SIZE}; // \text{insertion}$$
$$\text{front} = (\text{front} + 1) \% \text{MAX_QUEUE_SIZE}; // \text{deletion}$$

■ [Program 3.7] : Add to a circular queue

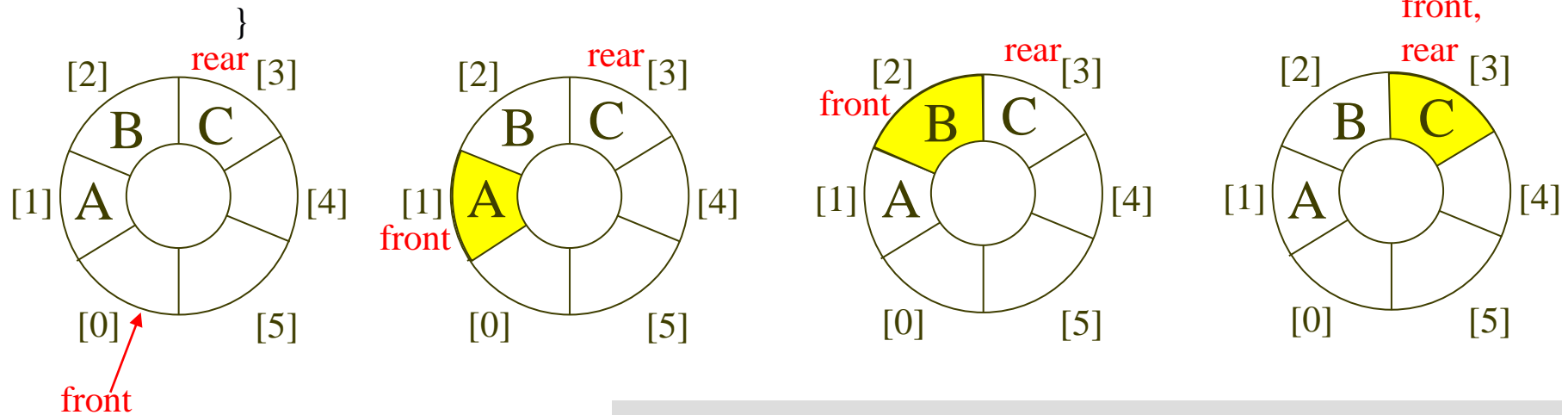
```
void addq(element item)
{ /* add an item to the queue */
    rear = (rear + 1) % MAX_QUEUE_SIZE;
    if (front == rear)
        queueFull(); /* print error and exit */
    queue[rear] = item;
}
```



■ [Program 3.8] : Delete from a circular queue

element deleteq()

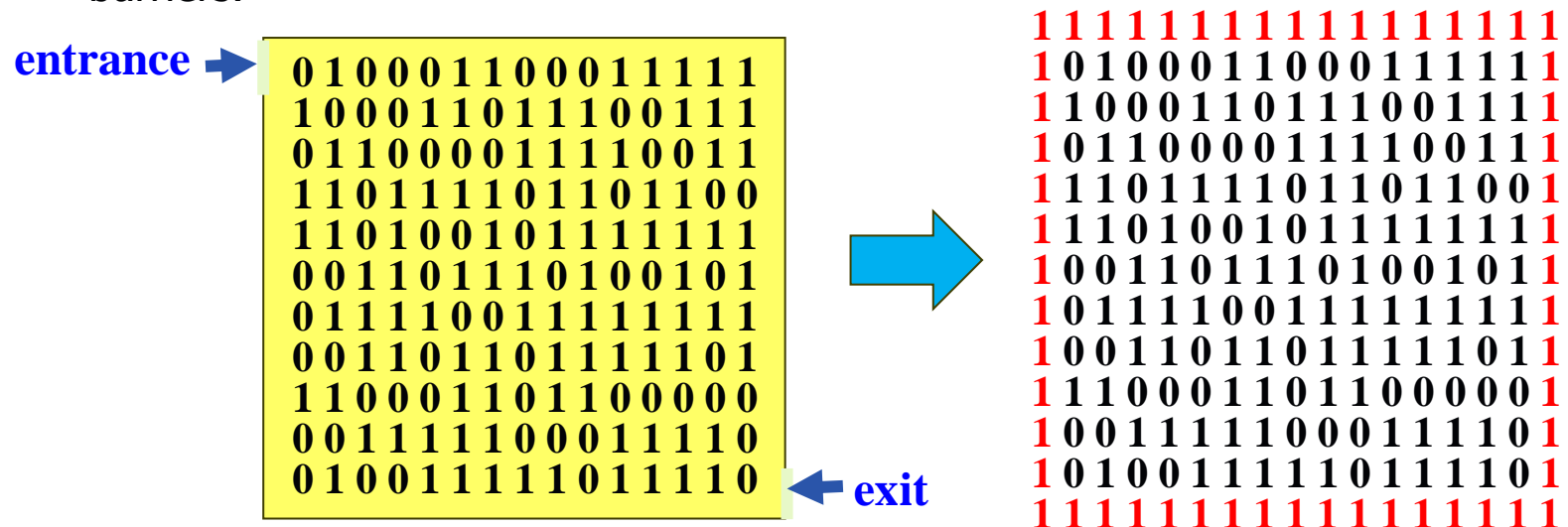
```
{ /* remove element at the front of the queue */
  if (front == rear)
    return queueEmpty(); /* return an error key */
  front = (front + 1) % MAX_QUEUE_SIZE;
  return queue[front];
}
```



3.5 A MAZING PROBLEM

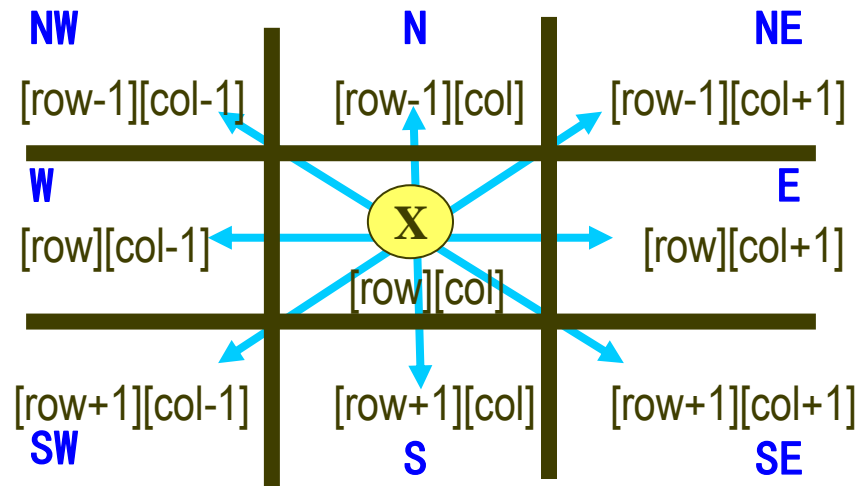
Representation of a maze :

- Two-dimensional array in which 0's represent the open paths and 1's the barriers.



- To avoid checking for the border conditions we surround the maze by a border of 1's. 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 $[m][p]$.

- Predefine the possible directions to move in an array, *move*.

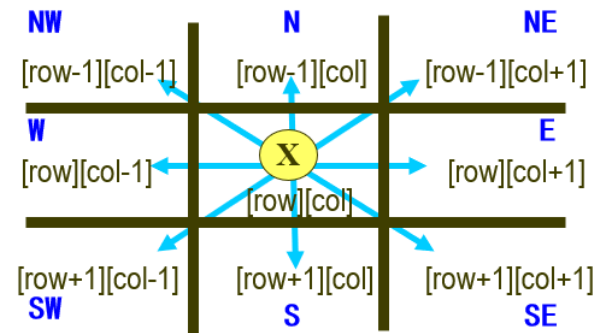


```
typedef struct {  
    short int vert;  
    short int horiz;  
} offsets;
```

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

- **[Figure 3.10]: Table of moves**

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



- If we are at position, $maze[row][col]$, we can find the position of the next move, $maze[nextRow][nextCol]$, by setting
 $nextRow = row + move[dir].vert;$
 $nextCol = col + move[dir].horiz;$
- To record the maze positions already checked, we maintain a second two-dimensional array, *mark*.
- A stack is used to store the positions on the path from the entrance to the current position.

■ [Program 3.11] : Initial maze program

```
initialize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty) {
    /* move to position at the top of stack */
    <row,col,dir> = delete from top of stack;
    while (there are more moves from current position) {
        <nextRow, nextCol> = coordinates of next move;
        dir = direction of move;
        if ((nextRow==EXIT_ROW) && (nextCol==EXIT_COL))
            success;
        if ((maze[nextRow][nextCol]==0) &&(mark[nextRow][nextCol]==0)) {
            /* legal move and haven't been there */
            mark[nextRow][nextCol] = 1;
            /* save current position and direction */
            add <row,col,dir> to the top of the stack;
            row = nextRow; col = nextCol; dir = north;
        }
    }
}
printf("No path found");
```


■ Defining a stack:

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

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

[Figure 3.11] Simple maze with a long path

■ Need to determine a reasonable bound for the stack size.

→ Since, an $m \times p$ maze, can have at most mp zeroes, it is sufficient for the stack to have this capacity.

■ [Program 3.12] : Maze search function

we assumed that arrays, *maze*, *mark*, *move*, *stack*, and constants *EXIT_ROW*, *EXIT_COL*, *TRUE*, *FALSE*, and variable, *top*, are declared as global.

```
void path(void)
{
    /* output a path through the maze if such a path exists */
    int i, row, col, nextRow, nextCol, 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 */
            nextRow = row + move[dir].vert;
            nextCol = col + move[dir].horiz;
            if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
                found = TRUE;
```

```

    else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
        mark [nextRow][nextCol] = 1;
        position.row = row; position.col = col;
        position.dir = ++dir;
        push(position);
        row = nextRow; col = nextCol; 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");
}

```

mark[1][1] = 1; top = 0;
stack[0].row = 1; stack[0].col = 1; stack[0].dir = 1;

i	
row	
col	
nextRow	
nextCol	
dir	
top	0

maze

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

mark

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

stack

row=1	col=1	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 */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	1
nextRow	
nextCol	
dir	1
top	-1

position

row=1	col=1	dir=1
-------	-------	-------

pop

maze

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

mark

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

stack

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	1
nextRow	0
nextCol	2
dir	1
top	-1

position

row=1	col=1	dir=1
-------	-------	-------

maze

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

mark

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

stack

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	1
nextRow	0
nextCol	2
dir	2
top	-1

position

row=1	col=1	dir=1
-------	-------	-------

maze

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

mark

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

stack

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	1
nextRow	1
nextCol	2
dir	2
top	-1

position

row=1	col=1	dir=1
-------	-------	-------

maze

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

mark

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

stack

versity


```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	1
nextRow	1
nextCol	2
dir	3
top	0

position

row=1	col=1	dir=3
-------	-------	-------

push



row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	1
nextCol	2
dir	0
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	0
nextCol	2
dir	0
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	0
nextCol	2
dir	1
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	0
nextCol	3
dir	1
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	0
nextCol	3
dir	2
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	1
nextCol	3
dir	2
top	0

position

row=1	col=1	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	1
nextCol	3
dir	3
top	1

position

row=1	col=2	dir=3
-------	-------	-------

push



row=1	col=2	dir=3
row=1	col=1	dir=3

maze

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

mark

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

stack

versity


```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	3
nextRow	1
nextCol	3
dir	0
top	1

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=3
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	3
nextRow	0
nextCol	3
dir	0
top	1

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=3
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) { (Assuming that all directions are already
    position = pop();         checked)
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	3
nextRow	0
nextCol	2
dir	8
top	1

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=3
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	0
nextCol	2
dir	3
top	0

position

row=1	col=2	dir=3
-------	-------	-------

pop

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	3
dir	3
top	0

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	3
dir	4
top	0

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	2
dir	4
top	0

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	2
dir	5
top	0

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity


```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	1
dir	5
top	0

position

row=1	col=2	dir=3
-------	-------	-------

maze

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

mark

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

stack

row=1	col=1	dir=3
-------	-------	-------

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	1
col	2
nextRow	2
nextCol	1
dir	6
top	1

position

row=1	col=2	dir=6
-------	-------	-------

push



row=1	col=2	dir=6
row=1	col=1	dir=3

maze

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

mark

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

stack

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	2
nextCol	1
dir	0
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	1
nextCol	1
dir	0
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	1
nextCol	1
dir	1
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	1
nextCol	2
dir	1
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	1
nextCol	2
dir	2
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	2
nextCol	2
dir	2
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity


```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	2
nextCol	2
dir	3
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	3
nextCol	2
dir	3
top	1

position

row=1	col=2	dir=6
-------	-------	-------

maze

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

mark

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

stack

row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	2
col	1
nextRow	3
nextCol	2
dir	4
top	2

position

row=2	col=1	dir=4
-------	-------	-------

push
↓

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	3
nextCol	2
dir	0
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	2
nextCol	2
dir	0
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	2
nextCol	2
dir	1
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	2
nextCol	3
dir	1
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	2
nextCol	3
dir	2
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity


```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	3
nextCol	3
dir	2
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

```

while (top > -1 && !found) {
    position = pop();
    row = position.row; col = position.col, dir = position.dir;
    while (dir < 8 && !found) {
        /* move in direction dir */
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].horiz;
        if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
        else if (!maze[nextRow][nextCol] && !mark [nextRow][nextCol]) {
            mark [nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
        }
        else ++dir;
    }
}

```

i	
row	3
col	2
nextRow	3
nextCol	3
dir	2
top	2

position

row=2	col=1	dir=4
-------	-------	-------

maze

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

mark

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

stack

row=2	col=1	dir=4
row=1	col=2	dir=6
row=1	col=1	dir=3

versity

■ Analysis of *path* :

The size of the maze determines the computing time of *path*.

Since each position within the maze is visited no more than once, the worst case time complexity of the algorithm is $O(mp)$ where m and p are, respectively, the number of rows and columns of the maze.

3.6 EVALUATION OF EXPRESSIONS

3.6.1 Expressions

- The representation and evaluation of expressions is of great interest to computer scientists.
- For examples :
 - $((rear+1 == front) \parallel ((rear == MAX_QUEUE_SIZE-1) \&\& ! front))$
 - $x = a/b - c + d * e - a * c$
- Tokens in expressions :
 - operators
 - operands
 - parentheses

- For $x = a/b - c + d * e - a * c$, when $a=4$, $b=c=2$, $d=e=3$, what is the value of x ?

$$\begin{aligned} & ((4 / 2) - 2) + (3 * 3) - (4 * 2) \\ &= 0 + 9 - 8 \\ &= 1 \end{aligned}$$

or

$$\begin{aligned} & (4 / (2 - 2 + 3)) * (3 - 4) * 2 \\ &= 4 / 3 * (-1) * 2 \\ &= -2.666... \end{aligned}$$

- If we wanted the second answer, we would have written it as follows,
 $x = (a / (b - c + d)) * (e - a) * c$.

- Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators.
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first.
- See Figure 3.12 for precedence hierarchy for C

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
> >= < <=	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

The associativity column indicates how we evaluate operators with the same precedence.

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

Figure 3.12: Precedence hierarchy for C

■ Ways of writing expressions :

Infix notation - $a * b$

Prefix notation - $* a b$

Postfix notation - $a b *$

} parenthesis-free notations

Although *infix* notation is the most common way of writing expressions, it is not the one used by compilers to evaluate expressions.

Compilers typically use a parentheses-free notation referred to as *postfix*

Infix	Postfix
$2+3*4$	$234*+$
$a*b+5$	$ab*5+$
$(1+2)*7$	$1\ 2+7*$
$a*b/c$	$ab*c/$
$((a/(b-c+d))*(e-a)*c$	$abc-d+ / ea-*c*$
$a/b-c+d*e-a*c$	$ab/c-de*+ac*-$

[Figure 3.13] Infix and postfix notation

3.6.2 Evaluating Postfix Expressions

- To evaluate a postfix expression, we make a single left-to-right scan of it.
 - 1) Place the operands on a stack until we find an operator.
 - 2) Remove, from the stack, the correct number of operands for the operator.
 - 3) Perform the operation
 - 4) Place the result back on the stack.

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

- To simplify our task, we assume that the expression contains only the binary operators

+, -, *, /, and %

and that the operands in the expression are single digit integers.

- **The complete declarations :**

```
#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 */  
int top = -1;
```

■ [Program 3.14] : Function to get a token from the input string

```
precedence getToken(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 */
    }
}
```

■ [Program 3.14] : Function to get a token from the input string

```
precedence getToken(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 */
    }
}
```

■ [Program 3.13] : Function to evaluate a postfix expression

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a global variable. '\0' is the end of the
   expression. The stack and top of the stack are global variables. getToken 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 */

token = getToken(&symbol, &n);
```

```

while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */
}

```

```
precedence token;  
char symbol;  
int op1, op2;  
int n = 0; /* counter for the expression string */
```

token	
symbol	
op1	
op2	
n	0
top	-1

initial state

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```
token = getToken(&symbol, &n);
```

```
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */
```

expr: 62/34-*

token	8
symbol	'6'
op1	
op2	
n	1
top	-1

stack

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'6'
op1	
op2	
n	1
top	0

stack

6

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'2'
op1	
op2	
n	2
top	0

stack

6

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'2'
op1	
op2	
n	2
top	1

stack

2
6

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	5
symbol	'/'
op1	
op2	
n	3
top	1

stack

2
6

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	5
symbol	'/'
op1	6
op2	2
n	3
top	-1

stack

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	5
symbol	'/'
op1	6
op2	2
n	3
top	0

stack

3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'3'
op1	6
op2	2
n	4
top	0

stack

3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'3'
op1	6
op2	2
n	4
top	1

stack

3
3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'4'
op1	6
op2	2
n	5
top	1

stack

3
3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	8
symbol	'4'
op1	6
op2	2
n	5
top	2

4
3
3

stack

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	3
symbol	'-'
op1	6
op2	2
n	6
top	2

4
3
3

stack

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	3
symbol	'-'
op1	3
op2	4
n	6
top	0

stack

3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	3
symbol	'-'
op1	3
op2	4
n	6
top	1

stack

-1
3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	4
symbol	'*'
op1	3
op2	4
n	7
top	1

stack

-1
3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	4
symbol	'*'
op1	3
op2	-1
n	7
top	-1

stack

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	4
symbol	'*'
op1	3
op2	-1
n	7
top	0

stack

-3

precedence

0	1	2	3	4	5	6	7	8
()	+	-	*	/	%	eos	operand

```

token = getToken(&symbol, &n);
while (token != eos)    { → break
    if (token == operand)
        push(symbol-'0'); /* stack insert */
    else {
        /* remove two operands, perform operation, and return result to the stack */
        op2 = pop(); /* stack delete */
        op1 = pop();
        switch (token) {
            case plus : push(op1+op2); break;
            case minus : push(op1-op2); break;
            case times : push(op1*op2); break;
            case divide: push(op1/op2); break;
            case mod : push(op1%op2);
        }
    }
    token = getToken(&symbol, &n);
}
return pop(); /* return result */

```

expr: 62/34-*

token	7
symbol	eos
op1	3
op2	-1
n	8
top	0

stack

-3

3.6.3 Infix To Postfix

- An algorithm for producing a postfix expression from an infix one :
 - 1) Fully parenthesize the expression.
 - 2) Move all binary operators so that they replace their corresponding right parenthesis.
 - 3) Delete all parentheses.

- For example, $a/b-c+d*e-a*c$

After step 1,

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

Performing step 2 and 3,

$ab/c-de*+ac*-$

1. Read the expression and parenthesize it.
2. Move the operators.

- Although this algorithm works well when done by hand, it is inefficient on a computer because it requires two passes.

- Note that the order of **operands** is the same in infix and postfix.
- Thus we can form the postfix equivalent by scanning the infix expression left-to-right.
However, the order in which the **operators** are output depends on their precedence.
- Since we must output the higher precedence operators first, we save operators until we know their correct placement.
A stack is one way of doing this, but removing operators correctly is problematic.

- Method:

Operators with higher precedence must be output *before* those with lower precedence. Therefore, we stack operators as long as the precedence of the operator at the top of the stack is *less than* the precedence of the incoming operator.

- Parenthesized expression:

Stack "(" and operators until we reach ")". At this point we unstack until we reach the corresponding "(" . Then delete "(" from the stack. (The right parenthesis is never put on the stack.)

■ **[Figure 3.15] Translation of $a+b*c$ to postfix**

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

■ **[Figure 3.16] Translation of $a*(b+c)/d$ to postfix**

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+
/	/			0	abc**
d	/			0	abc**d
<i>eos</i>	/			0	abc**d/

- The analysis of the two examples suggests a precedence-based scheme for stacking and unstacking operators.
- We need two types of precedence, an *in-stack precedence* (*isp*) and an *incoming precedence* (*icp*).
 - Remove an operator from the stack only if its *isp* \geq *icp* of the new operator.

```
precedence stack[MAX_STACK_SIZE];  
/* isp and icp arrays -- index is value of precedence  
lparen, rparen, plus, minus, times, divide, mod, eos */  
int isp[] = {0, 19, 12, 12, 13, 13, 13, 0};  
int icp[] = {20, 19, 12, 12, 13, 13, 13, 0};
```

■ [program 3.15] : Function to convert from infix to postfix

```
void postfix(void)
{
    /* output the postfix of the expression. The expression string, stack, and the top are global */
    char symbol;
    int n = 0;
    precedence token;

    stack[0] = eos;
    for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
        if (token == operand)
            printf("%c", symbol);
        else if (token == rparen) {
            /* unstack tokens until left parenthesis */
            while (stack[top] != lparen)
                printToken(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])
        printToken(pop());
    push(token);
}
}
while ( (token = pop()) != eos)
    printToken(token);
printf("\n");
}
```



```
char symbol;  
int n = 0;  
precedence token;  
  
stack[0] = eos;
```

token	
symbol	
n	0
top	0

stack eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output:

token	8
symbol	'a'
n	1
top	0

stack

eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: a

token	8
symbol	'a'
n	1
top	0

stack

eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: a

token	4
symbol	'*'
n	2
top	0

stack

eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: a

token	4
symbol	'*'
n	2
top	0

stack

eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: a

token	4
symbol	'*
n	2
top	1

stack

'*
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: a

token	0
symbol	'('
n	3
top	1

stack

'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: a

token	0
symbol	'('
n	3
top	1

stack

'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: a

token	0
symbol	'('
n	3
top	2

stack

'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: a

token	8
symbol	'b'
n	4
top	2

stack

'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: ab

token	8
symbol	'b'
n	4
top	2

stack

'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: ab

token	2
symbol	'+'
n	5
top	2

stack

'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: ab

token	2
symbol	'+'
n	5
top	2

stack

'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: ab

token	2
symbol	'+'
n	5
top	3

stack

'+'
'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: ab

token	8
symbol	'c'
n	6
top	3

stack

'+'
'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: abc

token	8
symbol	'c'
n	6
top	3

stack

'+'
'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: abc

token	5
symbol	'/'
n	7
top	3

stack

'+'
'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: abc

token	5
symbol	'/'
n	7
top	3

stack

'+'
'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: abc

token	5
symbol	'/'
n	7
top	4

stack

'/'
'+'
'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: abc

token	8
symbol	'd'
n	8
top	4

stack

'/'
'+'
'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: abcd

token	8
symbol	'd'
n	8
top	4

stack

'/'
'+'
'('
'*'
eos

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

```
for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
```

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

```
        printf("%c", symbol);
```

```
    else if (token == rparen) {
```

```
        /* unstack tokens until left parenthesis */
```

```
        while (stack[top] != lparen)
```

```
            printToken(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])
```

```
            printToken(pop());
```

```
        push(token);
```

```
    }
```

```
}
```

expr: a*(b+c/d)

output: abcd

token	1
symbol	'\')
n	9
top	4

stack

'/'
'+'
'('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: abcd/

token	1
symbol	') '
n	9
top	3

stack

' + '
' ('
' * '
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

expr: a*(b+c/d)

output: abcd/+

token	1
symbol	'\')
n	9
top	2

stack

'\('
'*'
eos

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

```

for (token = getToken(&symbol, &n); token != eos; token = getToken(&symbol, &n)) {
    if (token == operand)
        printf("%c", symbol);
    else if (token == rparen) {
        /* unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            printToken(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])
            printToken(pop());
        push(token);
    }
}

```

→break

expr: a*(b+c/d)

output: abcd/+

token	1
symbol	'\')
n	9
top	1

stack

'*'
eos

expr: a*(b+c/d)

output: abcd/+*

```
while ( (token = pop()) != eos)
    printToken(token);
```

token	1
symbol	'\y'
n	9
top	1

stack

eos

■ Analysis of *postfix* :

Let n be the number of tokens in the expression.

$\Theta(n)$ time is spent extracting tokens and outputting them.

$\Theta(n)$ time is spent in two *while* loops

as the number of tokens that get stacked and unstacked is linear in n .

So, the time complexity of function *postfix* is $\Theta(n)$.