Chapter 3
Stacks and Queues

2024 Spring

Ri Yu

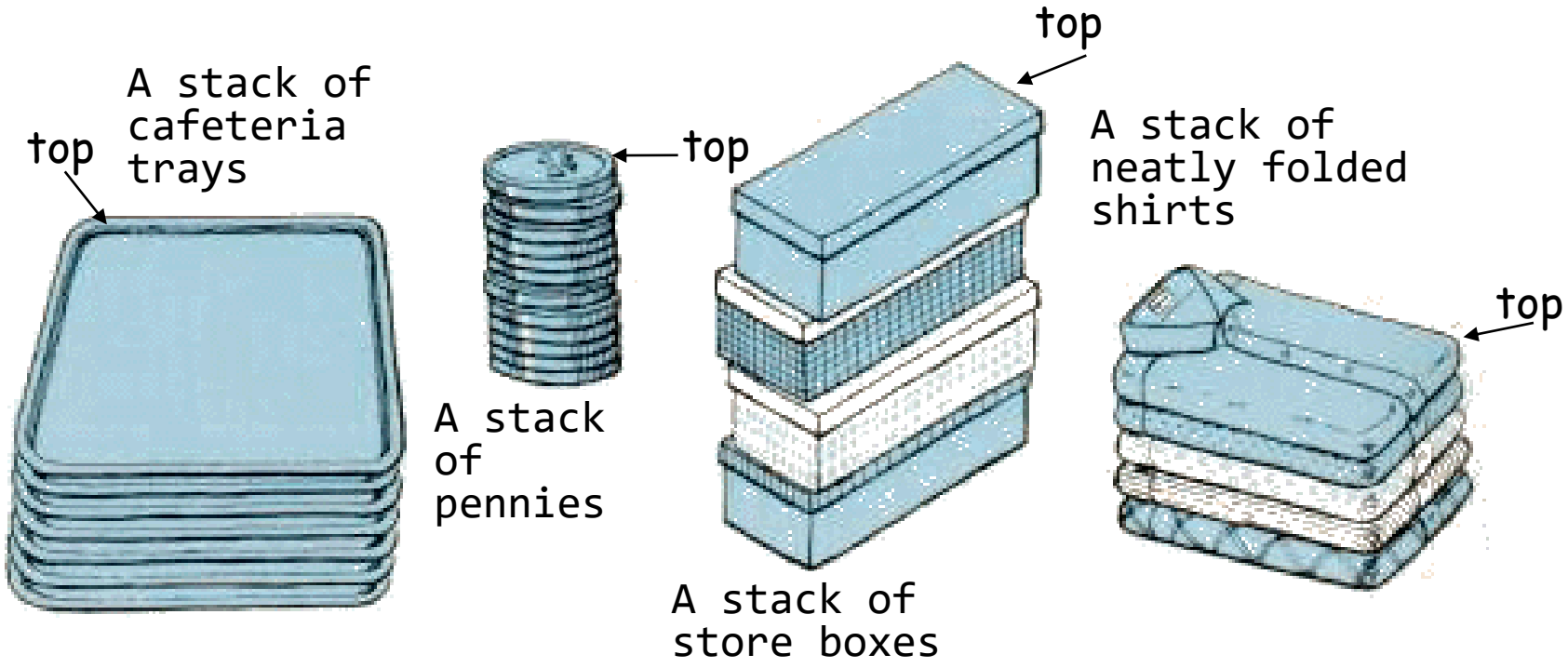
Ajou University

Contents

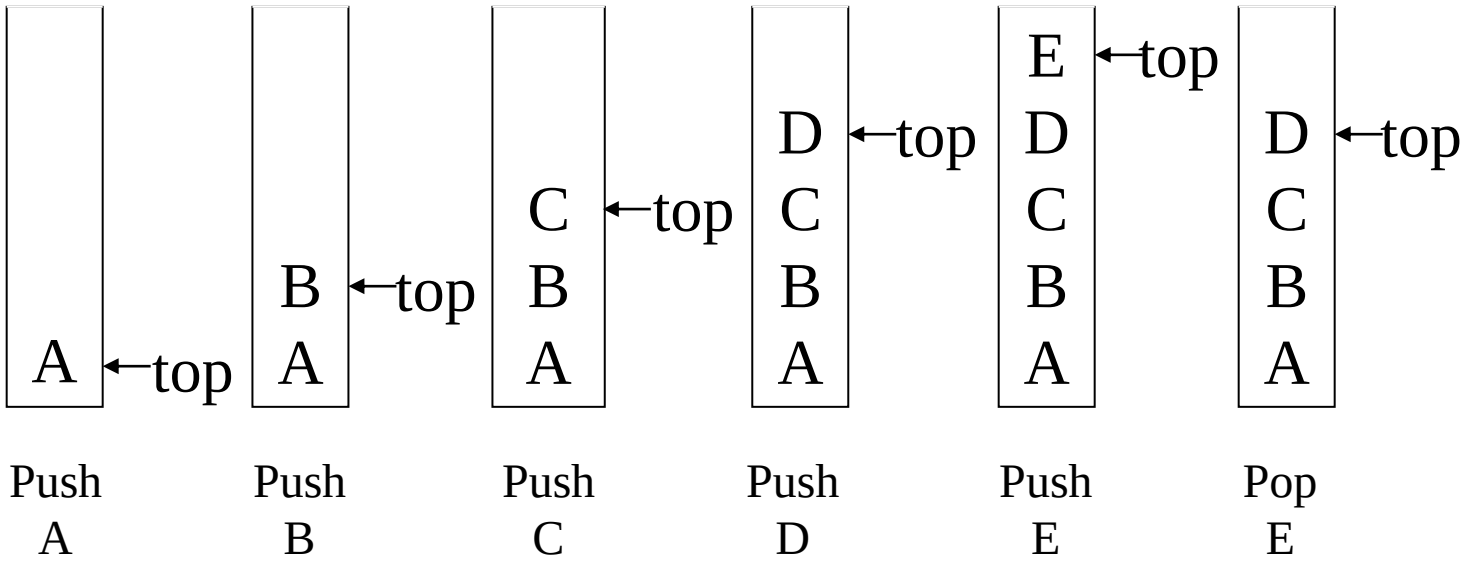
Stack

- Definition: an ordered list in which insertions (also called push ed and adds) and deletions (also called pops and removes) are made **at one end called the *top***
- Given a stack $S=(a_0,\ldots,a_{n-1})$
  - a_0 is bottom element
  - a_{n-1} is top element
  - a_i is on top of element a_{i-1}, $0<i<n$
- *Last-In-First-Out (LIFO)*
  - Insert the new element into the stack on the top end
  - We can only delete and get the top element of the stack

Examples of Stack



Examples of Stack

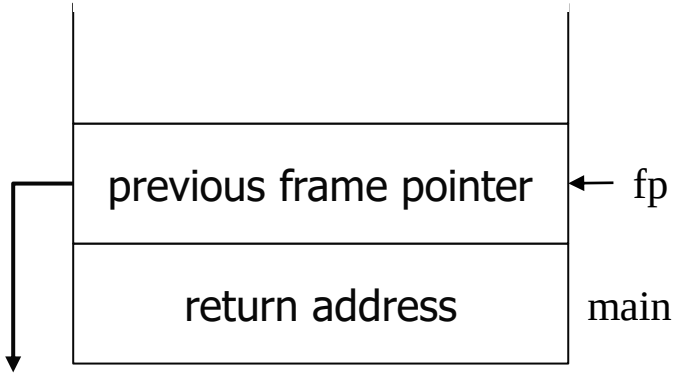


System Stack

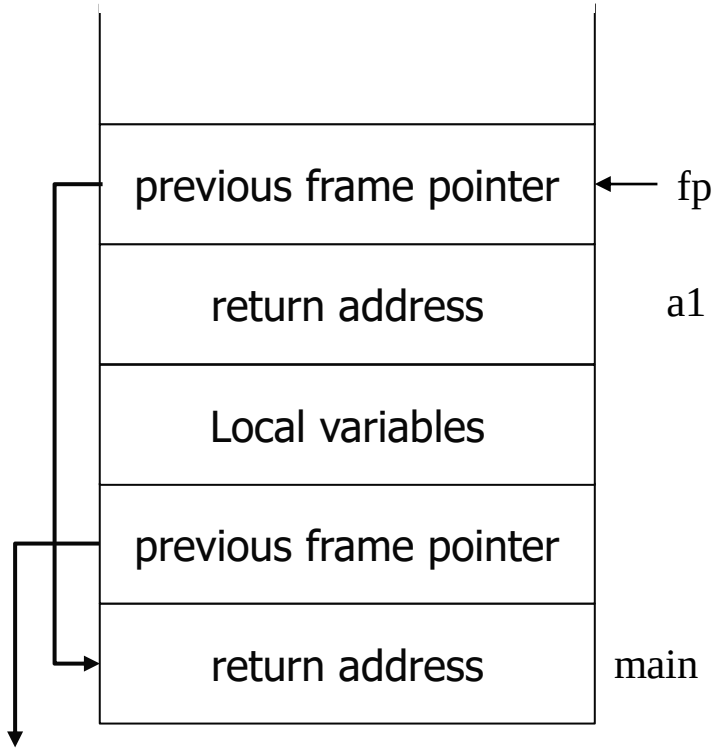
- Used by a program at run-time to process **function calls**.
- A program places an activation record or a **stack frame** on top of the system stack when it invokes a function.
- The previous stack (old) frame pointer points to the stack frame of the invoking function.
- The return address contains the location of the statement to be executed after the function terminates.
- If the function invokes another function, the local variables, except those declared static, and the parameters of the invoking function are added to its stack frame.

System Stack

- System stack after function call
  - fp: a pointer to current stack frame



(a) system stack before a1 is invoked



(b) system stack after a1 is invoked

Stack ADT

ADT *Stack* is
  objects: a finite ordered list with zero or more elements
  functions:
   for all *stack*∈*Stack*, *item*∈*element*, *maxStackSize*∈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* == CreateS(*maxStackSize*))
            return TRUE
            else return FALSE
  Element Pop(*stack*) ::=
             if(IsEmpty(*stack*)) **return**
             else remove and **return** the item on the top of the stack

Stack Implementation

- Use a one-dimensional array, stack[MAX_STACK_SIZE], where MAX_STACK_SIZE is the maximum number of entries

```
#define MAX_STACK_SIZE 100  /* maximum stack size */
typedef struct {
         int key;
         /* other fields */
} element;
element stack[MAX_STACK_SIZE];
int top = -1;                          /* denotes an empty stack */
void push( element item ) {
         /* add an item to the global stack */
         if ( top >= MAX_STACK_SIZE-1 )
                  stackFull();
         stack[++top] = item;
}
element pop() {
         /* delete and return the top element from the stack */
         if ( top == -1 )
                  return stackEmpty();        /* returns an error key */
         return stack[top--];
}
void stackFull() {
         fprintf( stderr, "Stack is full, cannot add element" );
         exit( EXIT_FAILURE );
}
```

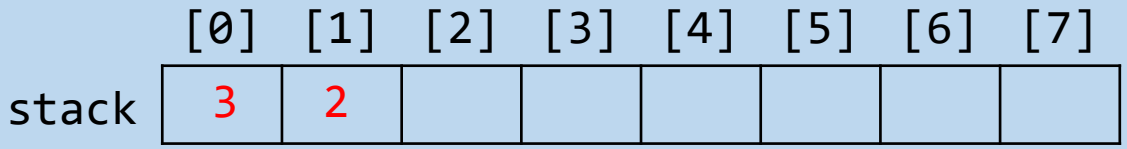
Example

```
main()
{
  element e,f;
  top = -1
  e.key=3;   push(e);
  e.key=2;   push(e);
  f=pop();
  printf (“%d  %d\n”, top, f.key);
}
```

top = 0

top = 1

top = 0



console

```
0    2
```

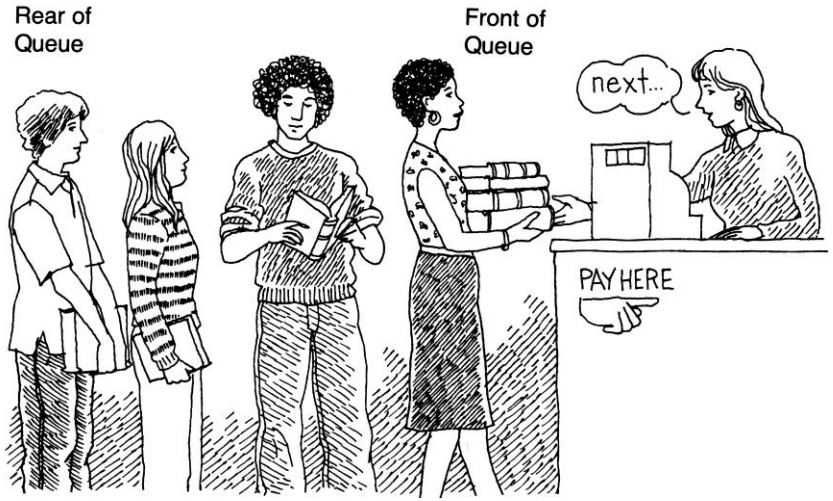
Queue

- Definition: an ordered list in which insertions (also called additi ons, puts, and pushes) and deletions (also called removals and pops) take place **at different ends**
  - all **insertions** take place one end, called the **rear**
  - all **deletions** take place at the opposite end, called the **front**

- *First-In-First-Out (FIFO) list: the first element inserted into a queue is the first element removed*
  - Insert the new element into the queue on the rear side
  - We can only delete/get the front element of the queue

Examples of Queue

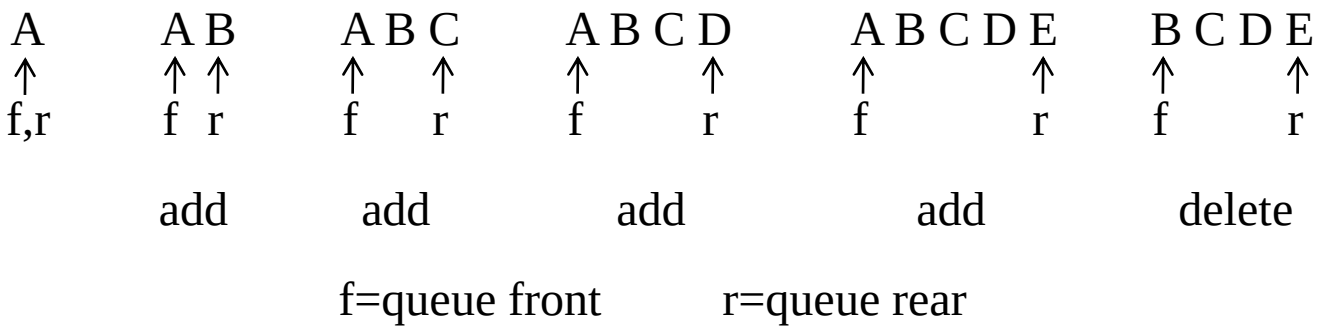
- In a shop, checkout line



- Job scheduling
  - Frequently used in computer programming
  - Job queue by an operating system
  - The jobs are processed in the order they enter the system

Queue

- Inserting and deleting elements in a queue
  - Insert the elements A, B, C, D, E, in that order
  - A is the first element we delete from the queue



Queue ADT

ADT *Queue* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *queue*∈*Queue*, *item*∈*element*, *maxQueueSize*∈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* ==CreateQ(*maxQueueSize*))
        return TRUE
        else return FALSE
    Element DeleteQ(*queue*) ::=
        if (IsEmptyQ(*queue*)) **return**
        else remove and **return** the *item* at front of queue

Queue Implementation

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

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

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

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

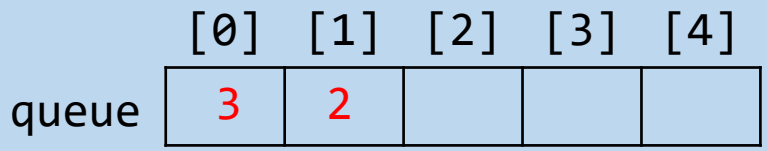
Example

```
main()
{
   element e,f;
   rear = front =-1
   e.key=3;    addq(e);
   e.key=2;    addq(e);
   f=deleteq();
   printf("%d %d %d\n", front, rear, f.key);
}
```

rear = 0, front = -1

rear = 1, front = -1

rear = 1, front = 0



console

```
0    1    3
```

Job Scheduling

- The creation of 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

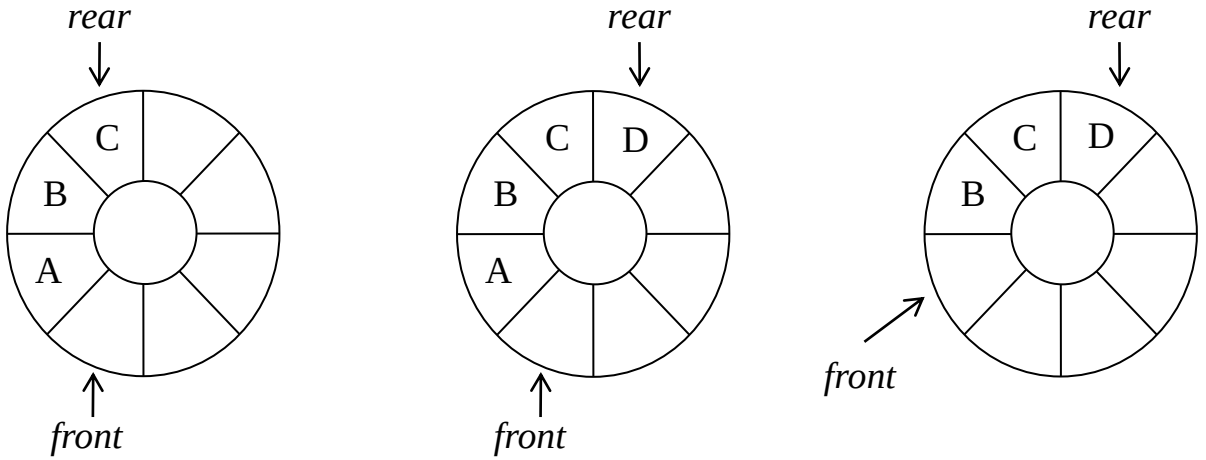
front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Insertion and deletion from a sequential queue

- As jobs enter and leave the system, the queue gradually shifts to the right
  - *queueFull* should move the entire queue to the left
    - Shifting an array is very time-consuming

Circular Queue

- Regard the array queue[*MAX_QUEUE_SIZE*] as circular
- The *front* index always points one position counterclockwise from the location of the front element in the queue
- The *rear* index points to the current end of the queue
- The position next to *MAX_QUEUE_SIZE*-1 is 0, and the position that precedes 0 is *MAX_QUEUE_SIZE*-1

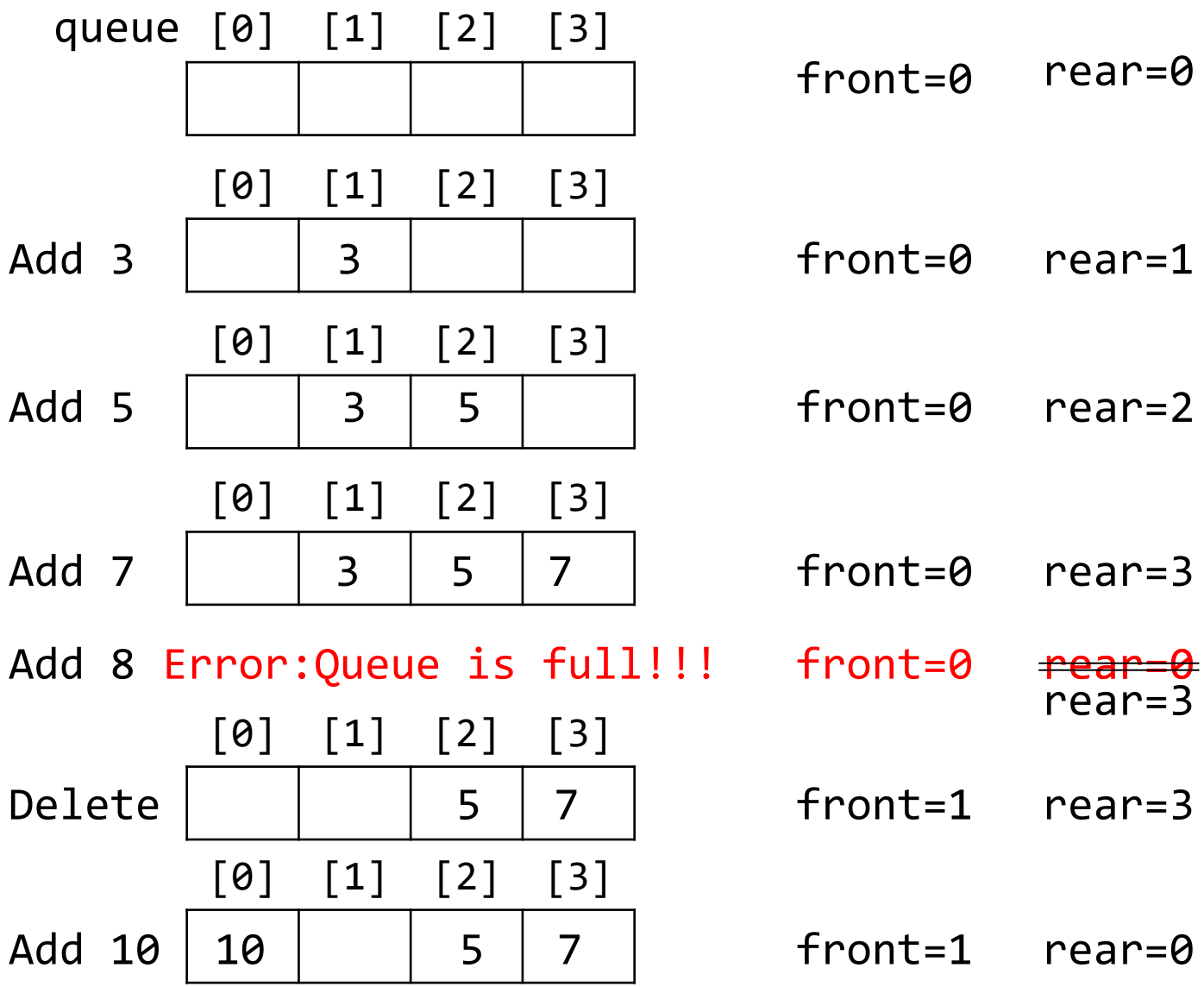


(a) Initial **(b) Addition** **(c) Deletion**

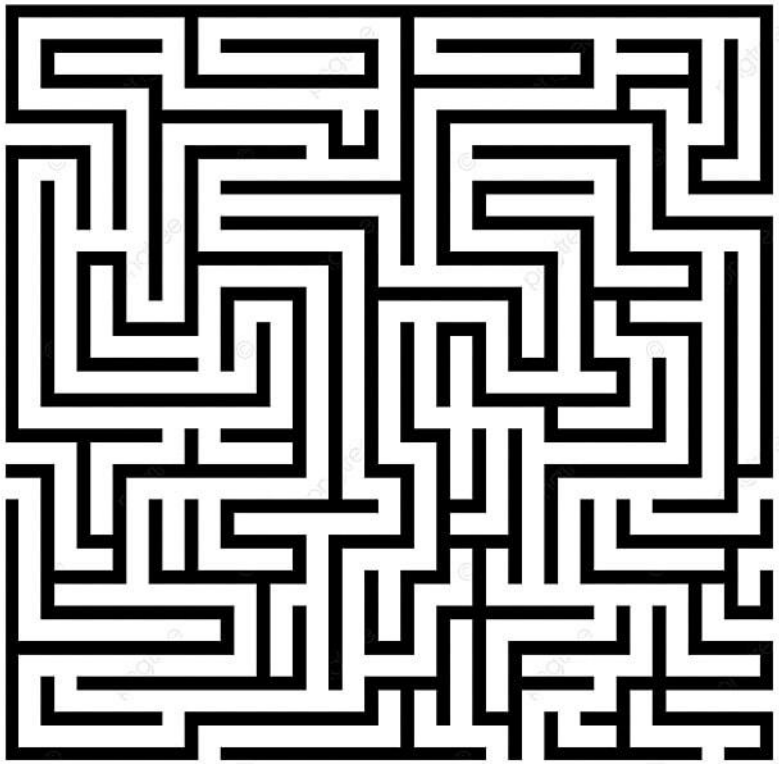
Circular Queue Implementation

```
void addq( element item )
{       /* add an item to the queue */
        rear = (rear+1) % MAX_QUEUE_SIZE;

        /* permitting a maximum of MAX_QUEUE_SIZE-1 rather than MAX_QUEUE_SIZE elements*/
        if ( front == rear )
                queueFull();     /* print error and exit */
        queue[rear] = item;
}

element deleteq()
{       /* remove front element from the queue */
        if( front == rear )
                return queueEmpty();
        front = (front+1) % MAX_QUEUE_SIZE;
        return queue[front];
}
```

Example


queue [0] [1] [2] [3]
front=0 rear=0
[0] [1] [2] [3]
Add 3
3
front=0 rear=1
[0] [1] [2] [3]
Add 5
3 5
front=0 rear=2
[0] [1] [2] [3]
Add 7
3 5 7
front=0 rear=3
Add 8 Error:Queue is full!!!
front=0
rear=0
rear=3
[0] [1] [2] [3]
Delete
5 7
front=1 rear=3
[0] [1] [2] [3]
Add 10
10 5 7
front=1 rear=0

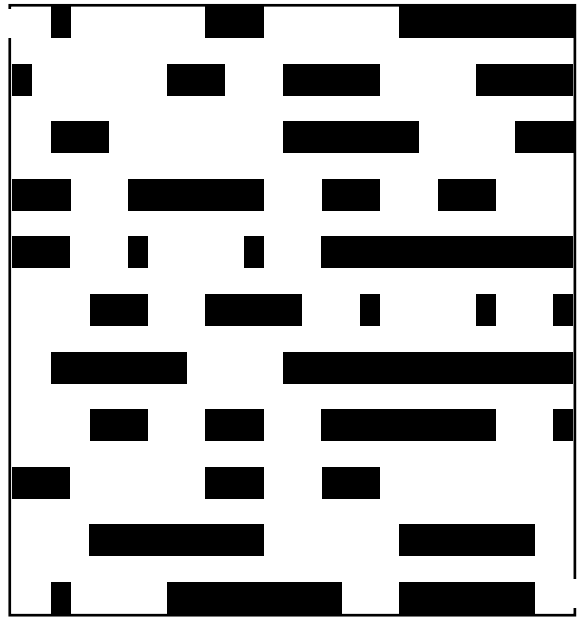
Maze



https://www.youtube.com/watch?v=KfTKT8Zp5r8

Maze

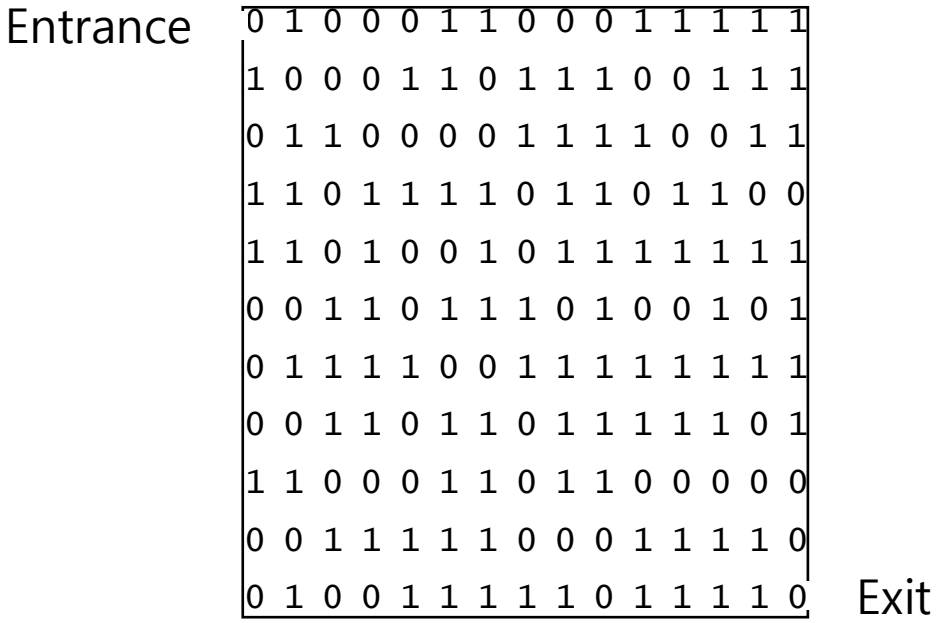
Entrance



Exit

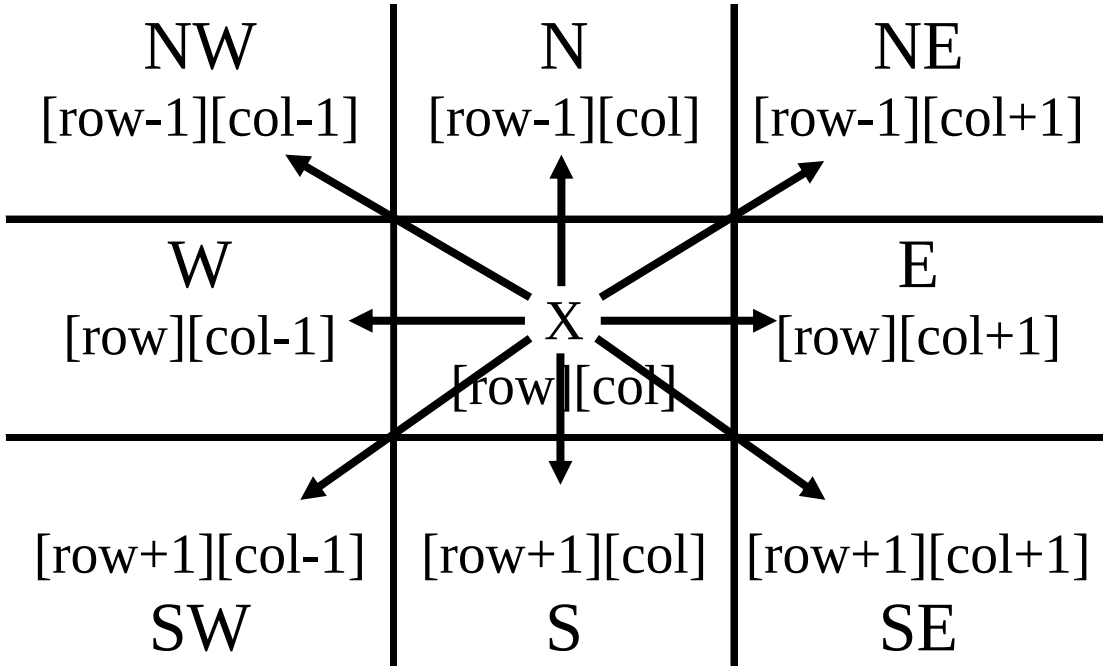
A Mazing Problem

- Representation of the maze
  - The most obvious choice:
    a two-dimensional array
    - 0s: the open paths
    - 1s: the barriers



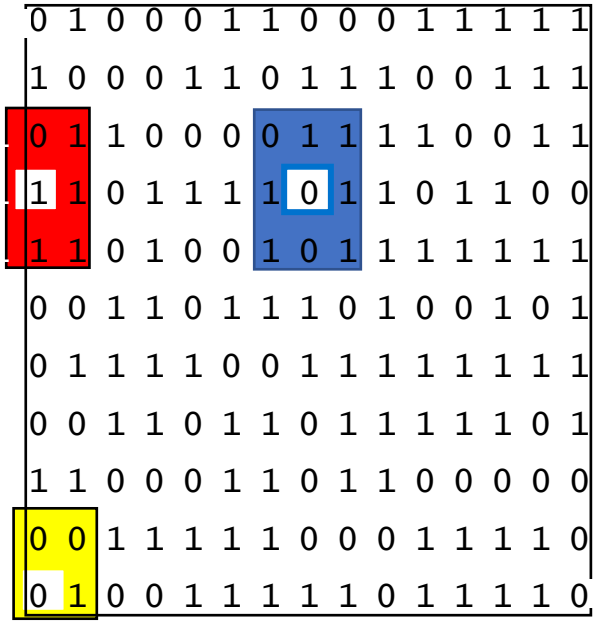
A Mazing Problem

- Let *X* denote the current location, *maze*[*row*][*col*]
  - Possible moves



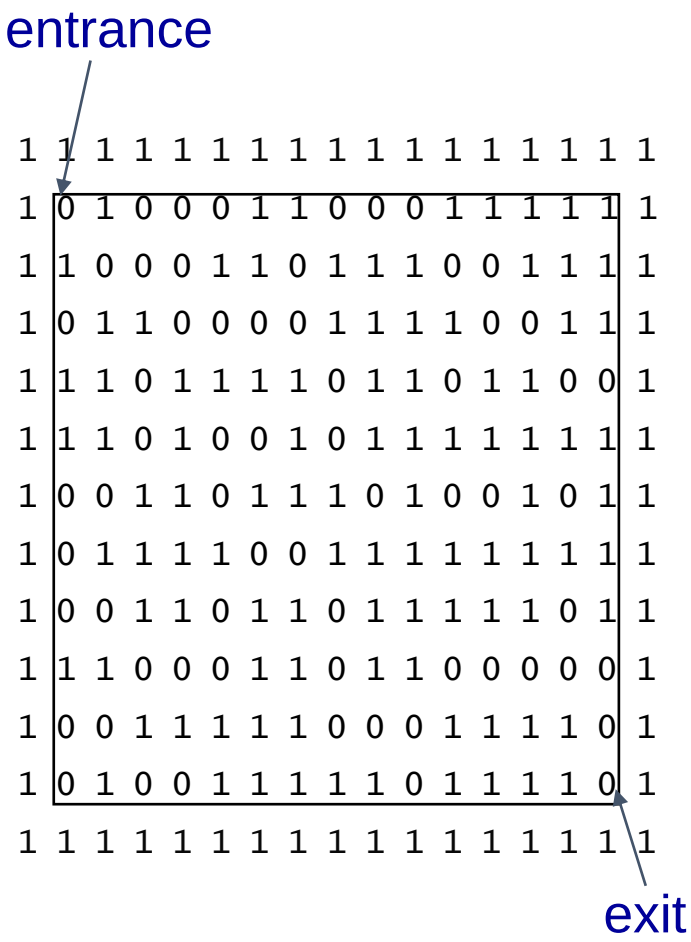
A Mazing Problem

- Representation of the maze
  - Not every position has eight neighbors.
    - Blue box: 8
    - Red box: 5
    - Yellow box : 3

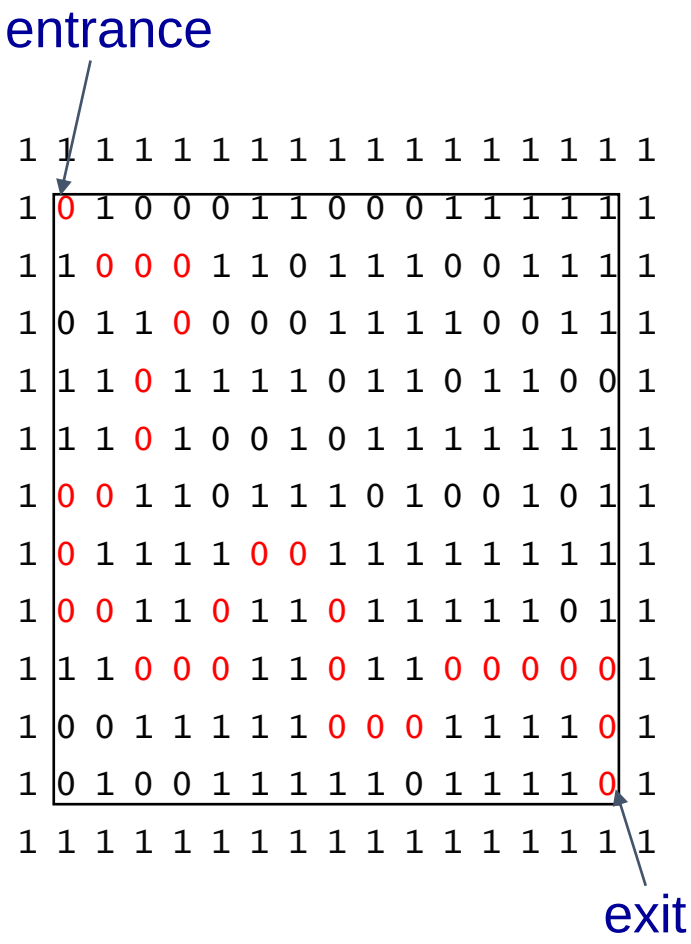


A Mazing Problem

- To avoid checking for border conditions, surround the maze by a border of ones
  - An *m* x *p* maze will require an (*m*+2) x (*p*+2) array
  - The entrance is at position [1][1] and the exit at [*m*][*p*]



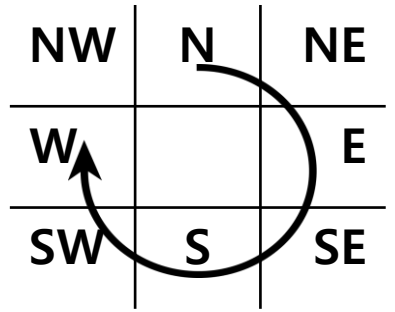
A Mazing Problem



A Mazing Problem

- We may have the chance to go in several directions
- Pick one and save our current position and the direction of the next move in the list (stack)
- If we have taken a false path, we can return and try another direction by getting the top element of the stack

A Mazing Problem



- Implementation

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

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

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

A Mazing Problem

- Since we do not want to return to a previously tried path
  - We maintain a second two-dimensional array, *mark*, to record the maze positions already checked
  - Initialize this array's entries to zero

- Representation of the stack

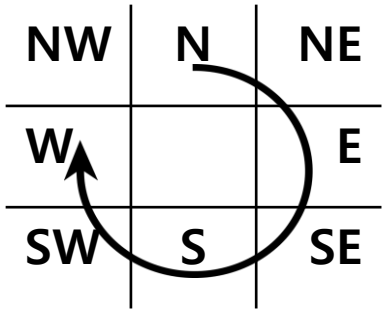
```
#define MAX_STACK_SIZE 100 /*maximum stack size*/
typedef struct {
        short int row;
        short int col;
        short int dir;
} element;
element stack[MAX_STACK_SIZE];
```

A Mazing Problem

- Initial maze algorithm

```
Initialize a stack to the maze's entrance coordinates and direction to north;
while ( stack is not empty ) {
        /* move to position at top of stack */
        <row, col, dir> = delete from top of stack;
        while ( there are more moves from current position )  {
                <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\n");
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]		1	1	1	1	1	1
[1]		0	0	1	1	0	1
[2]				0	0	1	1
[3]				0	1	1	
[4]					0	1	
[5]						0	
[6]							



ENTRY = (1,1), EXIT = (5,5)

```
                PUSH (1,1,N)
                POP
  (1,1)         PUSH (1,1,SE)      mark[1][2]=1
  (1,2)         PUSH (1,2,S)       mark[2][3]=1
  (2,3)         PUSH (2,3,SE)      mark[2][4]=1
  (2,4)         PUSH (2,4,E)       mark[1][5]=1
  (1,5)         POP
  (2,4)         PUSH (2,4,W)       mark[3][3]=1
  (3,3)         PUSH (3,3,S)       mark[4][4]=1
  (4,4)           FOUND!!!
```

(1,1)(1,2)(2,3)(2,4)(3,3)(4,4)(5,5)

Maze (Original)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]	1	1	1	1	1	1	1
[1]	1	0	0	1	1	0	1
[2]	1	1	1	0	0	1	1
[3]	1	1	1	0	1	1	1
[4]	1	1	0	1	0	1	1
[5]	1	0	1	1	0	0	1
[6]	1	1	1	1	1	1	1

stack

[5]	
[4]	(3,3,S)
[3]	(2,4,W)
[2]	(2,3,SE)
[1]	(1,2,S)
[0]	(1,1,SE)

Maze Search Function

```
elements stack[MAX_STACK_SIZE];
offset move[8];
int maze[MAX_ROWS][MAX_COLS], mark[MAX_ROWS][MAX_COLS];
int top;

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;
```

Maze Search Function

```
  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;
   } /* while (dir < 8 & !found)
} /* while (top>-1 && !found) */
```

Maze Search Function

```
  if (found) {
      printf("The path is:\n");
      printf("row  col\n");
      for (i=0; i<=top; i++)
         printf("%2d%5d\n", 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");
}
```

Expressions

```
((rear+1==front) || ((rear==MAX_QUEUE_SIZE-1) && !front))
x = a/b - c+d*e - a*c
```

- If we examine expression, we notice that it contains:
  - operators: ==, +, -, ||, &&, !
  - operands: *rear, front, MAX_QUEUE_SIZE*
  - parentheses: ( )

Evaluation of Expressions

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

Let a=4, b=c=2, d=e=3. What is the x value and why?

- Precedence rule: *,/ > +, -
- Associativity rule: left to right
- Order can be changed by using parentheses.

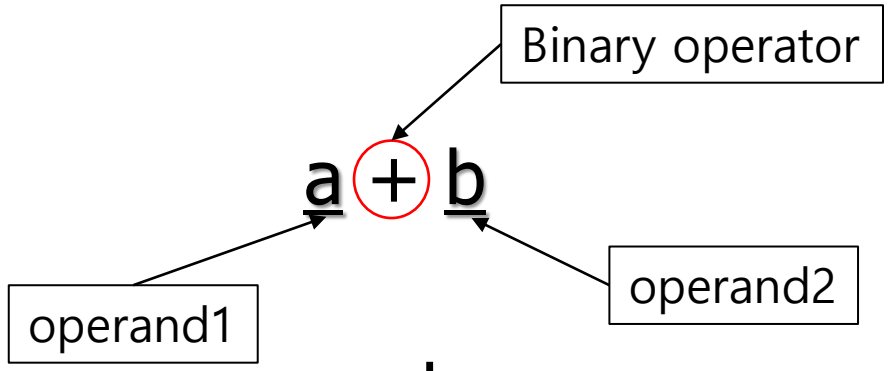
Evaluation of Expressions

- *Use fully parenthesized expression to get rid of the ambiguity.*
- *Use one pair of parentheses for each operator.*

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

```
x = ((((a/b) - c)+(d*e)) - (a*c))
```

Evaluation of Expressions



+ab

ab+

Evaluation of Expressions

- Infix notation
  - Binary operator is in-between its two operands
  - e.g. a+b, a-c*d, (a+b)*c
- Prefix notation (Polish notation)
  - Operator appears before its operands – normally used in the arithmetic expressions of Lisp
  - e.g. +ab, -a*cd, *+abc
- Postfix notation
  - Each operator appears after its operands - used by compiler
  - e.g. ab+, acd*-, ab+c*

Evaluation of Expressions

Infix notation	Prefix notation	Postfix notation
A+B*C	+A*BC	ABC*+
(A+B)*C	*+ABC	AB+C*

In case of prefix and postfix notion, we don't need to use **parentheses** anymore.

Infix to Postfix

1. Fully parenthesize the expression
2. Move all binary operators so that they replace their corresponding right parentheses
3. Delete all parentheses

e.g. **A/B-C+D*E-A*C**

→ **(((A/B)-C)+(D*E))-(A*C))**

→ **(((A/B)-C)+(D*E)) (A*C)-**

→ **((A/B)-C) (D*E)+AC*-**

→ **(A/B)C-DE*+AC*-**

→ **AB/C-DE*+AC*-**

Infix to Postfix

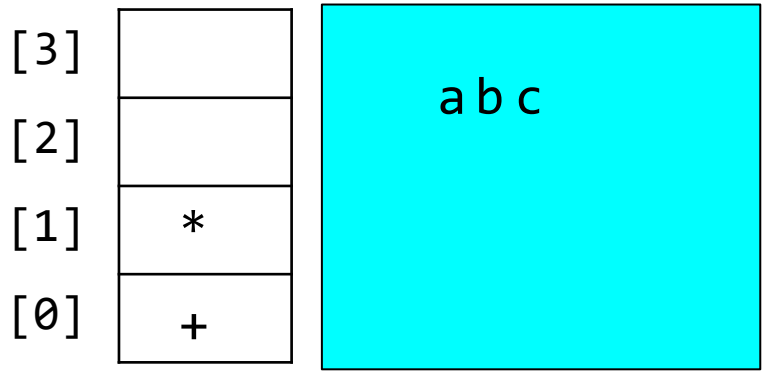
- Scan the infix expression from left to right
- Operands are passed to the output expression as they are encountered (Order of operands is same in infix and postfix expression)
- Save the operators until we know their correct placement and output the higher precedence operators first

→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

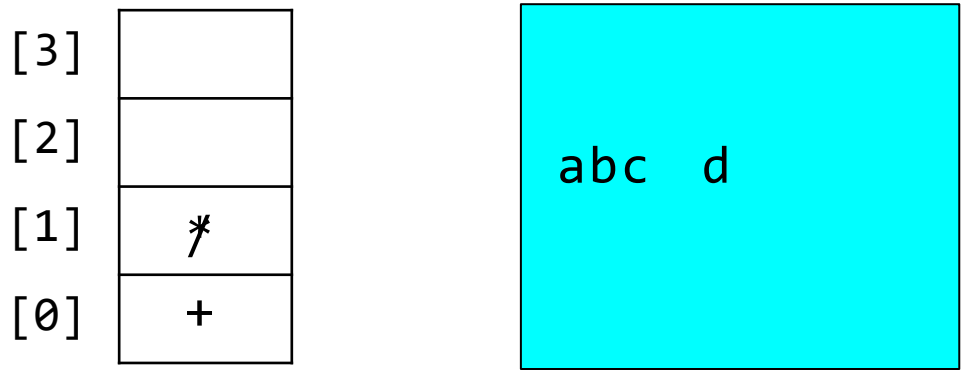
- Unstack when we reach the end of the expression

Example

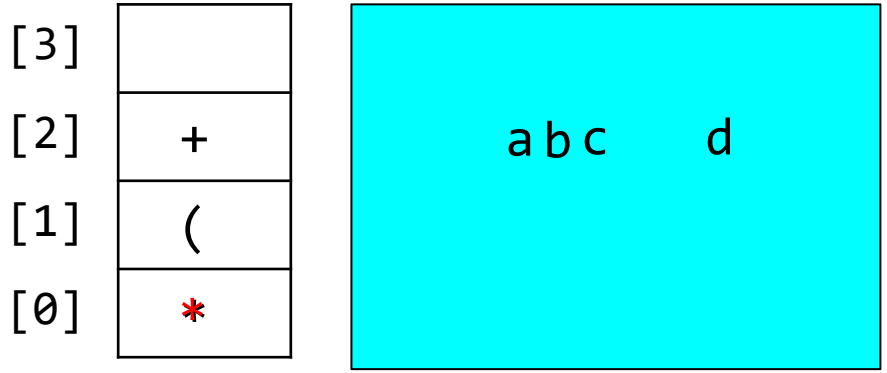
a+b*c



a+b*c/d



a*(b+c)*d



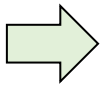
Infix to Postfix

(incoming) left parenthesis is placed in the stack whenever it is found in the expression → high precedence operator

Stack operators until we reach the right parenthesis

(in stack) left parenthesis is unstacked only when its matching right parenthesis is found → low precedence operator

Right parenthesis is never placed in the stack



Two types of precedence :
in-stack precedence(isp) and *incoming precedence(icp)*

Example

a*(b+c-d)%e → abc+d-*e%

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

Infix to Postfix Implementation

```
#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 */

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

Left parenthesis has the highest priority when incoming, and the lowest priority while in the stack

Infix to Postfix Implementation

```
void postfix(void)
{
  char symbol; precedence token; int n = 0;
  stack[0] = eos;
  for (token=getToken(&symbol,&n); token!=eos; token=getToken(&symbol,&n)){
    if (token == operand) printf("%c", symbol);
    else if (token == rparen){
       while (stack[top] != lparen) printToken(pop());
       pop(); /* discard the left parenthesis */
    }
    else {
       while (isp[stack[top]] >= icp[token]) printToken(pop());
       push(token);
    }
  }
  while ((token=pop()) != eos) printToken(token);
}
```

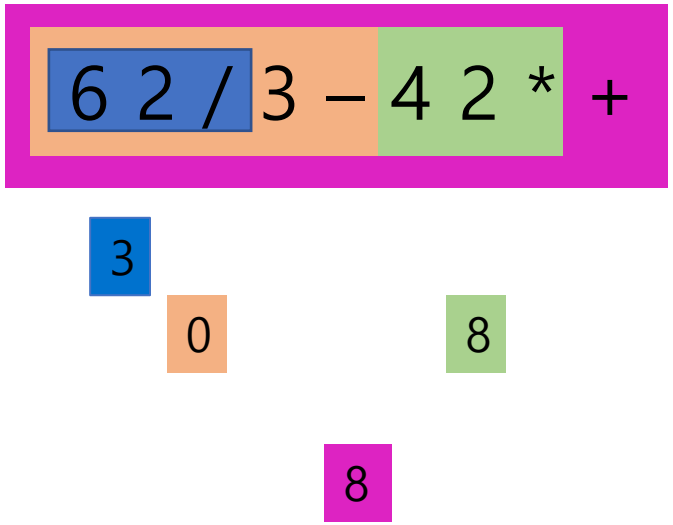
Infix to Postfix Implementation

```
precedence getToken (char *symbol, int * n)
{
   *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;
   }
}
```

Evaluating Postfix Expression

For convenience, let's assume that we use only binary operators and single-digit integers for an expression.

- Strategies
  1) Place the operands on a stack until we find an operator
  2) Remove the two operands
  3) Perform the operation and place the result on a stack



Evaluating Postfix Expression

- Scan the Postfix string from left to right
- Initialize an empty stack
- Place the operands on a stack until we find an operator
- If we find an operator,
    1. Remove, from the stack, the correct number of operands for the operator
    2. Perform the operation
    3. Place the result back on the stack

- Continue this fashion until we reach the end of the expression
- Then, remove the answer from the top of the stack

Example

Assumptions:
1) Binary operators only
2) Single digit integer

62/3-42*+

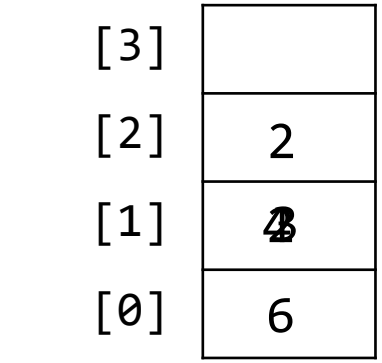
Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	3			0
3	3	3		1
-	0			0
4	0	4		1
2	0	4	2	2
*	0	8		1
+	8			0

Evaluating Postfix Expression

```
int eval(void){
   precedence token;  char symbol;  int op1,op2;
   int n = 0; top = -1;
   token = getToken(&symbol, &n);
   while (token != eos) {
      if (token==operand) push(symbol-’0’);
      else {
         op2=pop(); 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);
         } /* switch */
      } /* else */
      token = getToken(&symbol, &n);
   } /* while */
   return pop(); /* return result */
}
```

62/3-42*+

stack




```
int eval(void){
 precedence token;  char symbol;
 int op1,op2;
 int n = 0; top = -1;
 token = getToken(&symbol, &n);
 while (token != eos) {
   if (token==operand) push(symbol-'0');
   else {
     op2=pop(); 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);
     } /* switch */
   } /* else */
   token = getToken(&symbol, &n);
 } /* while */
return pop(); /* return result */
}
```

```
int stack [MAX_STACK_SIZE];
/* global variable */
```

Next Topic

Linked List