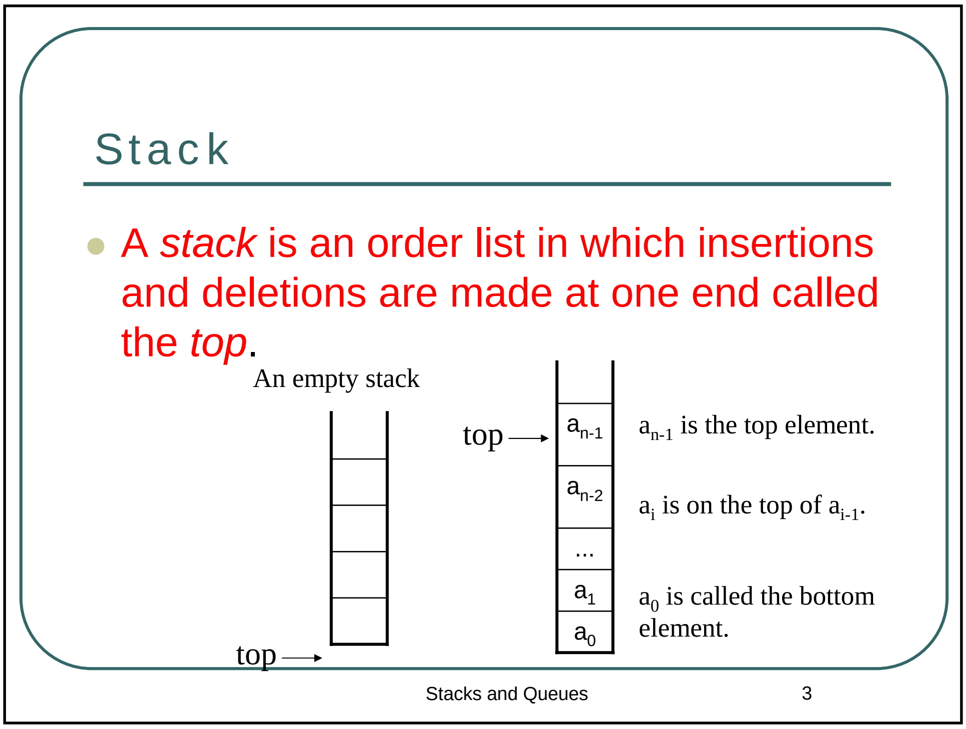
Stacks and Queues

The stack ADT

A stack is also known as a Last-In-First-Out (LIFO) list.



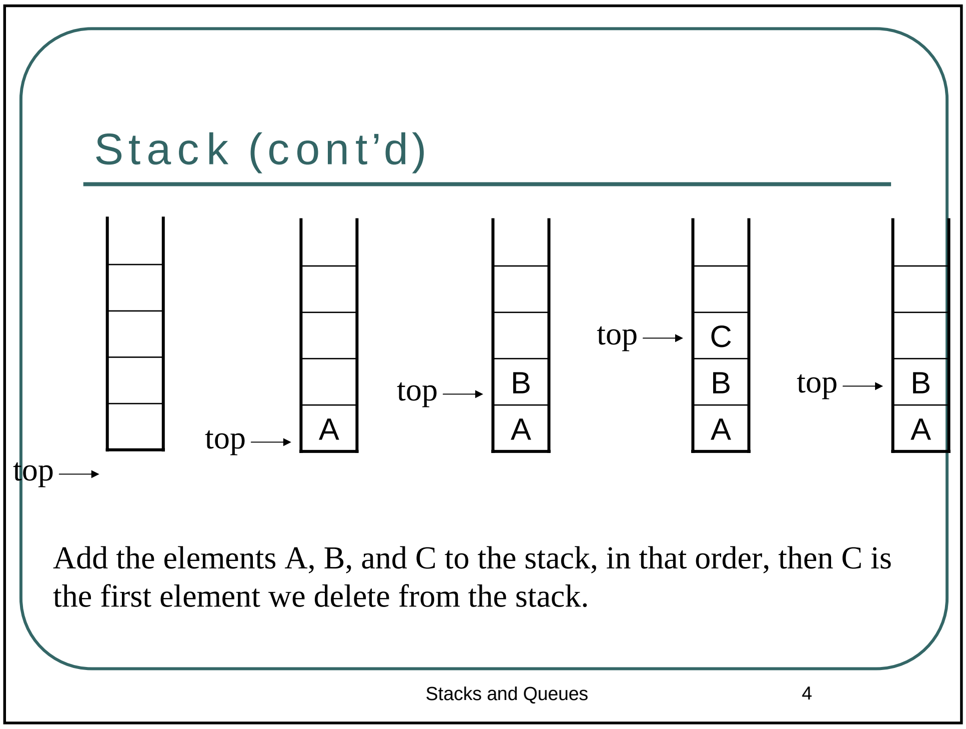
Stack

- A *stack* is an order list in which insertions and deletions are made at one end called the *top*.

a_{n-1} is the top element.

a_i is on the top of a_{i-1}.

a_0 is called the bottom element.



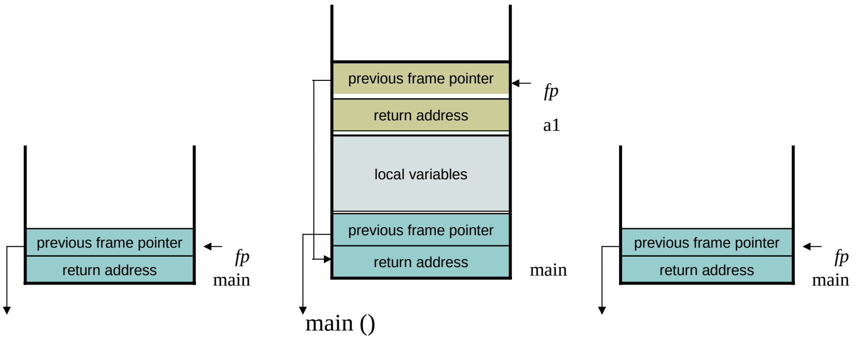
Stack (cont'd)

Add the elements A, B, and C to the stack, in that order, then C is the first element we delete from the stack.

System stack

- The *system stack* is used by a program at run time to process function calls.
  - Whenever a function is invoked, the program creates a structure, an activation record or a stack frame, and places it on top of the system stack.
    - The activation record includes
      - *a pointer* to the stack frame of the invoking function;
      - *a return address* which contains the location of the statement to be executed after the invoked function terminates.

System stack (cont'd)



```
main ()
{
        int localvariables;
        a1();
}
```

ADT Stack

```
structure Stack {
// objects: A finite ordered list with zero or more elements.
functions:
      Stack CreateS(max_stack_size);
      Boolean IsFull(stack,max_stack_size);

      Stack Push(stack,item);
      //if IsFull(), then StackFull(); else insert item into the top of the stack.

      Boolean IsEmpty(stack);

      Element Pop(stack);
      // if IsEmpty(), then StackEmpty() and return 0;
       else remove and return a pointer to the top element of the stack.
};
```

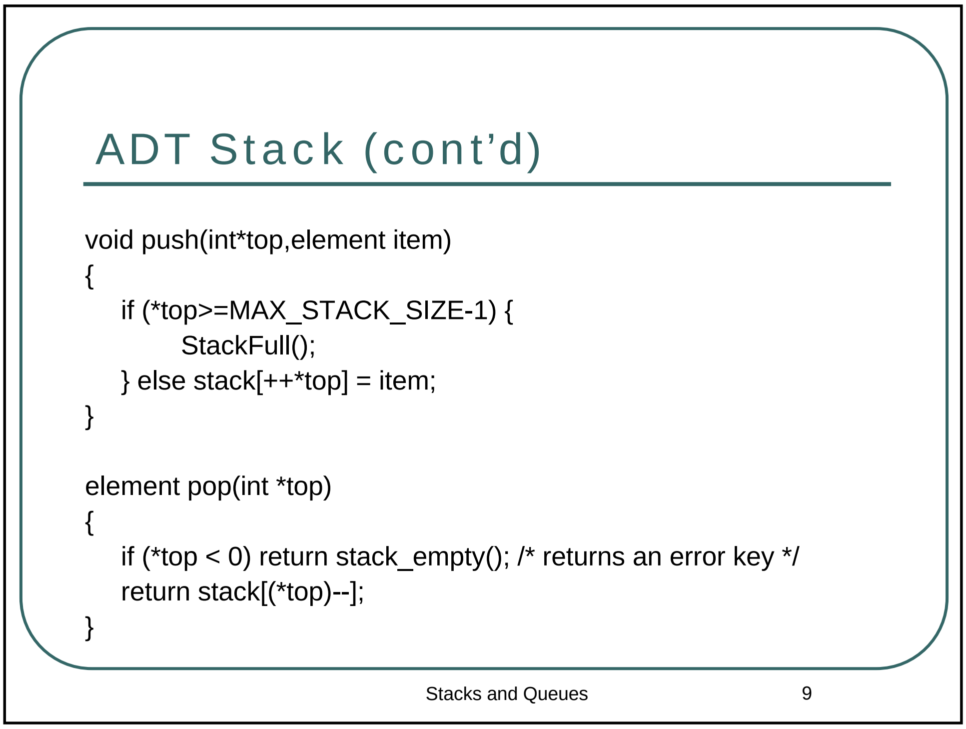
ADT Stack (cont'd)

- The easiest way to implement this ADT is using a one-dimensional array.

```
typedef struct {
    int key;
    /* other fields */
} element;

element stack[MAX_STACK_SIZE];
int top = -1;
```

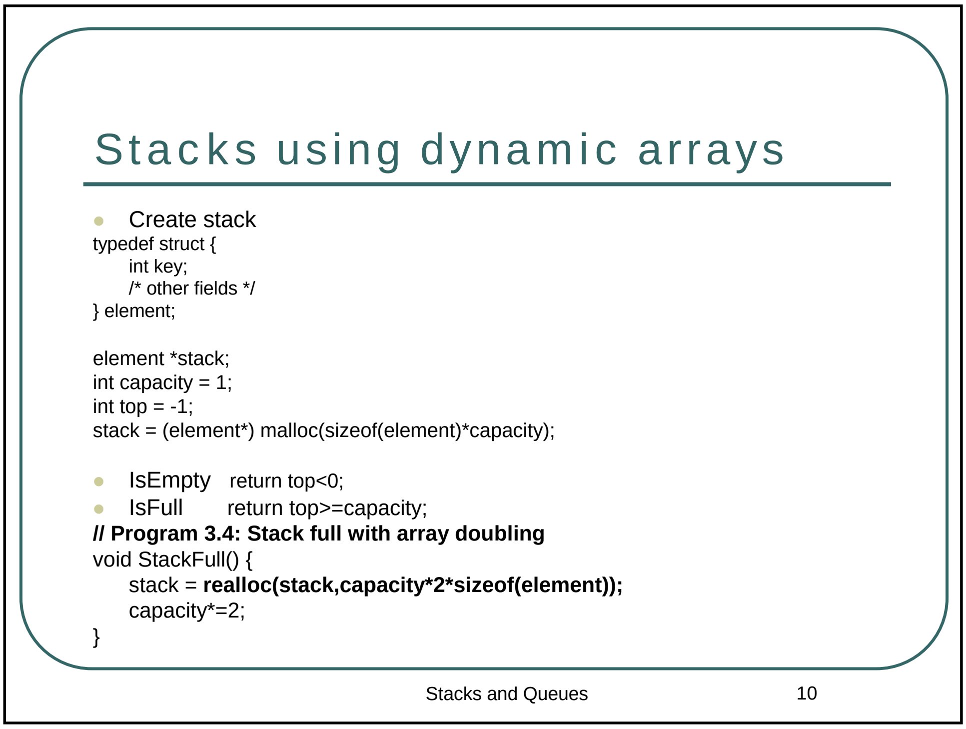
```
Boolean IsEmpty(stack) ::= top < 0
Boolean IsFull(stack) ::= top >= MAX_STACK_SIZE-1
```



ADT Stack (cont'd)

```
void push(int*top,element item)
{
    if (*top>=MAX_STACK_SIZE-1) {
          StackFull();
    } else stack[++*top] = item;
}

element pop(int *top)
{
    if (*top < 0) return stack_empty(); /* returns an error key */
    return stack[(*top)--];
}
```



Stacks using dynamic arrays

- Create stack

```
typedef struct {
     int key;
     /* other fields */
} element;

element *stack;
int capacity = 1;
int top = -1;
stack = (element*) malloc(sizeof(element)*capacity);
```

- IsEmpty return top<0;
- IsFull return top>=capacity;

// Program 3.4: Stack full with array doubling

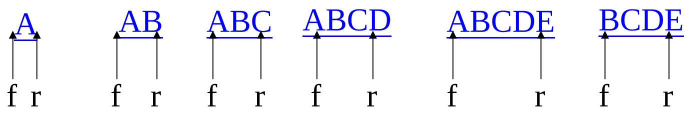
```
void StackFull() {
     stack = realloc(stack,capacity*2*sizeof(element));
     capacity*=2;
}
```

The queue ADT

A queue is also known as a First-In-First-Out (FIFO) list.

Queue

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



add add add add delete

f: queue front
r: queue rear

ADT Queue

```
structure Queue {
//objects: A finite ordered list with zero or more elements.
functions:
    Queue CreateQ(max_queue_size);
    Boolean  IsFullQ(queue, max_queue_size);
    Queue AddQ(queue,item);
    // if IsFull(), then QueueFull(); else insert item at rear of the queue

    Boolean IsEmptyQ(queue);
    Element DeleteQ(queue);
    // if IsEmpty, then QueueEmpty() and return 0;
    // else remove the item at the front of the queue and return it.
};
```

ADT Queue (cont’d)

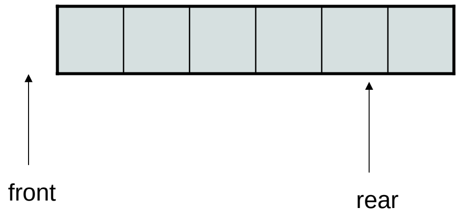
- The simplest way to implement this ADT is using a one-dimensional array and two variables, *front* and *rear*.
  - *front* is one less than the position of the first element in the queue
  - *rear* is the position of the last element in the queue

```
#define MAX_QUEUE_SIZE 100
typedef struct {
        int key;
        /* other fields */
} element;
element queue[MAX_QUEUE_SIZE];
int front = -1;
int rear = -1;
}
```

ADT Queue (cont'd)

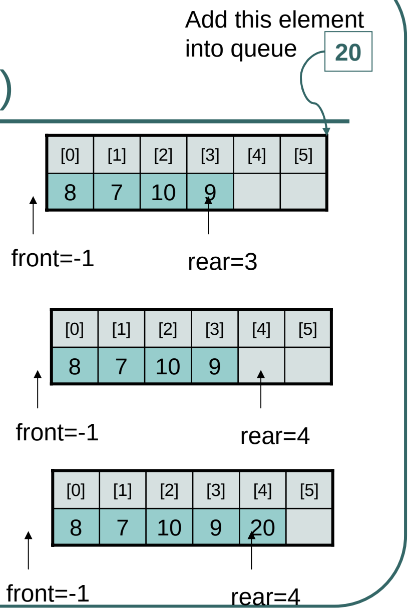
Boolean IsFull(queue) ::= rear == MAX_QUEUE_SIZE-1

Boolean IsEmpty(queue) ::= front== rear



ADT Queue (cont'd)

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



Add this element into queue **20**

[0]	[1]	[2]	[3]	[4]	[5]
8	7	10	9		

front=-1 rear=3

[0]	[1]	[2]	[3]	[4]	[5]
8	7	10	9		

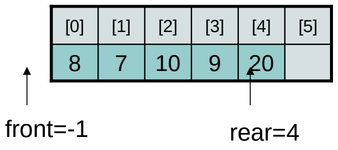
front=-1 rear=4

[0]	[1]	[2]	[3]	[4]	[5]
8	7	10	9	20	

front=-1 rear=4

ADT Queue (cont'd)

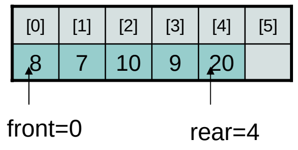
```
element deleteq(int*front, int rear)
{
    if(*front == rear) {
        return queue_empty();
    }
    return queue[++*front];
}
```



[0]	[1]	[2]	[3]	[4]	[5]
8	7	10	9	20	

front=-1

rear=4



[0]	[1]	[2]	[3]	[4]	[5]
8	7	10	9	20	

front=0

rear=4

Job scheduling

- The operating system often stores jobs for processing in a queue.

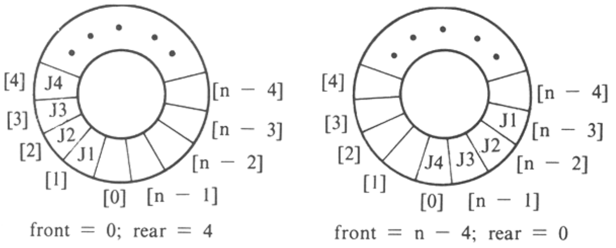
front	rear	Q[0]	[1]	[2]	[3]	[4]	[5]	[6]	...	Comments
−1	−1		queue		empty					initial
−1	0	J1								Job 1 joins Q
−1	1	J1	J2							Job 2 joins Q
−1	2	J1	J2	J3						Job 3 joins Q
0	2		J2	J3						Job 1 leaves Q
0	3		J2	J3	J4					Job 4 joins Q
1	3			J3	J4					Job 2 leaves Q

Job scheduling (cont'd)

front	*rear*	$q[0]$	[1]	[2]	···	$[n-1]$	Next Operation
-1	$n-1$	J1	J2	J3	···	Jn	initial state
0	$n-1$		J2	J3	···	Jn	delete J1
-1	$n-1$	J2	J3	J4	···	$Jn+1$	add $Jn+1$ (jobs J2 through Jn are moved)
0	$n-1$		J3	J4	···	$Jn+1$	delete J2
-1	$n-1$	J3	J4	J5	···	$Jn+2$	add $Jn+2$

call queue_full() to move the entire queue to the left O(MAX_QUEUE_SIZE)

More efficient representation: Circular queue



A more efficient queue representation is obtained by regarding the array queue[MAX_QUEUE_SIZE] as circular.

- Initially, we have front=rear=0;

More efficient representation: Circular queue (cont'd)

```
void addq(element item)
{
    int newrear=(rear+1)%MAX_QUEUE_SIZE;
    if (front==newrear) queue_full();
    else queue[rear=newrear]=item;
}

element deleteq()
{
    if (front==rear) { return queue_empty();}
    front = (front+1)% MAX_QUEUE_SIZE;
    return queue[front];
}
```

Discussion

- addq and deleteq are O(1).
- *front* will always point one position counterclockwise from the first element in the queue.

```
void addq(element item)
{
    int newrear=(rear+1)%MAX_QUEUE_SIZE;
    if (front==newrear) queue_full();
    else queue[rear=newrear]=item;
}
```

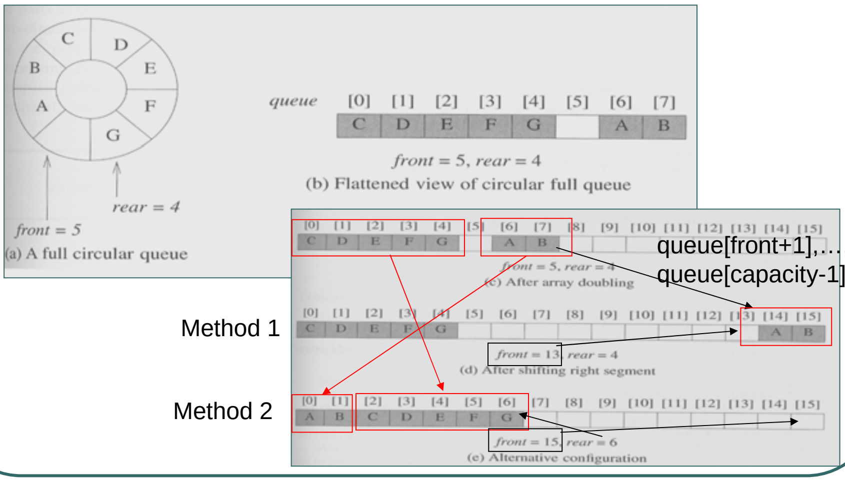
```
element deleteq()
{
    if (front==rear) { return queue_empty();}
    front = (front+1)% MAX_QUEUE_SIZE;
    return queue[front];
}
```

- In this implementation, *rear* is never equal to *front* unless the queue is empty.
    - It permits the maximum of MAX_QUEUE_SIZE-1 rather than MAX_QUEUE_SIZE elements to be in the queue.

Discussion (cont'd)

- One way to use all MAX_QUEUE_SIZE elements would be use an additional variable, *LastOp*, to record the last operation performed on the queue.
    - the variable is initialized to DELETE.
    - if *rear*==*front* && *LastOp*==ADD, the queue is full.
    - if *rear*==*front* && *LastOp*==DELETE, the queue is empty.

Circular queue using dynamic allocated arrays



(a) A full circular queue

(b) Flattened view of circular full queue

(c) After array doubling

(d) After shifting right segment

(e) Alternative configuration

queue[front+1],…, queue[capacity-1]

Method 1

Method 2

Evaluation of expression

Expression

- An expression is made up of operands, operators, and delimiters.
    - A/B-C+D*E-A*C
- arithmetic operators

  +,-,*,/, unary minus, and %
- relational operators

  <,<=,==,<.,>=, >, &&, ||, and !.

Priority of operators

- Which is the meaning of the expression A/B-C+D*E-A*C ?
  - ((A/B)-C)+(D*E)-(A*C) or
    (A/(B-C+D))*(E-A)*C
  - To fix the order of evaluation, each operator is assigned a **priority**.
  - Then, within any pair of parentheses , the operators with the highest priority will be evaluated first.

Priority of operators (cont'd)

Priority of operators in C

priority	operator
1	unary minus,!
2	*,/,%
3	+,-
4	<,<=,>=,>
5	==,!=
6	&&
7	||

The C rule is that for all priorities, evaluation of operators of the same priority will proceed left to right.

- A/B*C will be evaluated as (A/B)*C.
- X=A/B-C+D*E-A*C will be evaluated as X=(((A/B)-C)+(D*E))-(A*C).

Postfix notation

- A compiler accepts an expression and produces correct code by reworking the expression into a form called *postfix notation*.
- The conventional way of writing an expression is called *infix*.
  - the operators come in-between the operands
- Infix A*B/C has postfix AB*C/.
  - Infix: A/B-C+D*E-A*C
  - Postfix: AB/C-DE*+AC*-

Postfix evaluation

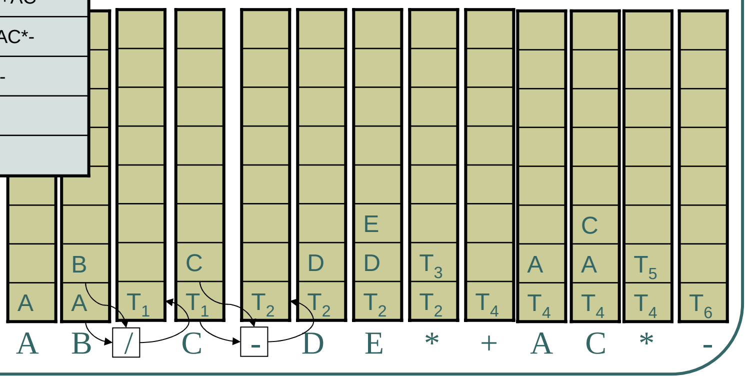
A/B-C+D*E-A*C   AB/C-DE*+AC*-

operation	postfix
T_1=A/B	T_1C-DE*+AC*-
T_2=T1-C	T_2DE*+AC*-
T_3=D*E	T_2T_3+AC*-
$T_4=T_2+T_3$	T_4AC*-
T_5=A*C	T_4T_5-
$T_6=T_4-T_5$	T_6

Postfix evaluation (cont'd)

operation	postfix
T_1=A/B	T_1C-DE*+AC*-
T_2=T1-C	T_2DE*+AC*-
T_3=D*E	T_2T_3+AC*-
T_4=T_2+T_3	T_4AC*-
T_5=A*C	T_4T_5-
T_6=T_4-T_5	T_6

AB/C-DE*+AC*-



Postfix evaluation (cont'd)

```
void eval(expression e)
{
    for(token x =get_token(e); x!= eos; x=getToken(e))
         if (x is an operand) push(x);
         else { // operator
                 remove the correct number of operands for operator x from stack;
                 perform the operation x and store the result  (if any) onto the stack;
         }
}
```

Infix to Postfix

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

- Example, convert A/B-C+D*E-A*C into its postfix representation
  - ((((A/B)-C)+(D*E))-(A*C))
  - ((((AB/) C-) (DE*) +) (AC*) -)
  - AB/C-DE*+AC*-

Infix to Postfix (cont'd)

- The order of the operands is the same in postfix and infix.
- Infix: A+B*C
  - A->B->C
- Postfix: ABC*+
  - A->B->C

next token	stack	output
none	empty	none
A	empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
		ABC*+

Infix to Postfix (cont'd)

next token	stack	output
none	empty	none
A	empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	ABC+
*	*	ABC+*
D	*	ABC+*D
done	empty	ABC+*D*

Write the postfix form of A*(B+C)*D.

Infix to Postfix (cont'd)

isp in-stack precedence	icp in-coming precedence	operator
0	20	(
19	19	)
12	12	+
12	12	-
13	13	*
13	13	/
13	13	%
0	0	Eos

Operators are taken out of the stack as long as their **in-stack precedence (isp)** is numerically **greater than or equal** to the **in-coming precedence (icp)** of the new operator.

Infix to Postfix (cont'd)

Token	Stack			Top	isp vs. icp	Output
	[0]	[1]	[2]			
A				-1		A
*	*			0		A
(	*	(		1	isp('*')<icp('(')	A
B	*	(		1		AB
+	*	(	+	2	isp('(')<icp('+')	AB
C	*	(	+	2		ABC
)	*			0	Unstack until '('	ABC+
*	*			0	isp('*')>=icp('*')	ABC+*
D	*			0		ABC+*D
eos				-1	isp('*')>=icp(eos)	ABC+*D*

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

Translation of A*(B+C)*D to postfix

Infix to Postfix (cont'd)

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

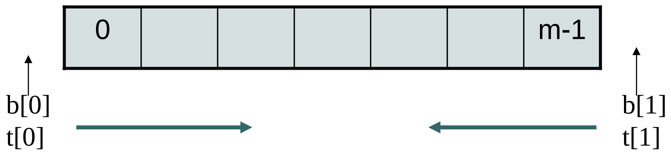
Infix to Postfix (cont'd)

- The complexity of function postfix is $\Theta(n)$.

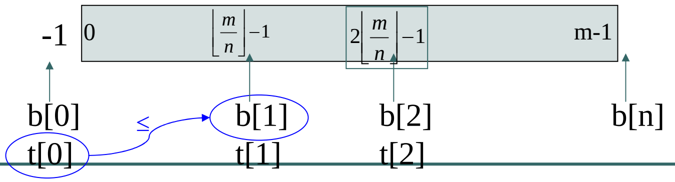
Multiple stacks and queues

Multiple stacks

- Represent two stacks in an array



- Represent multiple stacks in an array



Multiple stacks (cont’d)

```
void push(int i, element item)
{
    if (t[i]==b[i+1]) stack_full(i);
    else M[++t[i]] = item;
}
element pop(int i)
{
    if (t[i]==b[i]) return stack_empty(i);
    return M[t[i]--];
}
```

stack_full(i)

Step 1.

- Determine the least, j, i<j<n, such that t[j]<b[j+1].
- Move stacks i+1, i+2, ..., j, one position to the right, thereby creating a space between stacks i and i+1.

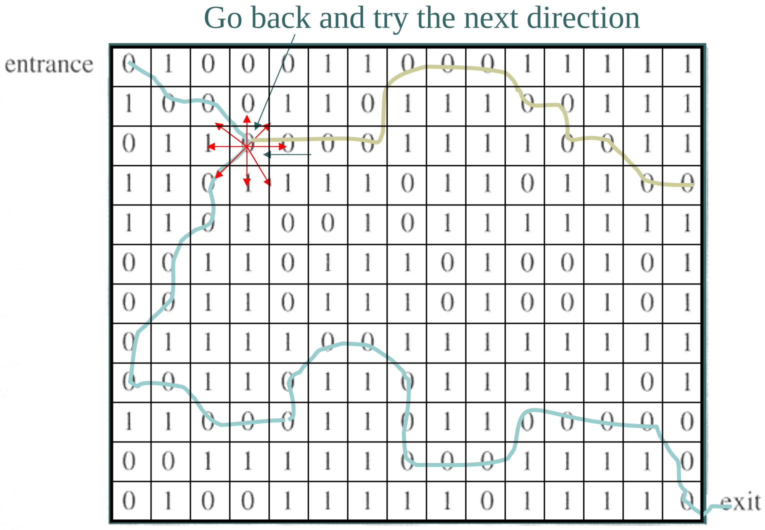
Step 2.

- If there is no j as in (1), then look to the left of stack i. Find the largest j such that 0<=j<i t[j]<b[j+1].
- Move stacks j+1, j+2, ..., i one space left.

Step 3.

- If there is no j satisfying either the conditions of (1) or (2), then all m spaces of M are utilized, and there is no free space.

A mazing problem



Go back and try the next direction

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

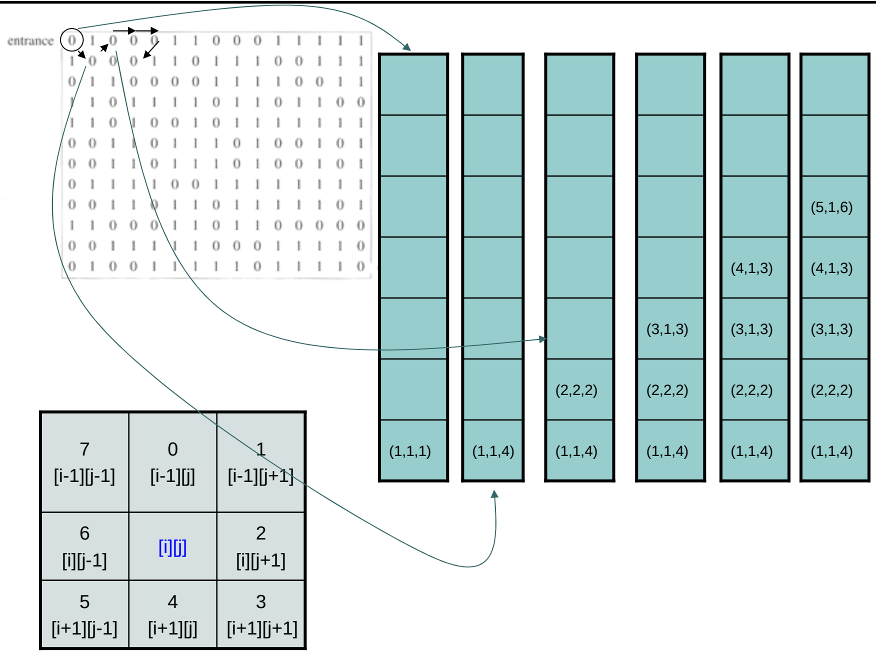
Discussion

- Use a stack to keep track of the moves.
  - this stack is also used to return to the last position and try the next direction of the last move.
- Use an array *mark* to prevent from going down the same path twice.

```
void path(void)
{    int i, row, col, next_row,next_col, dir,found = FALSE;
     element position;
     mark[1][1] =1;
     top = 0;
     stack[0].row = 1; stack[0].col=1; stack[0].dir = 1;
     while(top > -1 && ! found) {
               position = pop(&top);        /* backtracking */
               row = position.row; col = positioin.col; dir = position.dir;
               while(dir < 8 && ! found) {
                            next_row=row + move[dir].vert; next_col = col+move[dir].horiz;
                            if (next_row==EXIT_ROW&&next_col==EXIT_COL) {
                                         found = TRUE;
                            } else if((!maze[next_row][next_col]) && (!mark[next_row][next_col])) {
                                         mark[next_row][next_col] = 1; /* mark this position */
                                         position.row =row; /* save current position */
                                         position.col = col;
                                         position.dir = dir+1;
                                         push(&top,position);
                                         row = next_row; col = next_col; dir = 0;
                            } else dir++;
               }
     }
     if(found) { /* output path */ }
}
```

explore

7 [i-1][j-1]	0 [i-1][j]	1 [i-1][j+1]
6 [i][j-1]	[i][j]	2 [i][j+1]
5 [i+1][j-1]	4 [i+1][j]	3 [i+1][j+1]



```
0 1 0 0 0 1 1 0 0 0 1 1 1 1 1
1 0 0 0 1 1 0 1 1 1 0 0 1 1 1
0 1 1 0 0 0 0 1 1 1 1 0 0 1 1
1 1 0 1 1 1 1 0 1 1 0 1 1 0 0
1 1 0 1 0 0 1 0 1 1 1 1 1 1 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 0 1 1 0 1 1 1 0 1 0 0 1 0 1
0 1 1 1 1 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 1 1 0 1 1 1 1 1 0 1
1 1 0 0 0 1 1 0 1 1 0 0 0 0 0
0 0 1 1 1 1 1 0 0 0 1 1 1 1 0
0 1 0 0 1 1 1 1 1 0 1 1 1 1 0
```

					(5,1,6)
				(4,1,3)	(4,1,3)
			(3,1,3)	(3,1,3)	(3,1,3)
		(2,2,2)	(2,2,2)	(2,2,2)	(2,2,2)
(1,1,1)	(1,1,4)	(1,1,4)	(1,1,4)	(1,1,4)	(1,1,4)

7 [i-1][j-1]	0 [i-1][j]	1 [i-1][j+1]
6 [i][j-1]	[i][j]	2 [i][j+1]
5 [i+1][j-1]	4 [i+1][j]	3 [i+1][j+1]