

DATA STRUCTURE AND ALGORITHM

CLASS 5

Seongjin Lee

Updated: 2017-03-06
DSA_2017_05

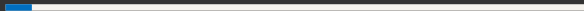
insight@gnu.ac.kr
<http://resourceful.github.io>
Systems Research Lab.
GNU



Table of contents

1. Stack
2. Queues
3. Circular Queues
4. A Mazing Problem
5. Evaluation of Expressions
6. Multiple Stacks Queues

STACK



Stack Abstract Data Type

Stack and Queue is

- special cases of the more general data type, **Ordered List**

ADT Stack

- Ordered List
- Insertions and deletions are made at one end called the top

Stack Abstract Data Type

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

- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

stack state

Stack Abstract Data Type

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

- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A

stack state

A ← top

Stack Abstract Data Type

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

- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A \rightarrow push B

stack state

A \leftarrow top B \leftarrow top
A

Stack Abstract Data Type

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

- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A → push B → push C

stack state

		C ← top
	B ← top	B
A ← top	A	A

Stack Abstract Data Type

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

- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A → push B → push C → push D

stack state

				D	← top
			C	C	
	B	← top	B	B	
A	A		A	A	

Stack Abstract Data Type

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

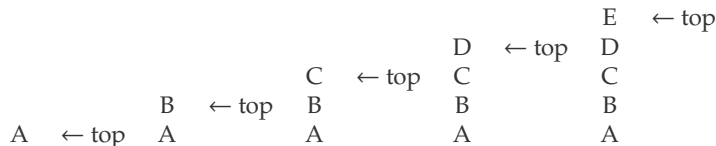
- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A → push B → push C → push D → push E

stack state



Stack Abstract Data Type

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

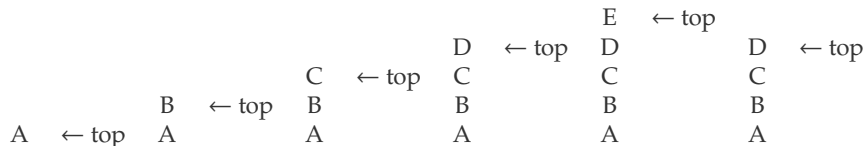
- a_0 : Bottom element
- a_{n-1} : Top element
- a_i : On top of element a_{i-1} ($0 < i < n$)

A.K.A Last-In-First-out (LIFO)

Inserting and deleting elements in a stack

push A → push B → push C → push D → push E → pop E

stack state



Stack Abstract Data Type: System Stack

Stack is used by a program at run-time to process function calls

Activation record (stack frame) initially contains only

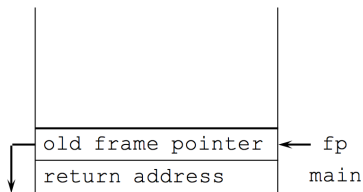
- a pointer to the previous stack frame
- a return address

If this invokes another function

- local variables
- parameters of the invoking function

Stack Abstract Data Type: System Stack

System Stack after function call

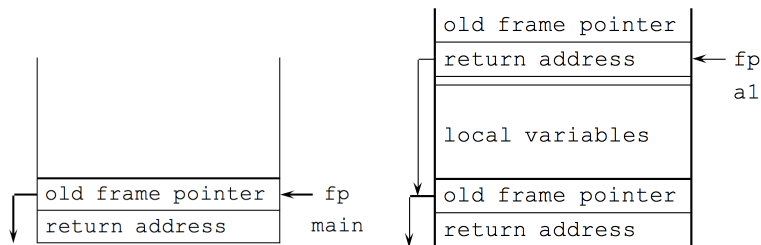


Stack Abstract Data Type: System Stack

System Stack after function call

Run-time program simply creates a new stack frame

- also for each recursive call



Stack Abstract Data Type

Structure: Stack is

Objects: a finite ordered list with zero or more elements

Functions:

For all $stack \in \text{Stack}$,

$item \in \text{element}$,

$max_stack_size \in \text{positive integer}$:

Stack CreateS(max_stack_size);

Boolean IsFull(stack, max_stack_size);

Stack Push(stack, item);

Boolean IsEmpty(stack);

Element Pop(stack);

Stack Abstract Data Type: Implementation

- Using a one-dimensional array
 - `stack[MAX_STACK_SIZE]`
 - where `MAX_STACK_SIZE`: maximum number of entries

```
1 #define MAX_STACK_SIZE 100
2 typedef struct {
3     int key; // only key field
4             // can add/modify fields to meet the requirements
5 } element;
6
7 element stack[MAX_STACK_SIZE];
8 int top = -1;
```


Stack Abstract Data Type: implementation

IsEmpty(stack)

```
return(top < 0);
```

IsFull(stack);

```
return(top >= MAX_STACK_SIZE-1);
```

Stack Abstract Data Type: implementation

Push(stack, item)

```
void push(int *ptop, element item){
    if (*ptop >= MAX_STACK_SIZE -1) {
        stack_full();
        return;
    }
    stack[++*ptop] = item;
}
```

Pop(int *ptop);

```
element pop(int *ptop){
    if(*ptop == -1)
        return stack_empty();
    return stack[(*ptop)--];
}
```

Stack Abstract Data Type: Application of Stack

- Procedure calls/returns
- Syntactic analyzer
- converting recursive procedures to non-recursive procedures

QUEUES



Queue Abstract Data Type: Characteristics

- Ordered list
- All insertions are made at one end, called rear
- All deletions are made at the other end, called front
- which item is to be removed first?
 - FIFO (First In First Out)
- All items except front/rear items are hidden

Queue Abstract Data Type: Insertion and Deletion

Operation

Queue state

Queue Abstract Data Type: Insertion and Deletion

Operation insert A

Queue state

A \leftarrow rear
 \leftarrow front

Queue Abstract Data Type: Insertion and Deletion

Operation insert A \rightarrow insert B

Queue state

		B	\leftarrow rear
A	\leftarrow rear	A	\leftarrow front
	\leftarrow front		

Queue Abstract Data Type: Insertion and Deletion

Operation insert A \rightarrow insert B \rightarrow insert C

Queue state

		C	\leftarrow rear
	B	B	
A	\leftarrow rear	A	\leftarrow front
	\leftarrow front		

Queue Abstract Data Type: Insertion and Deletion

Operation insert A → insert B → insert C → insert D

Queue state

				D	← rear
			C	C	
	B	← rear	B	B	
A	← rear	A	← front	A	← front
	← front				

Queue Abstract Data Type: Insertion and Deletion

Operation insert A → insert B → insert C → insert D → delete D

Queue state

				D	← rear	
			C	← rear	C	D
			B		B	← rear
			A	← front	A	C
A	← rear	B	← rear			B
← front	A	← front	A	← front	A	← front

Queue Abstract Data Type: Implementation

Simplest scheme

- one-dimensional array, and two variables: front and rear

```
1  #define MAX_QUEUE_SIZE 100
2  typedef struct {
3      int key;
4      /* other fields */
5  } element;
6
7  element queue[MAX_QUEUE_SIZE];
8  int rear = -1;
9  int front = -1;
```

Queue Abstract Data Type: Implementation

IsEmptyQ(queue)

```
return (front == rear)
```

IsFullQ(queue)

```
return rear == (MAX_QUEUE_SIZE-1)
```

Queue Abstract Data Type

addq(*prear, element item)

```
1 void addq(int *prear, element item){
2     if(*prear == MAX_QUEUE_SIZE - 1){
3         queue_full();
4         return;
5     }
6     queue[++*prear] = item;
7 }
```

deleteq(*pfront, int rear)

```
1 element deleteq(int *pfront, int rear){
2     if(*pfront == rear){ // rear is used to check for an empty queue
3         return queue_empty();
4     }
5     return queue[++*pfront];
6 }
```

Queue Abstract Data Type: Example - Sequential Queue

Job Scheduling: Creation of job queue

- in the OS which does not use priorities, 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

Queue Abstract Data Type: Example - Sequential Queue

Problem

- Queue gradually shifts to the right
- `queue_full(rear == MAX_QUEUE_SIZE-1)` signal does not always mean that there are `MAX_QUEUE_SIZE` items in queue
- There may be empty spaces available
- data movement: $O(\text{MAX_QUEUE_SIZE})$

Solution:

Queue Abstract Data Type: Example - Sequential Queue

Problem

- Queue gradually shifts to the right
- `queue_full(rear == MAX_QUEUE_SIZE-1)` signal does not always mean that there are `MAX_QUEUE_SIZE` items in queue
- There may be empty spaces available
- data movement: $O(\text{MAX_QUEUE_SIZE})$

Solution:

- Circular Queue

CIRCULAR QUEUES



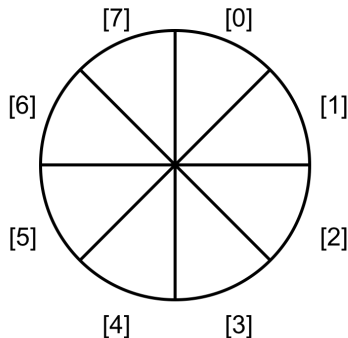
Circular Queues

More efficient Queue representation

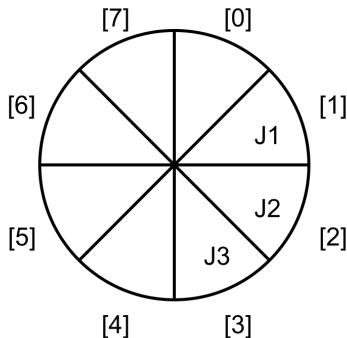
- regard the array `queue[MAX_QUEUE_SIZE]` as circular
- initially front and rear to 0 rather than -1
- the front index always points one position counterclockwise from the first element in the queue
- the rear index point to the current end of the queue

Circular Queues

empty and nonempty circular queues



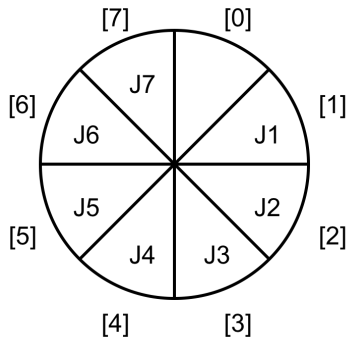
front = 0
rear = 0



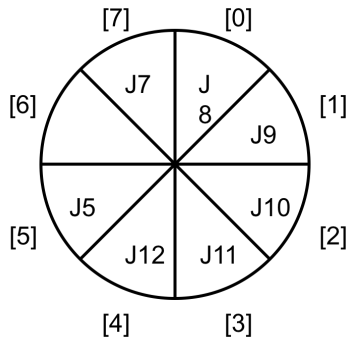
front = 0
rear = 3

Circular Queues

full circular queues



front = 0
rear = 7



front = 6
rear = 5

Circular Queues: Implementation

- Use modulus operator for circular rotation

Circular rotation of the rear

```
rear = (rear + 1) % MAX_QUEUE_SIZE;
```

Circular rotation of the front

```
front = (front + 1) % MAX_QUEUE_SIZE;
```

Circular Queues: Implementation

Add to a circular queue

- rotate rear before we place the item in the rear of the queue

```
1 void addq(int front, int *prear, element item){
2     *prear = (*prear + 1) % MAX_QUEUE_SIZE;
3     if (front == *prear) {
4         queue_full(prear);
5         /* reset rear and print error */
6         return;
7     }
8     queue[*prear] = item;
9 }
```

Circular Queues: Implementation

Delete from a circular queue

```
1 element deleteq(int *pfront, int rear){  
2     element item;  
3     if (*pfront == rear)  
4         return queue_empty();  
5     /* queue_empty returns an error key */  
6     *pfront = (*pfront + 1) % MAX_QUEUE_SIZE;  
7     return queue[*pfront];  
8 }
```


Circular Queues: Implementation notes

Tests for a full queue and an empty queue are the same

- To distinguish between the case of full and empty, permit a maximum of $\text{MAX_QUEUE_SIZE} - 1$

No data movement necessary

- Ordinary queue: $O(n)$
- Circular queue: $O(1)$

A MAZING PROBLEM



A Mazing Problem

The representation of the maze

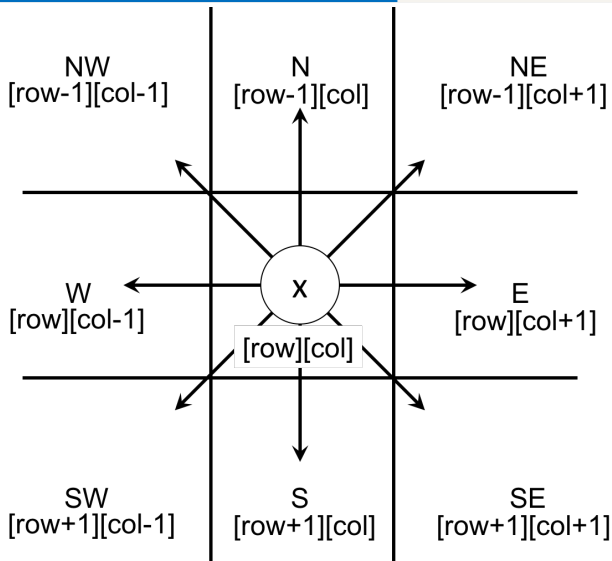
- two-dimensional array
- element 0 : open path
- element 1 : barriers

entrance →

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

exit →

A Mazing Problem: Allowable Movements



A Mazing Problem: Conditions

[row][col] which is on border

- has only three (or two) neighbors
- surround the maze by a border of 1's

$m \times p$ maze

- require $(m + 2) * (p + 2)$ array
- entrance position: [1][1]
- exit position: [m][p]

A Mazing Problem: Data Type

```
1 typedef struct {
2     short int vert;
3     short int horiz;
4 } offsets
5
6 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

A Mazing Problem: Positioning of moves

Position of next move

- move from current position: `maze[row][col]` to the next position `maze[next_row][next_col]`

```
1 next_row = row + move[dir].vert;  
2 next_col = col + move[dir].horiz;
```

A Mazing Problem: Approach

Maintain a second two-dimensional array, mark

- avoid returning to a previously tried path
- initially, all entries are 0
- mark to 1 when the position is visited

A Mazing Problem: Initial maze algorithm

```
1  initialize a stack to the maze's entrance coordinates
2      and direction to north;
3  while (stack is not empty) {
4      /* move to position at top of stack */
5      <row,col,dir> = delete from top of the stack;
6      while (there are more moves from current position) {
7          <next_row, next_col> = coordinates of next move;
8          dir = direction of move;
9          if ((next_row == EXIT_ROW) &&
10             (next_col == EXIT_COL))
11              success;
12          if (maze[next_row][next_col] == 0 &&
13             mark[next_row][next_col] == 0) {
14              mark[next_row][next_col] = 1;
15              add <row, col, dir> to the top of the stack;
16              row = next_row;
17              col = next_col;
18              dir = north;
19          }
20      }
21  }
```

```
22  printf("no path found\n");
```

A Mazing Problem: Data Type

```
1 #define MAX_STACK_SIZE 100
2 typedef struct {
3     short int row;
4     short int col;
5     short int dir;
6 } element;
7 element stack[MAX_STACK_SIZE];
```

bound for the stack size

- the stack need only as many positions as there are zeroes in the maze

EVALUATION OF EXPRESSIONS

Expressions

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

To understand the meaning of a expressions and statements

- figure out the order in which the operations are performed

Operator precedence hierarchy

associativity

- how to evaluate operators with the same precedence

Precedence hierarchy for C

token	precedence	associativity
() [] -> .	17	left-to-right
-- ++	16	left-to-right
-- ++ ! ~ - + & * sizeof (type)	15	right-to-left
* / %	14	right-to-left
+ -	13	left-to-right
<< >>	12	left-to-right
> >= < <=	11	left-to-right
== !=	10	left-to-right
&	9	left-to-right
^	8	left-to-right
	7	left-to-right
&&	6	left-to-right
	5	left-to-right
?:	4	left-to-right
= += -= /= *= %= <<= >>= &= ^= =	3	right-to-left
,	2	right-to-left
	1	left-to-right

Evaluation of Expressions

Human Style

1. assign priority to each operator
2. use parenthesis and evaluate inner-most ones
$$(((A * (b + c)) + (d / e)) - (a / (c * d)))$$

Compiler Style (in postfix form)

1. translation (infix to postfix)
2. evaluation (postfix)

prefix form: (operator) operand operand

infix form: operand (operator) operand

postfix form: operand operand (operator)

Prefix, Infix, and Postfix Notation

Prefix	Infix	Postfix
+ 2 * 3 4	2 + 3 * 4	2 3 4 * +
+ * a b 5	a * b + 5	a b * 5 +
* + 1 2 7	(1 + 2) * 7	1 2 + 7 *
/ * a b c	a * b / c	a b * c /
*/a + - b c d * - e a c	((a / (b - c + d)) * (e - a)) * c	a b c - d + / e a - * *
+ - / a b c - * d e * a c	a / b - c + d * e - a * c	a b / c - d e * + a c * -

Evaluation of postfix expression

- scan left-to-right
- place the operands on a stack until an operator is found
- perform operations

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1

Evaluating Postfix Expression

6 2 / 3 - 4 2 * +

token	[0]	[1]	[2]	top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Evaluating Postfix Expression

`get_token()`

- used to obtain tokens from the expression string

`eval()`

- if the token is operand, convert it to number and push to the stack
- otherwise
 - pop two operands from the stack
 - perform the specified operation
 - push the result back on the stack

Evaluating of Expression

```
1  #define MAX_STACK_SIZE 100 /* maximum stack size */
2  #define MAX_EXPR_SIZE 100 /* max size of expression */
3
4  typedef enum {lparen, rparen,
5               plus, minus,
6               times, divide,
7               mode, eos, operand
8               } precedence;
9
10 int stack[MAX_STACK_SIZE]; /* global stack */
11 char expr[MAX_EXPR_SIZE]; /* input string */
```

represent stack by a global array

- accessed only through top
- assume only the binary operator $+$, $-$, $*$, $/$, and $\%$
- assume single digit integer

Function to evaluate a postfix expression I

```
1  int eval(){
2      precedence token;
3      char symbol;
4
5      int op1, op2;
6
7      int n = 0;
8      int top = -1;
9
10     token = get_token(&symbol, &n);
11
12     while (token != eos) {
13         if (token == operand)
14             push(&top, symbol-'0');
15         else {
16             op2 = pop(&top);
17             op1 = pop(&top);
18
19             switch (token) {
20                 case plus: push(&top, op1+op2);
```

Function to evaluate a postfix expression II

```
21             break;
22         case minus: push(&top, op1-op2);
23             break;
24         case times: push(&top, op1*op2);
25             break;
26         case divide: push(&top, op1/op2);
27             break;
28         case mod: push(&top, op1%op2);
29     }
30 }
31 token = get_token(&symbol, &n);
32 }
33 return pop(&top);
34 }
```

Function to get a token I

```
1 precedence get_token(char *psymbol, int *pn) {
2     *psymbol = expr[(*pn)++];
3     switch (*psymbol)
4         case '(' : return lparen;
5         case ')' : return rparen;
6         case '+' : return plus;
7         case '-' : return minus;
8         case '*' : return times;
9         case '/' : return divide;
10        case '%' : return mod;
11        case ' ' : return eos;
12        default : return operand; /* no error checking */
13    }
14 }
```

Complexity

- time: $O(n)$ where n : number of symbols in expression
- space: stack `expr[MAX_EXPR_SIZE]`

Infix to Postfix

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 parentheses
3. delete all parentheses

e.g. $a / b - c + d * e - a * c$

1. $(((((a / b) - c) + (d * e)) - (a * c)))$
2. $ab/c-de^*+ac^*-$

Requires two pass

Infix to Postfix

Form a postfix in one pass

- order of operands is the same in infix and postfix
- order of operators depends on precedence
- we can use stack

Simple expression: $a + b * c$

- $a\ b\ c\ *\ +$

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a
+	+			0	a

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a
+	+			0	a
b	+			0	ab

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc

Infix to Postfix

- Output operator with higher precedence before those with lower precedence

Translation of $a + b * c$ to postfix

token	[0]	[1]	[2]	top	output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

Infix to Postfix: Parenthesized expression

parentheses make the translation process more difficult

- equivalent postfix expression is parenthesis-free

expression $a * (b + c) * d$

- yield $a\ b\ c\ +\ *d\ *$ in postfix

right parenthesis

- pop operators from a stack until left parenthesis is reached

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+
*	*			0	abc+*

Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d

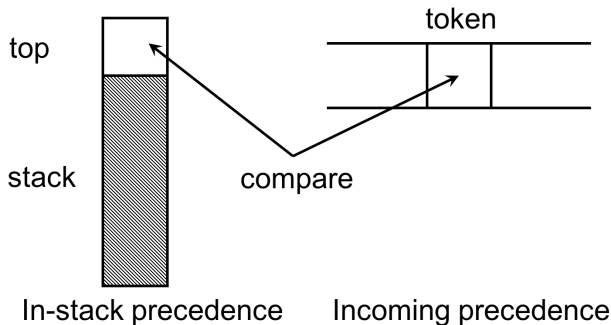
Infix to Postfix: Parenthesized expression

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

token	[0]	[1]	[2]	top	output
a				-1	a
*	*			0	a
(*	(1	a
b	*	(1	ab
+	*	(+	2	ab
c	*	(+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

Infix to Postfix

A precedence-based scheme for stacking and unstacking operators



$\text{isp}[\text{stack}[\text{top}]] < \text{icp}[\text{token}] : \text{push}$

$\text{isp}[\text{stack}[\text{top}]] \geq \text{icp}[\text{token}] : \text{pop and print}$

Infix to Postfix

Use two types of precedence (because of the '(' operator)

- in-stack precedence (isp)
- incoming precedence (icp)

```
1 precedence stack[MAX_STACK_SIZE];
2 /* isp and icp arrays
3    -- index is value of precedence
4       lparen, rparen, plus, minus,
5       times divide, mode, eos */
6
7 static int isp[] = {0, 19, 12, 12, 13, 13, 13, 0};
8 static int icp[] = {20, 19, 12, 12, 13, 13, 13, 0};
```

Infix to Postfix: the function I

Function to convert from infix to postfix

```
1 void postfix(void) {  
2     char symbol;  
3     precedence token;  
4     int n = 0;  
5     int top = 0;  
6     stack[0] = eos;  
7  
8     for (token = get_token(&symbol, &n);  
9         token != eos;  
10        token = get_token(&symbol, &n)) {  
11  
12        if (token == operand)  
13            printf("% c", symbol);  
14        else if (token == rparen) {  
15            while (stack[top] != lparen)  
16                print_token(pop(&top));  
17  
18            pop(&top);
```

Infix to Postfix: the function II

```
19         } else {
20             while (isp[stack[top]] >= icp[token])
21                 print_token(pop(&top));
22
23             push(&top, token);
24         }
25     }
26     while ((token = pop(&top)) != eos)
27         print_token(token);
28
29     printf("\n");
30 }
```

Infix to Postfix

postfix

- no parenthesis is needed
- no precedence is needed

complexity

- time: $O(r)$ where r : number of symbols in expression
- space: $S(n) = n$ where n : number of operators

MULTIPLE STACKS QUEUES

Multiple Stacks and Queues

Multiple Stacks

We need n stacks simultaneously

- maximum size of each stack is unpredictable
- size of each stack is dynamically varying
- efficient memory utilization for multiple stacks is difficult

Multiple Stacks and Queues

Sequential mappings of stacks into an array

○ `memory[MEM_SIZE]` **case $n = 2$**

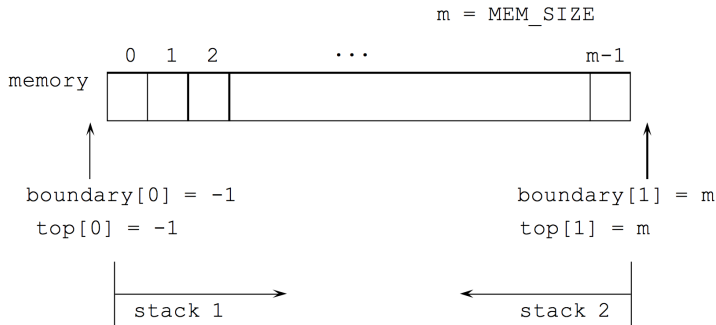
The first stack:

- bottom element: `memory[0]`
- grow toward `memory[MEM_SIZE-1]`

The second stack:

- bottom element: `memory[MEM_SIZE-1]`
- grow toward `memory[0]`

Multiple Stacks and Queues



$\text{top}[0] = -1$: stack 1 is empty

$\text{top}[1] = m$: stack 2 is empty

Multiple Stacks and Queues: case $n \geq 3$

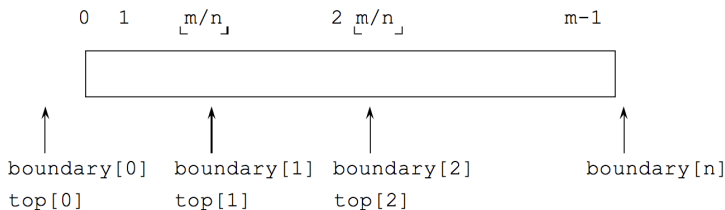
- *boundary*: point to the position immediately to the left of the bottom element
- *top*: point to the top element

```
1 #define MEM_SIZE 100
2 #define MAX_STACKS 10
3
4 element memory[MEM_SIZE];
5
6 int top[MAX_STACKS];
7 int boundary[MAX_STACKS];
8 int n; /* number of stacks, n < MAX_STACKS */
```

Multiple Stacks and Queues: case $n \geq 3$

divide the array into roughly equal segments

```
1 top[0] = boundary[0] = -1;
2 for(i = 1; i < n; i++)
3     top[i] = boundary[i] = (MEM_SIZE/n)*i - 1;
4 boundary[n] = MEM_SIZE - 1;
```



initial configuration for n stacks in memory $[m]$

Multiple Stacks and Queues: case $n \geq 3$

initially

$\text{boundary}[i] = \text{top}[i] = \lfloor m/n \rfloor * i - 1$

i^{th} stack is empty

- $\text{top}[i] == \text{boundary}[i]$

i^{th} stack is full

- $\text{top}[i] == \text{boundary}[i+1]$

Multiple Stacks and Queues: case $n \geq 3$

push an item to the i^{th} stack

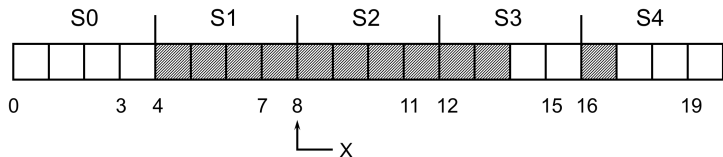
```
1 void push(int i, element item) {  
2     /* add an item to the i-th stack */  
3     if (top[i] == boundary[i+1])  
4         stack_full(i);  
5     memory[++top[i]] = item;  
6 }
```

pop an item from the i^{th} stack

```
1 element pop(int i) {  
2     /* remove top element from the i-th stack */  
3     if (top[i] == boundary[i])  
4         return stack_empty(i);  
5     return memory[top[i]--];  
6 }
```

Multiple Stacks and Queues: case $n \geq 3$

Ex. $n = 5$, $m = 20$, $\text{push}(1, x)$



$b[0] = -1$ $b[1] = 3$ $b[2] = 7$ $b[3] = 11$ $b[4] = 15$

$t[0] = -1$ $t[1] = 7$ $t[2] = 11$ $t[3] = 13$ $t[4] = 16$

Multiple Stacks and Queues: case $n \geq 3$

1. find the least j for $i < j < n$ such that there is a free space between stacks j and $(j+1)$
 - i.e.) $t[j] < b[j+1]$
 - move stack $i+1, i+2, \dots, j$ one position to the right creating a space between stacks i and $(i+1)$
2. if there is no such a j , then look up the left direction
3. data movement: $O(m)$