Data Structure and Algorithm

Class 5

Seongjin Lee

Updated: 2021-02-14
DSA_2017_05

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

Table of contents

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

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

- 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 $\leftarrow$ top

Stack Abstract Data Type

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

- 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 $\quad\rightarrow$ push B

stack state

		B	$\leftarrow$ top
A	$\leftarrow$ top	A	

Stack Abstract Data Type

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

- 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 $\rightarrow$ push C

stack state

				C	$\leftarrow$ top
		B	$\leftarrow$ top	B	
A	$\leftarrow$ top	A		A	

Stack Abstract Data Type

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

- 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 $\rightarrow$ push C $\rightarrow$ push D

stack state

$$
\begin{array}{llllllll}
 & & & & & & D & \leftarrow \text{top} \\
 & & & & C & \leftarrow \text{top} & C & \\
 & & B & \leftarrow \text{top} & B & & B & \\
A & \leftarrow \text{top} & A & & A & & A &
\end{array}
$$

Stack Abstract Data Type

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

- 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 $\rightarrow$ push C $\rightarrow$ push D $\rightarrow$ push E

stack state

								E	$\leftarrow$ top
						D	$\leftarrow$ top	D	
				C	$\leftarrow$ top	C		C	
		B	$\leftarrow$ top	B		B		B	
A	$\leftarrow$ top	A		A		A		A	

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

- 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 $\rightarrow$ push C $\rightarrow$ push D $\rightarrow$ push E $\rightarrow$ pop E

stack state

								E	$\leftarrow$ top		
						D	$\leftarrow$ top	D		D	$\leftarrow$ top
				C	$\leftarrow$ top	C		C		C	
		B	$\leftarrow$ top	B		B		B		B	
A	$\leftarrow$ top	A		A		A		A		A	

Stack Abstract Data Type: System Stack

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

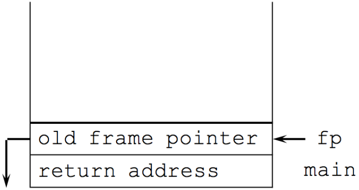
Activation record (stack frame) initially contains only

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

If this invokes another function

- local variables
- parameters of the invoking function

System Stack after function call

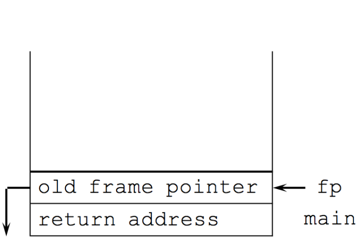


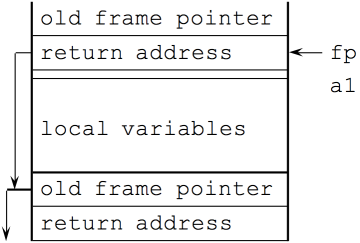
Stack Abstract Data Type: System Stack

System Stack after function call

Run-time program simply creates a new stack frame

- also for each recursive call





Stack Abstract Data Type: System Stack

Frequent function calls may occupy all of the stack memory and may cause a stack overflow.

The source of the picture is the “Stack Overflow” company’s logo.

Stack Abstract Data Type

```
Structure: Stack is
   Objects: a finite ordered list with zero or more elements
   Functions:
   For all stack ∈ Stack,
   item ∈ element,
   max_stack_size ∈ positive integer:
       Stack CreateS(max_stack_size);
       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

```
#define MAX_STACK_SIZE 100
typedef struct {
    int key; // only key field
             // can add/modify fields to meet the requirements
} element;

element stack[MAX_STACK_SIZE];
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 non-recursive procedures to recursive procedures
- Save return address when calling subfunction

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 $\leftarrow$ front, rear

Queue Abstract Data Type: Insertion and Deletion

Operation	insert A	$\rightarrow$ insert B
Queue state		

		B	$\leftarrow$ rear
A	$\leftarrow$ front, rear	A	$\leftarrow$ front

Queue Abstract Data Type: Insertion and Deletion

Operation insert A $\rightarrow$ insert B $\rightarrow$ insert C

Queue state

				C	$\leftarrow$ rear
		B	$\leftarrow$ rear	B	
A	$\leftarrow$ front, rear	A	$\leftarrow$ front	A	$\leftarrow$ front

Queue Abstract Data Type: Insertion and Deletion

Operation insert A $\rightarrow$ insert B $\rightarrow$ insert C $\rightarrow$ insert D

Queue state

						D	$\leftarrow$ rear
				C	$\leftarrow$ rear	C	
		B	$\leftarrow$ rear	B		B	
A	$\leftarrow$ front, rear	A	$\leftarrow$ front	A	$\leftarrow$ front	A	$\leftarrow$ front

Queue Abstract Data Type: Insertion and Deletion

Operation	insert A	$\rightarrow$ insert B	$\rightarrow$ insert C	$\rightarrow$ insert D	$\rightarrow$ delete A
Queue state					
				D $\leftarrow$ rear	D $\leftarrow$ rear
			C $\leftarrow$ rear	C	C
		B $\leftarrow$ rear	B	B	B $\leftarrow$ front
	A $\leftarrow$ front, rear	A $\leftarrow$ front	A $\leftarrow$ front	A $\leftarrow$ front	

Queue Abstract Data Type: Implementation

Simplest scheme

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

```
#define MAX_QUEUE_SIZE 100
typedef struct {
    int key;
    /* other fields */
} element;

element queue[MAX_QUEUE_SIZE];
int rear = -1;
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: Application of Queue

- Buffer
- Job scheduling

addq(*prear, element item)

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

deleteq(*pfront, int rear)

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

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(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(MAX_QUEUE_SIZE)

Solution:

- Circular Queue

Circular Queues

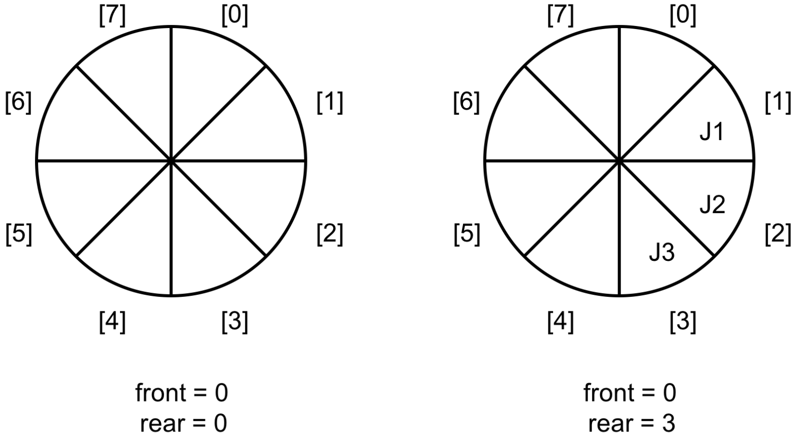
Circular Queues

More efficient Queue representation

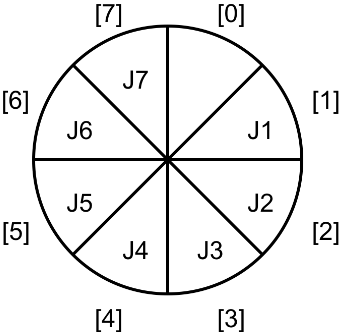
- regard the array `queue[MAX_QUEUE_SIZE]` as circular
- initially `front` and `rear` to `0` rather than `-1`
- the `front` index always points one position counterclockwise from the first element in the queue
- the `rear` index point to the current end of the queue

Circular Queues

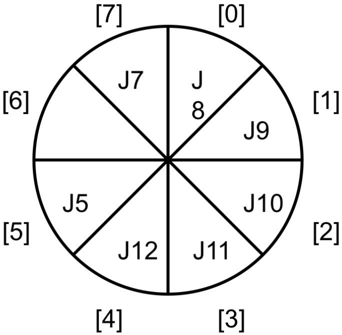
empty and nonempty circular queues



full circular queues



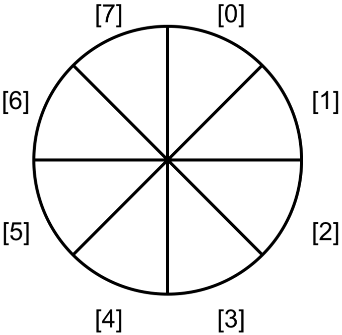
front = 0
rear = 7



front = 6
rear = 5

Circular Queues: Quiz

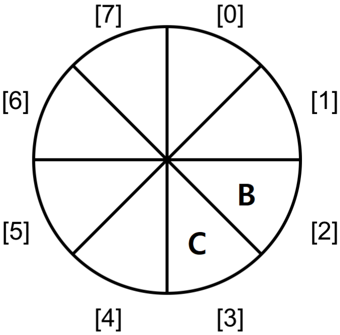
What is the result ?



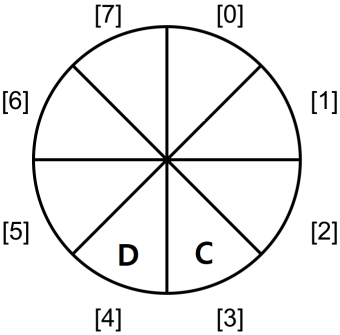
front = 0
rear = 0

```
addq(A)
addq(B)
addq(C)
deleteq()
addq(D)
deleteq()
```

Circular Queues: Quiz Result



addq(A)
addq(B)
addq(C)
deleteq()



addq(D)
deleteq()

front = 2
rear = 4

Circular Queues: Thinking

Why do not fill data in the empty space in front?

- If all the data is filled in such a state, front and rear become the same, so it is impossible to distinguish whether it is empty or full.
- So, before all queue is filled, the size of the queue must be expanded.

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

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

Circular Queues: Implementation

Delete from a circular queue

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

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 `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

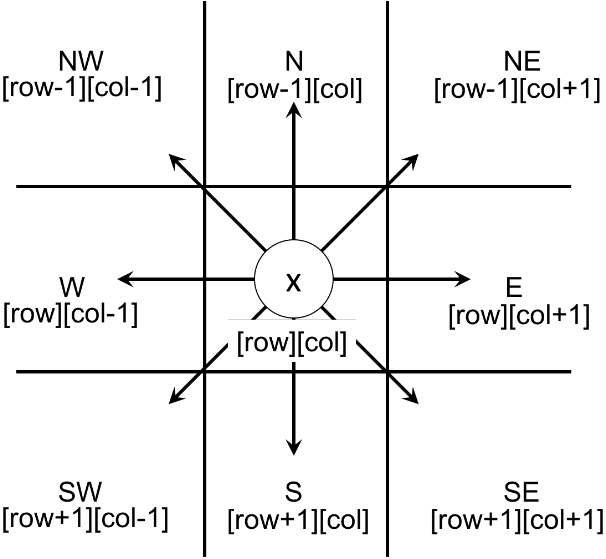
- two-dimensional array
- element 0 : open path
- element 1 : barriers

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*p` maze

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

A Mazing Problem: Data Type

```c
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

Position of next move

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

```
next_row = row + move[dir].vert;
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

```
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 the stack;
    while (there are more moves from current position) {
        <next_row, next_col> = coordinates of next move;
        dir = direction of move;
        if ((next_row == EXIT_ROW) &&
            (next_col == EXIT_COL))
            success;
        if (maze[next_row][next_col] == 0 &&
            mark[next_row][next_col] == 0) {
            mark[next_row][next_col] = 1;
            add <row, col, dir> to the top of the stack;
            row = next_row;
            col = next_col;
            dir = north;
        }
    }
}
printf("no path found\n");
```

A Mazing Problem: Data Type

```
#define MAX_STACK_SIZE 100
typedef struct {
    short int row;
    short int col;
    short int dir;
} element;
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	15	right-to-left
(type)	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
$+2\ *3\ 4$	$2+3*4$	$2\,3\,4\ *+$
$+*a\ b\ 5$	$a*b+5$	$a\ b\ *5+$
$*+1\,2\,7$	$(1+2)*7$	$1\,2\ +7*$
$/*a\ b\ c$	$a*b/c$	$a\ b\ *c\ /$
$*/a\ +-b\ c\ d\ *-e\ a\ c$	$((a/(b-c+d))*(e-a)*c$	$a\ b\ c\ -d\ +/e\ a\ -*c\ *$
$+-/a\ b\ c\ -*d\ e\ *a\ c$	$a/b-c+d*e-a*c$	$a\ b\ /c\ -d\ e\ *+a\ c\ *-$

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

How many times integer pushes have occurred ?

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

```c
#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,
              mode, eos, operand
             } precedence;

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

```
int eval(){
    precedence token;
    char symbol;

    int op1, op2;

    int n = 0;
    int top = -1;

    token = get_token(&symbol, &n);

    while (token != eos) {
        if (token == operand)
            push(&top, symbol-'0');
        else {
            op2 = pop(&top);
            op1 = pop(&top);

            switch (token) {
                case plus: push(&top, op1+op2);
```

Function to evaluate a postfix expression II

```
                        break;
                case minus: push(&top, op1-op2);
                        break;
                case times: push(&top, op1*op2);
                        break;
                case divide: push(&top, op1/op2);
                        break;
                case mod: push(&top, op1%op2);
            }
        }
        token = get_token(&symbol, &n);
    }
    return pop(&top);
}
```

Function to get a token I

```
precedence get_token(char *psymbol, int *pn) {
    *psymbol = expr[(*pn)++];
    switch (*psymbol)
        case ‘(‘ : return lparen;
        case ‘)‘ : return rparen;
        case ‘+‘ : return plus;
        case ‘-‘ : return minus;
        case ‘*‘ : return times;
        case ‘/‘ : return divide;
        case ‘%‘ : return mod;
        case ‘ ‘ : return eos;
        default : return operand; /* no error checking */
    }
}
```

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

- 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

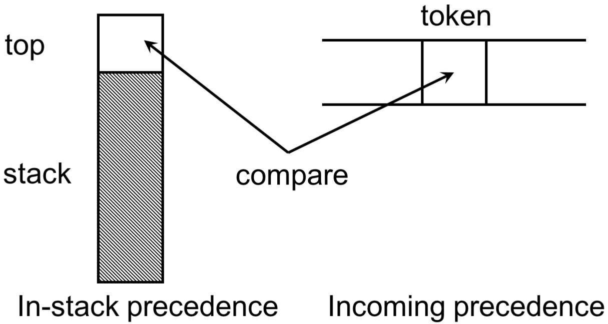
Infix to Postfix: Parenthesized expression

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



`isp[stack[top]] < icp[token]` : push

`isp[stack[top]]` $\geq$ `icp[token]` : pop and print

Infix to Postfix

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

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

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays
   -- index is value of precedence
      lparen, rparen, plus, minus,
      times divide, mode, eos */

static int isp[] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp[] = {20, 19, 12, 12, 13, 13, 13, 0};
```

Function to convert from infix to postfix

```
void postfix(void) {
    char symbol;
    precedence token;
    int n = 0;
    int top = 0;
    stack[0] = eos;

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

        if (token == operand)
            printf("% c", symbol);
        else if (token == rparen) {
            while (stack[top] != lparen)
                print_token(pop(&top));

            pop(&top);
```

Infix to Postfix: the function II

```
        } else {
            while (isp[stack[top]] >= icp[token])
                print_token(pop(&top));

            push(&top, token);
        }
    }
    while ((token = pop(&top)) != eos)
        print_token(token);

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

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