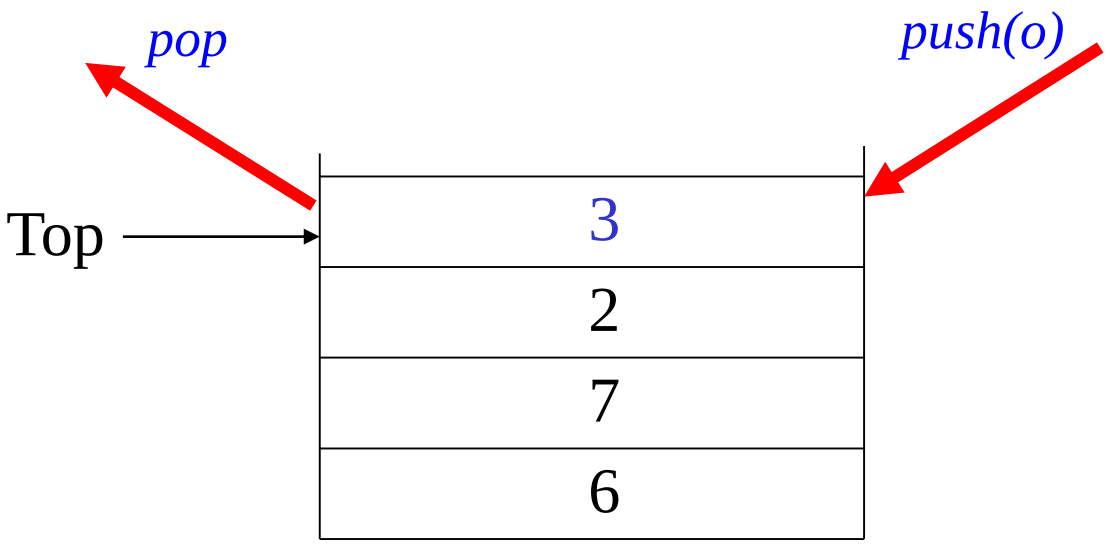
Stacks

What is a Stack?

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*.
- The operations: push (insert) and pop (delete)



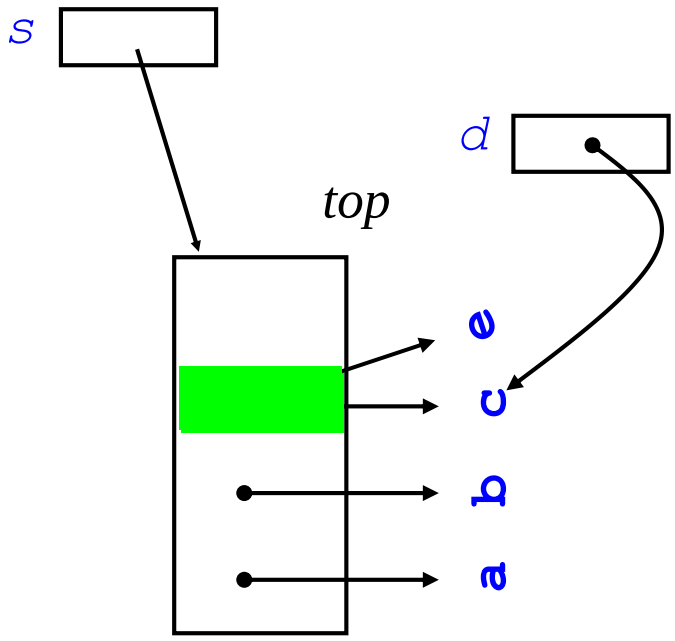
Stack ADT Interface

- The main functions in the Stack are

```
boolean isEmpty();          // return true if empty
boolean isFull(S);          // return true if full
void push(S, item);         // insert item into stack
void pop(S);                // remove most recent item
void clear(S);              // remove all items from stack
Item peek(S);               // retrieve most recent item
```

Sample Operation

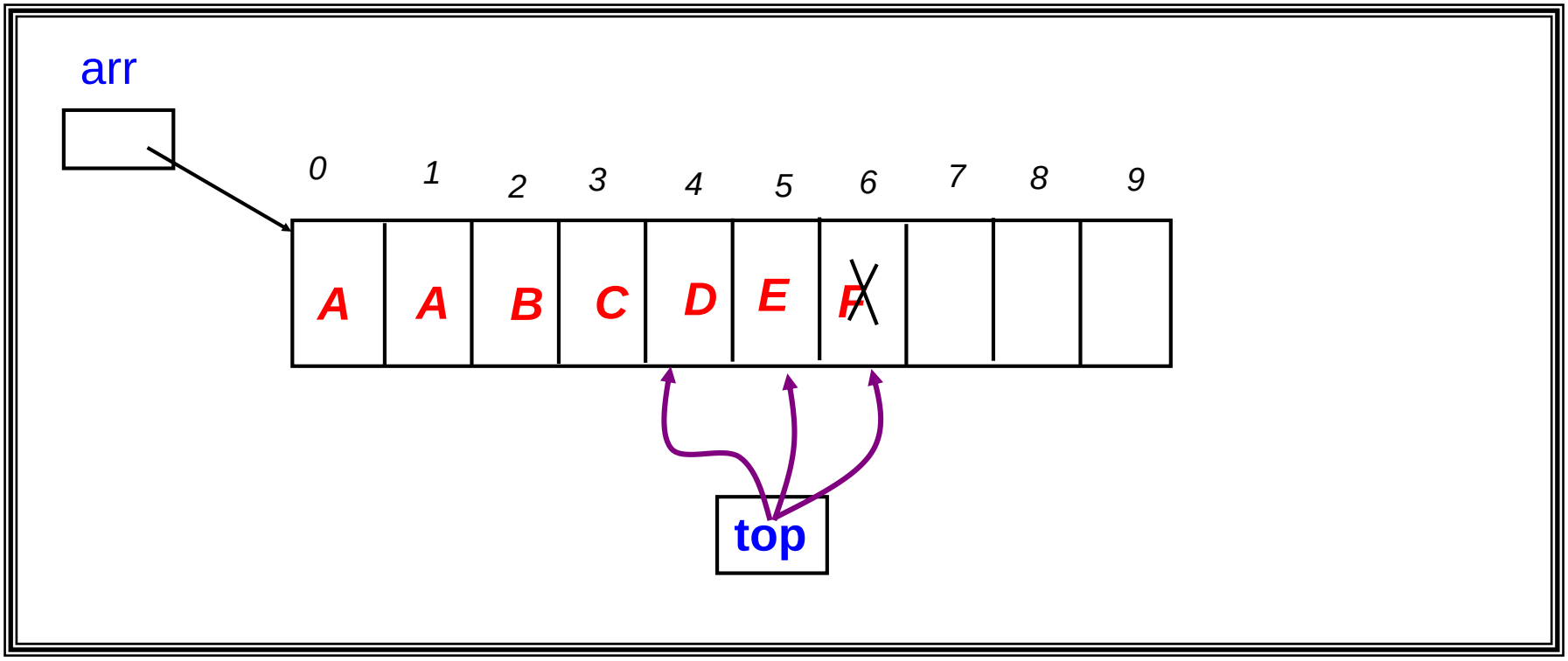
- `Stack S = malloc(sizeof(stack));`
- `push(S, "a");`
- `push(S, "b");`
- `push(S, "c");`
- `d=top(S);`
- `pop(S);`
- `push(S, "e");`
- `pop(S);`



Implementation by Array

- use Array with a top index pointer as an implementation of stack

StackAr



Code

```
typedef struct {
    int A[MAX];
    int top;
} STACK;
```

```
void clear(STACK *pS)
{
    pS->top = -1;
}

BOOLEAN isEmpty(STACK *pS)
{
    return (pS->top < 0);
}

BOOLEAN isFull(STACK *pS)
{
    return (pS->top >= MAX-1);
}
```

Deleting from a Stack

```
BOOLEAN pop(STACK *pS, int *px)
{
     if (isEmpty(pS))
          return FALSE;
     else {
          (*px) = pS->A[(pS->top)--];
          return TRUE;
     }
}
```

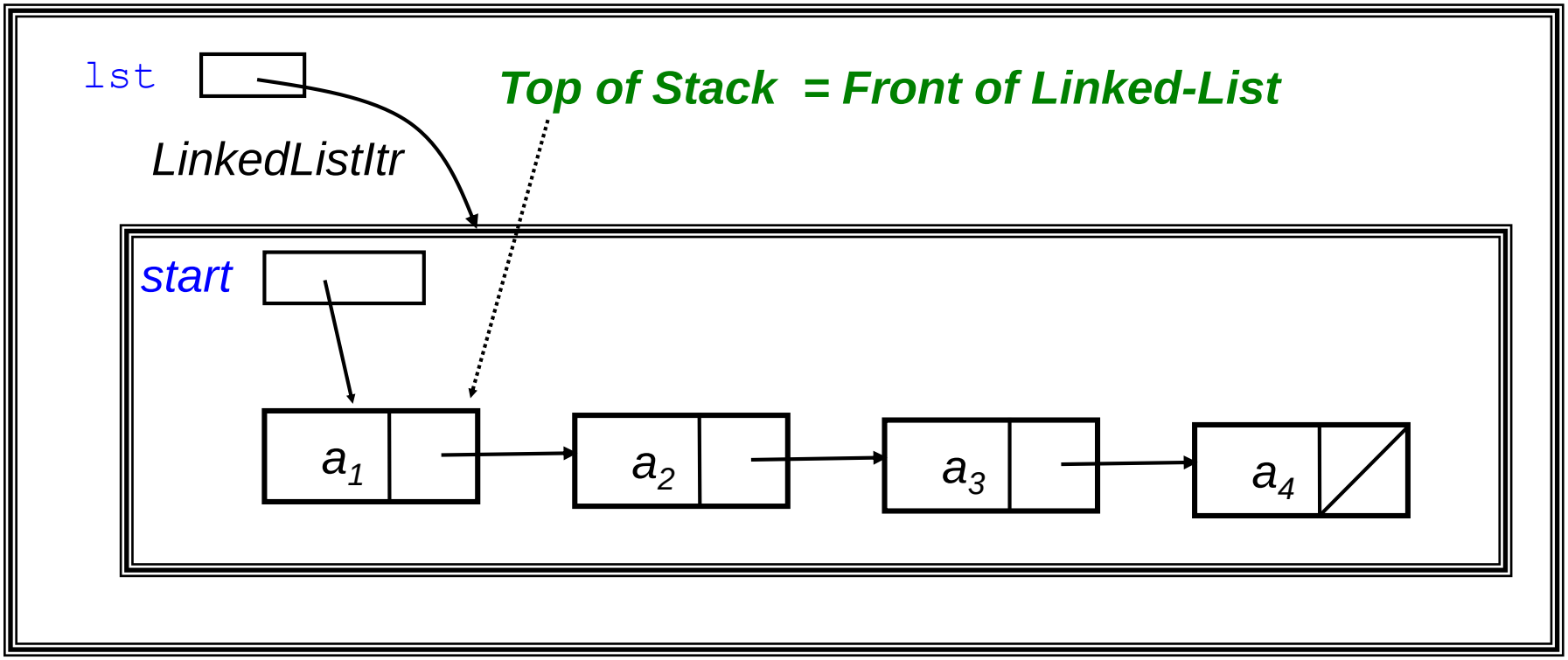
Inserting data into a stack

```
BOOLEAN push(int x, STACK *pS)
{
    if (isFull(pS))
        return FALSE;
    else {
        pS->A[++(pS->top)] = x;
        return TRUE;
    }
}
```

Implementation by Linked Lists

- Can use a Linked List as implementation of stack

StackLL



Code

```
struct Node {
    int element;
    Node * next;
};
typedef struct Node * STACK;
```

```
void clear(STACK *pS)
{
    (*pS) = NULL;
}

BOOLEAN isEmpty(STACK *pS)
{
    return ((*pS) == NULL);
}

BOOLEAN isFull(STACK *pS)
{
    return FALSE;
}
```

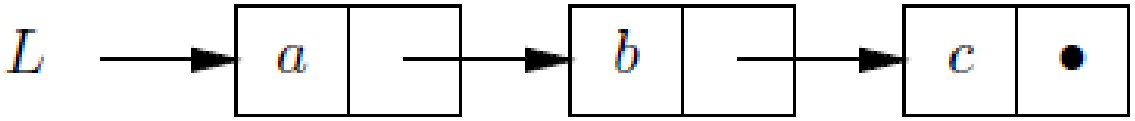
More code

```
BOOLEAN pop(STACK *pS, int *px)
{
    if ((*pS) == NULL)
        return FALSE;
    else {
        (*px) = (*pS)->element;
        (*pS) = (*pS)->next;
        return TRUE;
    }
}
```

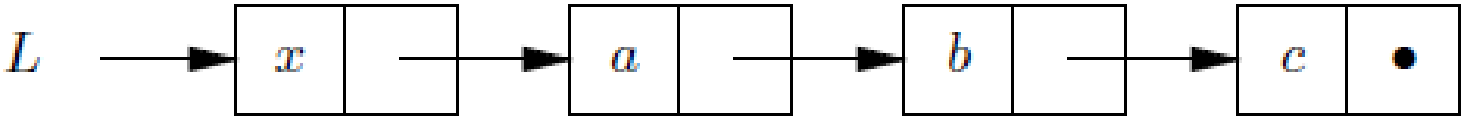
More Code

```
BOOLEAN push(int x, STACK *pS)
{
    STACK newCell;

    newCell = (STACK) malloc(sizeof(struct CELL));
    newCell->element = x;
    newCell->next = (*pS);
    (*pS) = newCell;
    return TRUE;
}
```

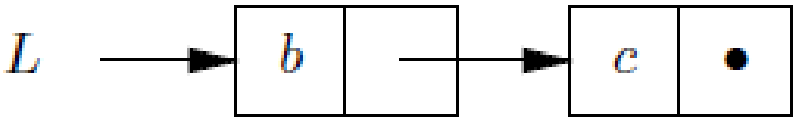
Effects



(a) List L.



(b) After executing $push(x, L)$.



(c) After executing $pop(L, x)$ on list L of (a).

Applications

- Many application areas use stacks:
  - function call stack
  - bracket matching
  - postfix calculation
  - Infix to postfix conversion

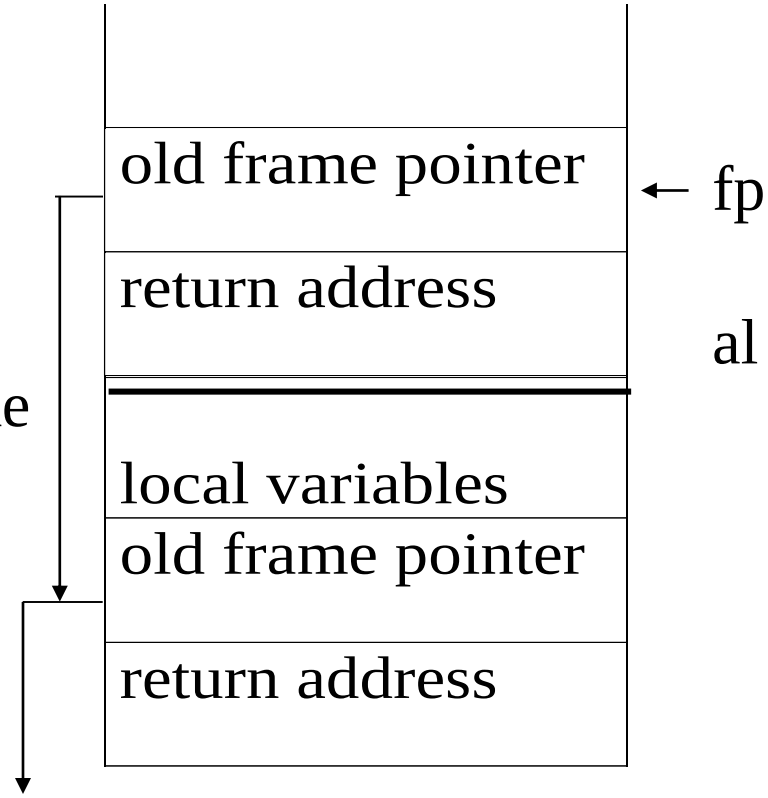
Function Call Stack

(activation record)

```
main()          a1( )
{               {
 a1();           …….;
statements;     …….;
}               }
```

fp: a pointer to current stack frame

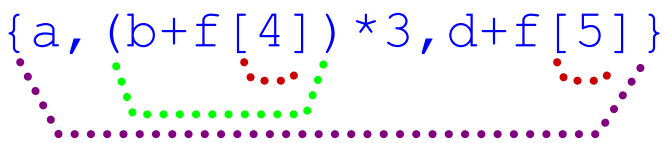
system stack before a1 is invoked



system stack after a1 is invoked

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - An Example:



  - Bad Examples:

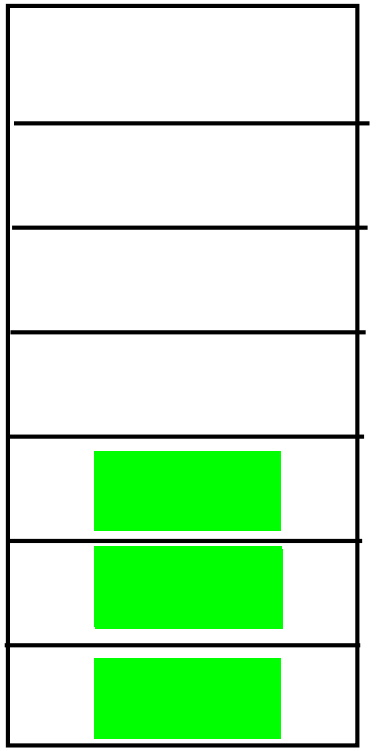
```
(..)..)          // too many closing brackets

(..(..)          // too many open brackets

[..(..]..)       // mismatched brackets
```

Informal Procedure

Initialize the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        return & remove most recent item
            from *the stack*
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

```
{a,(b+f[4])*3,d+f[5]}
```

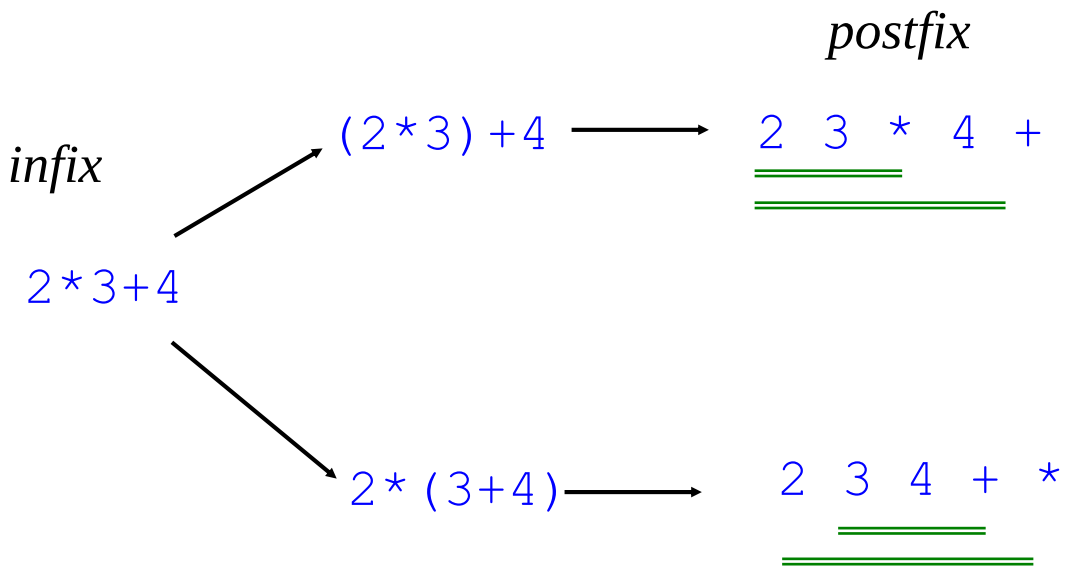


Stack

Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**



Evaluation of Expressions

- **operator precedence**
- **operator associativity**

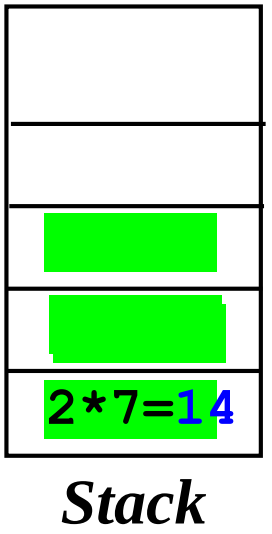
token	precedence	associativity
() [] -> .	17	left-to-right
-- ++	16	left-to-right
-- ++ ! ~ - + & * sizeof	15	right-to-left
(type)	14	right-to-left
* / %	13	left-to-right
+ -	12	left-to-right
<< >>	11	left-to-right
> >= < <=	10	left-to-right
== !=	9	left-to-right
&	8	left-to-right
^	7	left-to-right
|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

precedence hierarchy for C language

Informal Procedure

Initialise stack S
For each item read.
    If it is an operand,
        push on the stack
    If it is an operator,
        pop arguments from stack;
        perform operation;
        push result onto the stack

Expr	
2	`push(S, 2)`
3	`push(S, 3)`
4	`push(S, 4)`
+	`arg2=topAndPop(S)`
	`arg1=topAndPop(S)`
	`push(S, arg1+arg2)`
*	`arg2=topAndPop(S)`
	`arg1=topAndPop(S)`
	`push(S, arg1*arg2)`



Postfix Evaluation

infix	postfix
2+3*4	2 3 4*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

infix and postfix notation

Evaluating Postfix Expression

```
6 2/3-4 2*+
```

token	stack [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

```
postfix evaluation
```

Infix to Postfix

Method 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

Eg a/b-c+d*e-a*c

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

ab/c-de*+ac*-

- requires two passes

Stack can also be used to convert infix to postfix

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

translation of a+b*c to postfix

Infix to Postfix expression

token	stack [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*

translation of a*(b+c)*d to postfix

Procedure Infix to Postfix

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

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

1. Read the infix expression as a string.
2. Scan the expression character by character till the end. Repeat the following operations
   a. If it is an operand ,display it
   b. If it is a left parenthesis ,push it onto the stack.
   c. If it is a right parenthesis, pop out elements from the stack until left parenthesis and display them

      Pop out the left parenthesis but don't display .
   d. If it is an operator compare its precedence with that of the operator at the top of stack.
      i. If it is greater push it onto the stack.
      ii. Else pop and display the elements in the stack until the precedence of operator on stack top is less than precedence of incoming operator precedence
      iii. Then push the incoming operator
3. Pop out any leftover elements in the stack and display them

Summary

- The ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - a strong relationship between recursion and stacks exists
- Stack can be implemented using arrays or linked lists