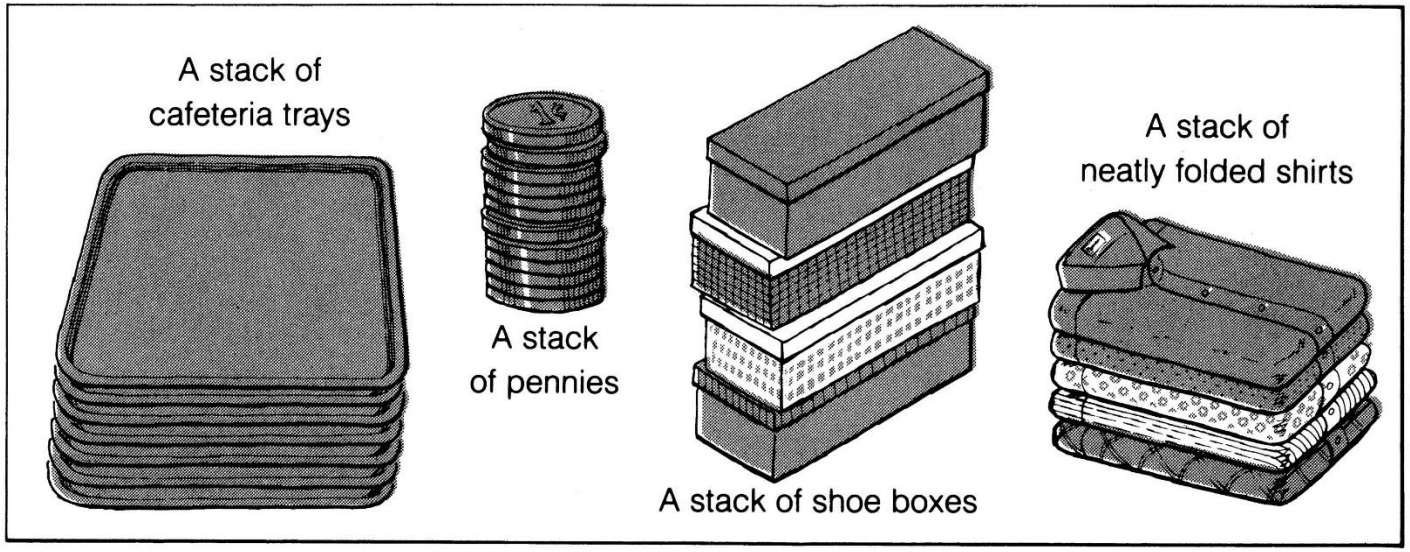
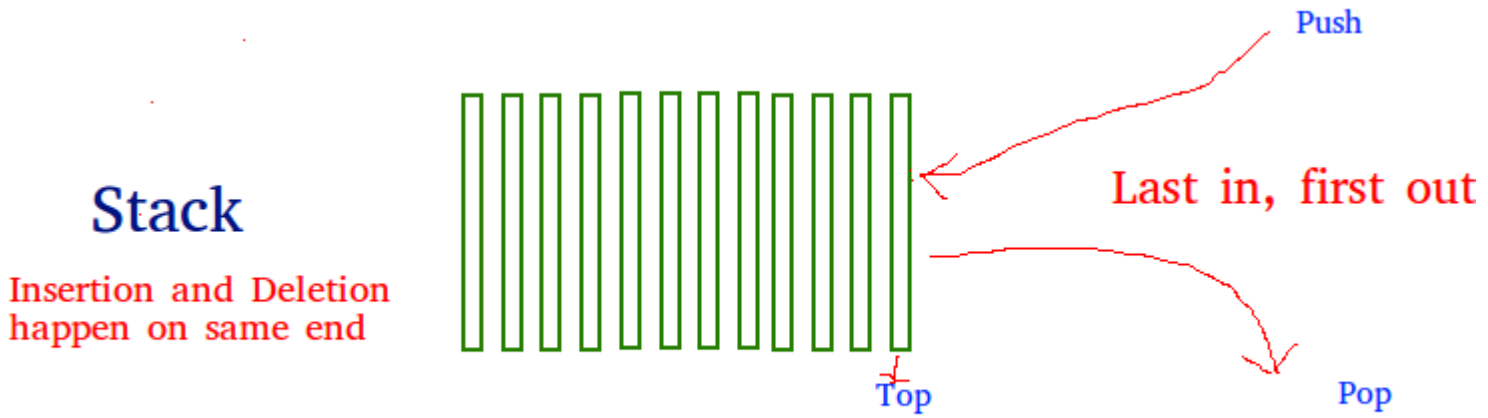
# What is a stack?

- It is an ordered group of homogeneous items of elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



# STACK

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).



# Stack as ADT

- A Stack contains elements of same type arranged in sequential order.
- All operations takes place at a single end that is top of the stack and following operations can be performed:

❑push() – Insert an element at one end of the stack called top.

❑pop() – Remove and return the element at the top of the stack, if it is not empty.

❑push() – Return the element at the top of the stack without removing it, if the stack is not empty.

❑size() – Return the number of elements in the stack.

❑isEmpty() – Return true if the stack is empty, otherwise return false.

❑isFull() – Return true if the stack is full, otherwise return false.

```
#include <iostream>
using namespace std;
#define MAX 1000
class Stack {
            int top;
  public:int a[MAX]; // Maximum size of Stack
          Stack() { top = -1; }
          void push(int);
          int pop();
          int push();
          void display();
          bool isEmpty();
};
```

```
void Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
          }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
            }
}
```

```
int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
```

```
int Stack::push()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}
```

```
bool Stack::isEmpty()
{
    return (top < 0);
}
void Stack::display() {
   if(top>=0) {
      cout<<"Stack elements are:";
      for(int i=top; i>=0; i--)
         cout<<a[i]<<" ";
         cout<<endl;
   } else
      cout<<"Stack is empty";}
```

```
int main()
{
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";
    s.display();
    cout<<"The element at the top of stack is"<<s.push();
    return 0;
}
```

```
include <iostream>
using namespace std;

int main()
{
    int n, num, digit, rev = 0;
    cout << "Enter a positive number: ";
    cin >> num;
    n = num;
    do
    {
        digit = num % 10;
        rev = (rev * 10) + digit;
        num = num / 10;
    } while (num != 0);

    cout << " The reverse of the number is: " << rev << endl;

    if (n == rev)
        cout << " The number is a palindrome.";
    else
        cout << " The number is not a palindrome.";
   return 0;
}
```

## Algorithm for C++ Program to Reverse a String

•Step 1. Take the string which you have to reverse as the input variable says str.

•Step 2:- Calculate the length of the string. The actual length of the string is one less than the number of characters in the string. Let actual length be len.

•Step 3:- Use a for loop to iterate the string using a temporary variable to swap the elements.

•Step 4:- Print the reversed string.

- To check whether it's a palindrome ??

# Checking for palindrome

- Algorithm
- Scan the string from left to right till the end.
- push every char you encounter during the scan to the stack.
- Rescan the string left to right till end and do the following for every char you encounter during scan.
  - Pop one char and compare current char with the popped one.
  - If no match report immediately non palindrome other wise proceed to the next char.

- Class palindrome{

```
Char data[MAX],str[mAX];
Palindrome(){
        top=-1;
}
Void read();
Void palindromecheck();
Void extract();
Char pop();
};
```

# Algorithm for checking expression

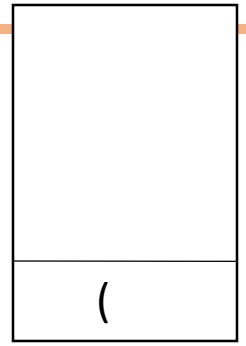
1.Scan the expression from left to right.

2.Whenever a scope opener( ‘(‘,’{‘,’[‘ ) is encountered while scanning the expression, it is pushed to stack.

3.Whenever as scope ender( ‘)’, ‘}’, ‘]’) is encountered, the stack is examined. If stack is empty, scope ender does not have a matching opener and hence string is invalid. If stack is non empty, we pop the stack and check whether the popped item corresponds to scope ender. If a match occurs, we continue. If it does not, the string is invalid.

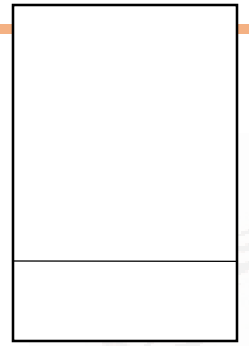
4. When the end of string is reached, the stack must be empty; otherwise 1 or more scopes have been opened which have not been closed and string is invalid.

Ex1:(a+b) * (c+d



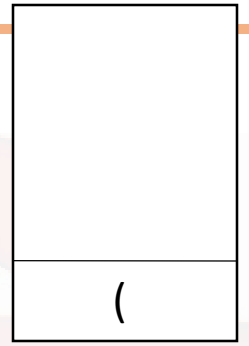

(

push ‘(‘



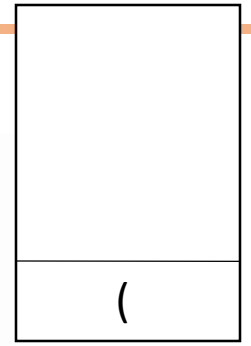
(a+b)

Pop ‘(‘ since there is ‘)’ & continue




(a+b) *(

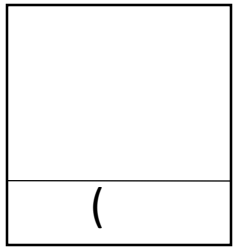
Push ‘(‘




(a+b) *(c+d

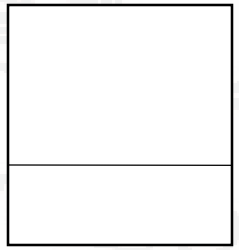
End of string reached but stack not empty. Hence invalid

Ex2:(a+b)*c+d)



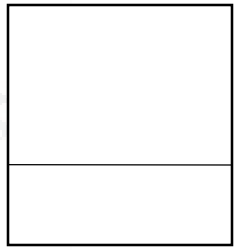

(

push ‘(‘



(a+b)

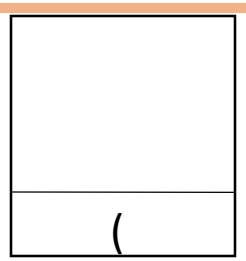
pop‘(‘ and continue



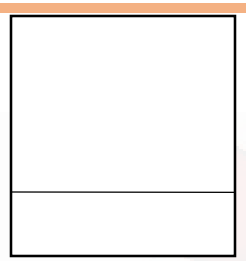
(a+b)*c+d)

Stack empty when ‘)’ encountered.hence invalid

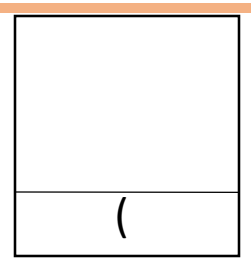
# Ex3: (a+b)*({c*d)



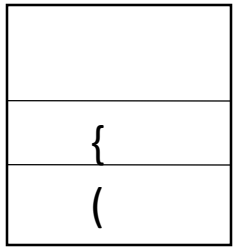

(
push ‘(‘



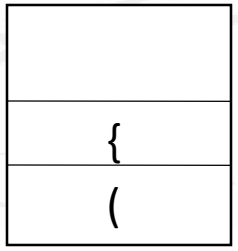
(a+b)
pop‘(‘ and
continue




(a+b)*(
push‘(‘ and
continue

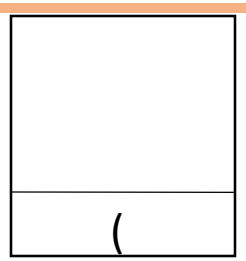



(a+b)*({
push ‘{ and
continue‘



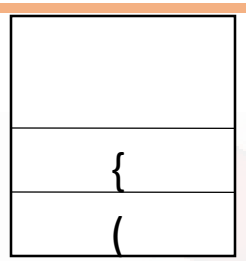

(a+b)*({c*d)
No match between closing scope ‘)’ and opening scope ‘{‘. Hence invalid.

# Ex4: (a+{b*c}+(c*d))



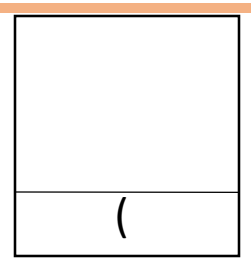

(

push ‘(‘ and continue



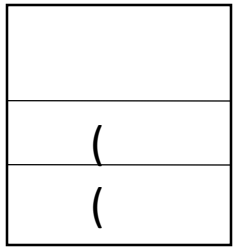

(a+{

push‘{‘ and continue



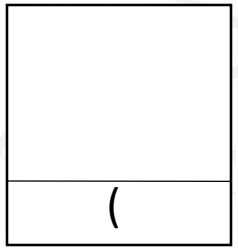

(a+{b*c}

pop‘{‘ and continue



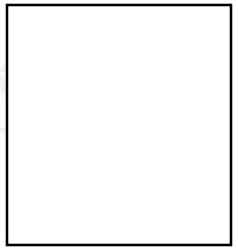

(a+{b*c}+(

push ‘( and continue‘




(a+{b*c}+(c*d)

Pop ‘(‘



(a+{b*c}+(c*d))

Pop ‘(‘.

End of string and stack empty. Hence valid

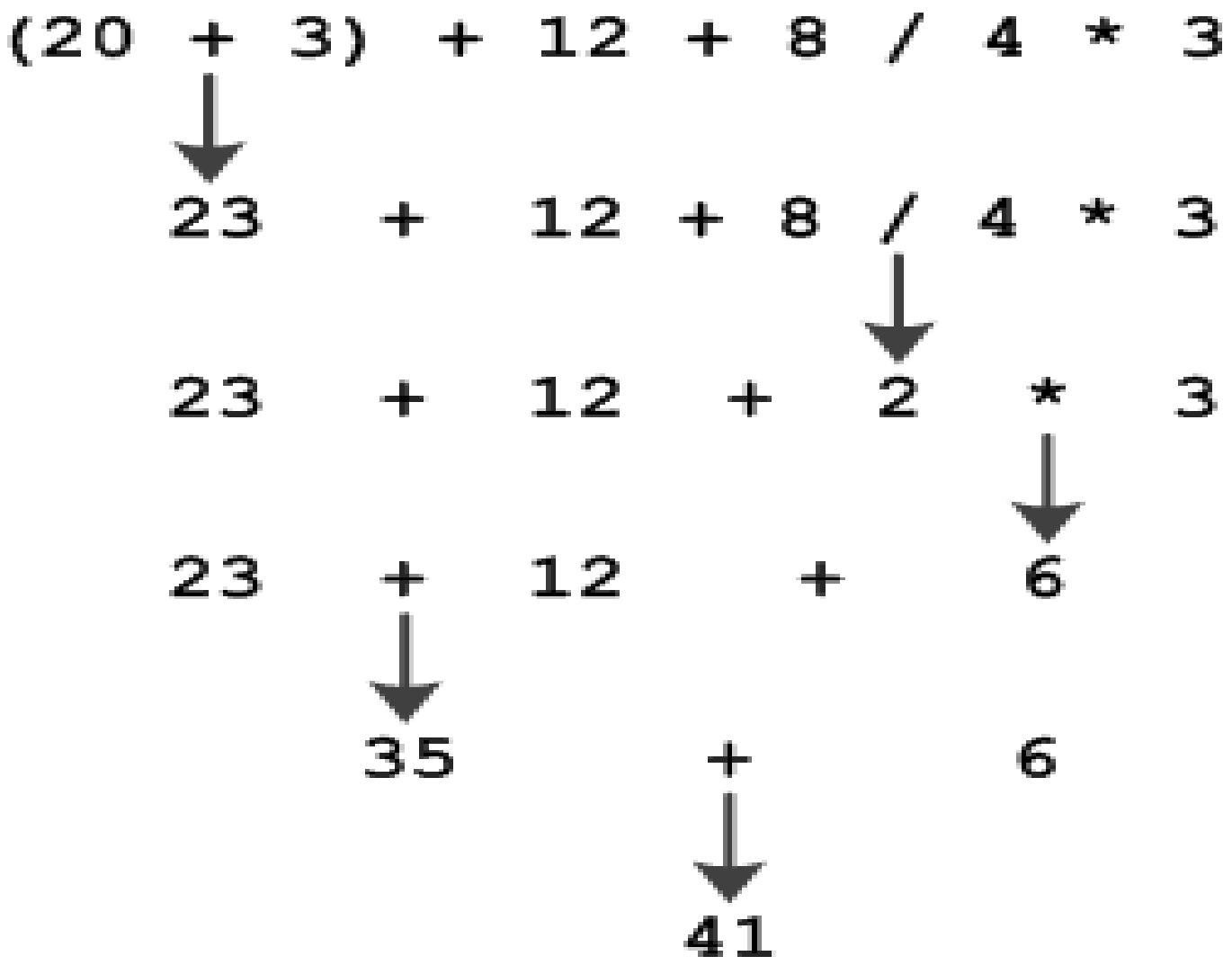
paren.txt

# Arithmetic Expressions

- Infix form
  - operand operator operand
    - 2+3 or
    - a+b
  - Need precedence rules
  - May use parentheses
    - 4*(3+5) or
    - a*(b+c)

# Arithmetic Expressions

- Postfix form
  - Operator appears **after** the operands
    - (4+3)*5 : 4 3 + 5 *
    - 4+(3*5) : 4 3 5 * +
  - No precedence rules or parentheses!
- Prefix Form
  - Operator appears before the operands
  - (4+3) : +43
- Input expression given in postfix form
  - How to evaluate it?


(20 + 3) + 12 + 8 / 4 * 3
23 + 12 + 8 / 4 * 3
23 + 12 + 2 * 3
23 + 12 + 6
35 + 6
41

# Precedence Rules

- **Arithmetic operators** follow the same **precedence** rules as in mathematics, and these are: exponentiation is performed first (when available), multiplication and division are performed next, addition and subtraction are performed last.

- A * B + C ➔ A B * C +

| | current symbol | operator stack content | postfix string |
|---|---|---|---|
| 1 | A | | A |
| 2 | * | * | A |
| 3 | B | * | A B |
| 4 | + | + | A B * {pop and print the '*' before pushing the '+'} |
| 5 | C | + | A B * C |
| 6 | '\0' | | A B * C + |

# Infix to postfix

**Steps:**

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, put it into postfix expression.

3. Else,

…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.

…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

After doing that Push the scanned operator to the stack.

(If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

| Infix Expression : A+B*(C^D-E) | | | | |
|---|---|---|---|---|
| **Token** | **Action** | **Result** | **Stack** | **Notes** |
| A | Add **A** to the result | A | | |
| + | Push **+** to stack | A | + | |
| B | Add **B** to the result | AB | + | |
| * | Push ***** to stack | AB | * + | * has higher precedence than + |
| ( | Push **(** to stack | AB | ( * + | |
| C | Add **C** to the result | ABC | ( * + | |
| ^ | Push **^** to stack | ABC | ^ ( * + | |
| D | Add **D** to the result | ABCD | ^ ( * + | |
| - | Pop **^** from stack and add to result | ABCD^ | ( * + | - has lower precedence than ^ |
| | Push **-** to stack | ABCD^ | - ( * + | |
| E | Add **E** to the result | ABCD^E | - ( * + | |
| ) | Pop **-** from stack and add to result | ABCD^E- | ( * + | Do process until ( is popped from stack |
| | Pop **(** from stack | ABCD^E- | * + | |
| | Pop ***** from stack and add to result | ABCD^E-* | + | Given expression is iterated, do Process till stack is not Empty, It will give the final result |
| | Pop **+** from stack and add to result | **ABCD^E-*+** | | |
| **Postfix Expression : ABCD^E-*+** | | | | |

# Infix to postfix

**Steps:**

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, put it into postfix expression.

3. Else,

…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.

…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.

After doing that Push the scanned operator to the stack.

(If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop and output from the stack until an '(' is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

- 3+4*5/6

Postfix 3

# Infix to prefix

- Expression = **(A+B^C)*D+E^5**

  **Step 1:** Reverse the infix expression.

  **5^E+D*)C^B+A(**

  **Step 2:** Make Every '(' as ')' and every ')' as '('

  **5^E+D*(C^B+A)**

IN_PRE.CPP

**Step 3:** Convert expression to postfix form.(While evaluating the operators with same precedence need not be popped out of the stack )

**5E^DCB^A+*+**

**Step 4:**Reverse the expression.

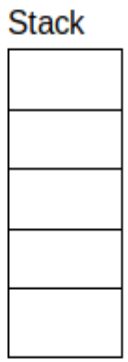
# Postfix to Infix

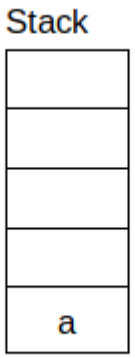
**Algorithm**

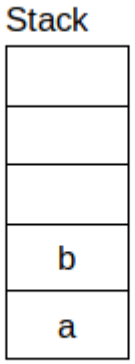
- While there are input symbol left
  - Read the next symbol from the input.
- If the symbol is an operand
  - Push it onto the stack
- Otherwise,
  - If the symbol is an operator.
  - Pop the  2 top values from the stack.(top1=op1,top2=op2, op2 opr op1)
- Put the operator, with the values as arguments and form a string.
- Push the resulted string back to stack.
- If there is only one value in the stack
  - That value in the stack is the desired infix string.

- Input postfix expression is ab+c*

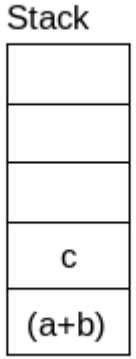
Initial empty stack



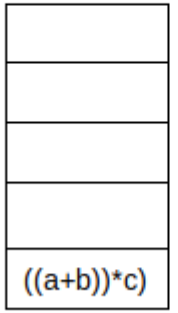




Popped operand a,b .
New string =(a+b)
Push this string into stack


Popped operand (a+b),c .
New string =((a+b)*c)
Push this string into stack



# Prefix to Infix

- Read the Prefix expression in reverse order (from right to left)
- If the symbol is an operand, then push it onto the Stack
- If the symbol is an operator, then pop two operands from the Stack
  Create a string by concatenating the two operands and the operator between them.
  **string = (top1(op1)+ operator + top2(op2))**
  And push the resultant string back to Stack
- Repeat the above steps until end of Prefix expression.

+a/*bc-de = (a+((b*c)/(d-e)))

NOTE: Read the prefix string in reverse order.

| SYMBOL | STACK |
|---|---|
| e | e |
| d | e \| d |
| - | (d-e) |
| c | (d-e) \| c |
| b | (d-e) \| c \| b |
| * | (d-e) \| (b*c) |
| / | ((b*c)/(d-e)) |
| a | ((b*c)/(d-e)) \| a |
| + | (a+((b*c)/(d-e))) |

Hence, the equivalent infix expression: (a+((b*c)/(d-e)))

# Postfix to Prefix

Postfix – Infix -Prefix

# Prefix to Postfix

- Prefix – Infix - postfix

# Evaluating Postfix Expressions

- Use a **stack,** assume binary operators +,*
- Input: postfix expression
- Scan the input
  - If operand,
    - **push** to stack
  - If operator
    - **pop** the stack twice
    - apply operator
    - **push** result back to stack

| Step | Input Symbol | Operation | Stack | Calculation |
|---|---|---|---|---|
| 1. | 4 | Push | 4 | |
| 2. | 5 | Push | 4,5 | |
| 3. | 6 | Push | 4,5,6 | |
| 4. | * | Pop(2 elements) & Evaluate | 4 | 5*6=30 |
| 5. | | Push result(30) | 4,30 | |
| 6. | + | Pop(2 elements) & Evaluate | Empty | 4+30=34 |
| 7. | | Push result(34) | 34 | |
| 8. | | No-more elements(pop) | Empty | 34(Result) |

- POSTFIX: 432*+5-

# Example

- **Input**

  5 9 8 + 4 6 * * 7 + *
- **Evaluation**

  push(5)

  push(9)

  push(8)

  push(pop() + pop()) /* be careful for ‘-’ */

  push(4)

  push(6)

  push(pop() * pop())

  push(7)

  push(pop() + pop())

  push(pop() * pop())

  print(pop())
- **What is the answer?**

# Exercise

- Input

  6 5 2 3 + 8 * + 3 + *

- Input

  a b c * + d e * f + g * +

- For each of the previous inputs
  - Find the infix expression

- 6 5 2 3 + 8 * + 3 + * Ans  288 Verify it.