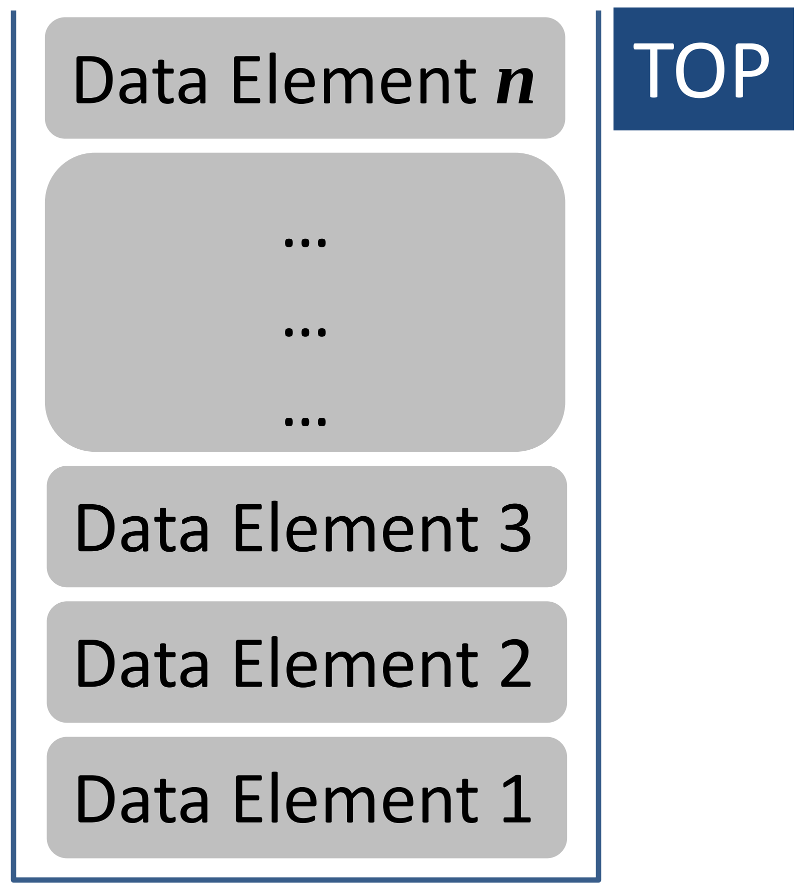
Stacks

Introduction

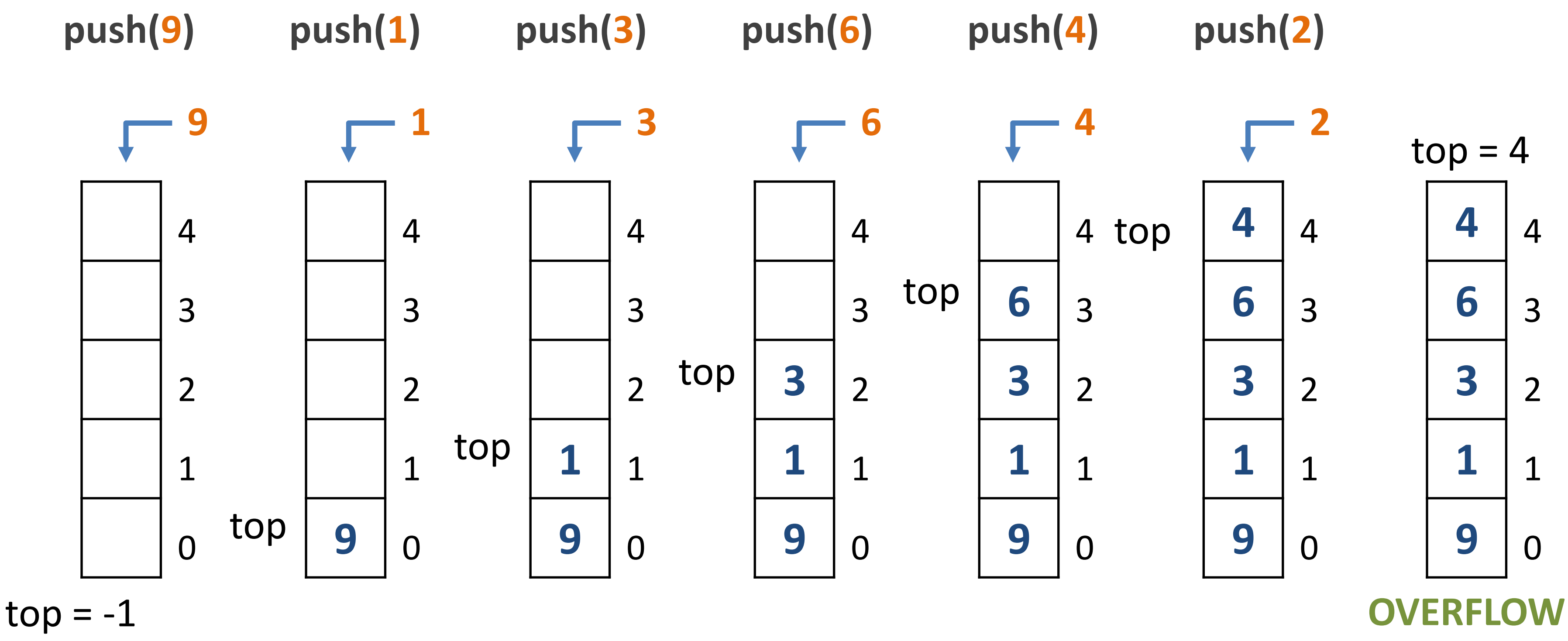
- Non primitive linear data structure.
- Allows operations at one end only.
- The top element can only be accessed at any time.
  - LIFO (Last-in-first-out) data structure.



Operations

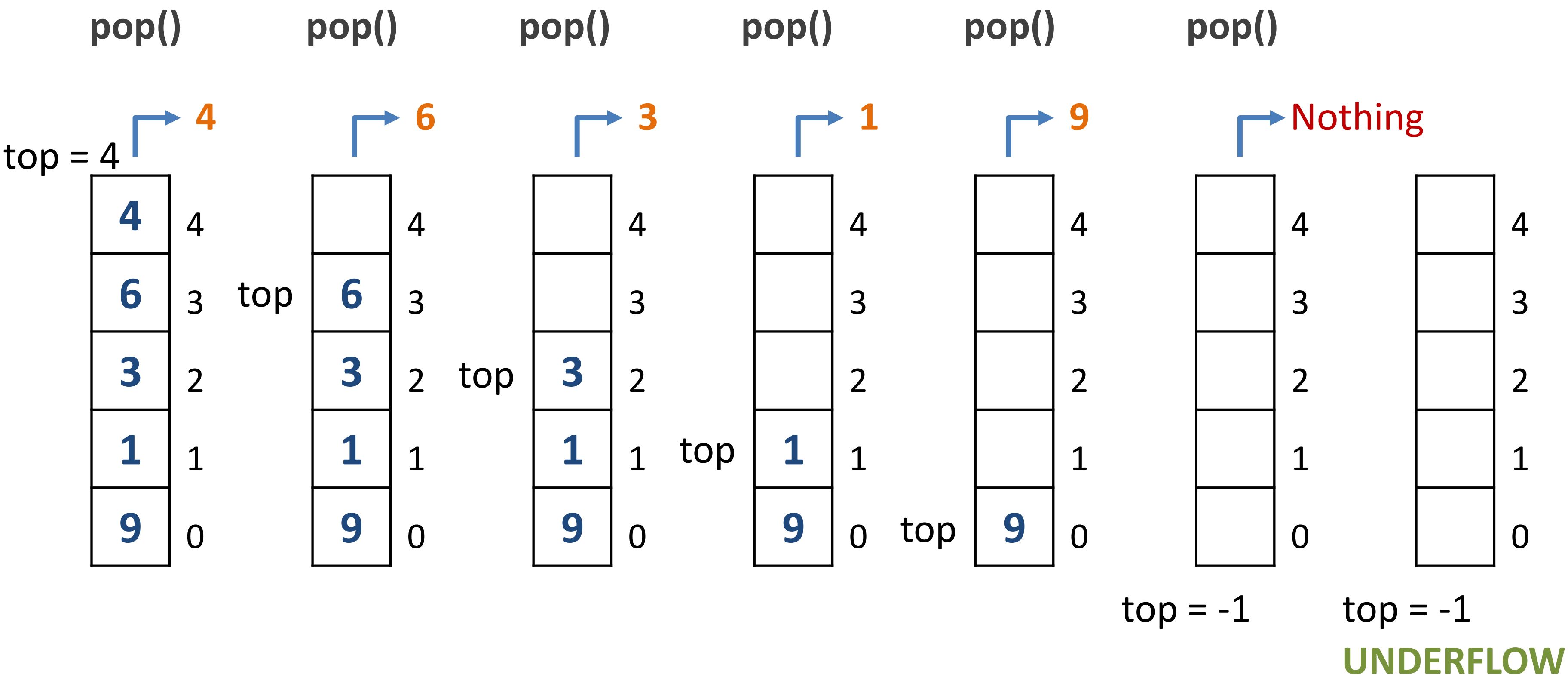
- Two primary operations:
  - **push()** – Pushing (storing) an element on the stack.
  - **pop()** – Removing (accessing) an element from the stack.
- Other operations for effective functionality:
  - **peek()** – Get the top data element of the stack, without removing it.
  - **isFull()** – Check if stack is full. ***OVERFLOW***
  - **isEmpty()** – Check if stack is empty. ***UNDERFLOW***

Stack – Push



The stack is full, no more elements can be added. **OVERFLOW**

Stack – Pop



The stack is empty, no element can be removed. **UNDERFLOW**

Stack as an ADT

- A stack is an ordered list of elements of same data type.
- Elements are always inserted and deleted at one end.
- Following are its basic operations:
  - ***S = init()*** – Initialize an empty stack.
  - ***isEmpty(S)*** – Returns "true" if and only if the stack S is empty, i.e., contains no elements.

Stack as an ADT

- ***isFull(S)*** – Returns "true" if and only if the stack S has a bounded size and holds the maximum number of elements it can.
- ***top(S)*** – Returns the element at the top of the stack S, or error if the stack is empty.
- ***S = push(S,x)*** – Push an element x at the top of the stack S.
- ***S = pop(S)*** – Pop an element from the top of the stack S.
- ***print(S)*** – Prints the elements of the stack S from top to bottom.

Implementation

- Using static arrays
  - Realizes stacks of a maximum possible size.
  - Top is taken as the maximum index of an element in the array.
- Using dynamic linked lists
  - Choose beginning of the list as the top of the stack.

Using Static Arrays

Algorithm for Push

- Let,
  - STACK[SIZE] is a one dimensional array that will hold the stack elements.
  - TOP is the pointer that points to the top most element of the stack.
  - DATA is the data item to be pushed.

1. If TOP == SIZE – 1
2. Display "Overflow condition"
3. Else
4. TOP = TOP + 1
5. STACK [TOP] = DATA

Algorithm for Pop

- Let,
  - STACK[SIZE] is a one dimensional array that will hold the stack elements.
  - TOP is the pointer that points to the top most element of the stack.
  - DATA is the element popped from the top of the stack.

1. If TOP < 0 (or TOP == -1)
2. Display "Underflow condition."
3. Else
4. DATA = STACK[TOP]
5. TOP = TOP – 1
6. Return DATA

Static Array Implementation

```
#define MAXLEN 100

typedef struct
{   int element[MAXLEN];
    int top;    } stack;

stack init ()
{   stack S;
    S.top = -1;
    return S; }

int isEmpty ( stack S )
{    return (S.top == -1); }

int isFull ( stack S )
{    return (S.top == MAXLEN - 1);
}

int top ( stack S )
{   if (isEmpty(S))
        printf("Empty stack\n");
    else
        return S.element[S.top]; }
```

Contd…

```
stack push ( stack S , int x )
{    if (isFull(S))
          printf("OVERFLOW\n");
     else
     {    ++S.top;
          S.element[S.top] = x;
     }
     return S; }
```

```
stack pop ( stack S )
{    if (isEmpty(S))
          printf("UNDERFLOW\n");
     else
     {    --S.top;    }
          return S; }
```

Contd...

```
void print ( stack S )
{       int i;
        for (i = S.top; i >= 0; --i)
            printf("%d",S.element[i]); }

int main ()
{       stack S;
        S = init();
        S = push(S,10);
        S = push(S,45);
        S = push(S,1);
        S = push(S,50);
        printf("Current stack : ");
        print(S);
        printf(" with top = %d.\n", top(S));
        S = pop(S);
        S = pop(S);
        printf("Current stack : ");
        print(S);
        printf(" with top = %d.\n", top(S));
        return 0;       }
```

Using Dynamic Linked Lists

Algorithm for Push

- Let,
  - TOP is the pointer that points to the top most element of the stack.
  - DATA is the data item to be pushed.

1. Create a node pointer (newNode).
2. newNode[data] = DATA.
3. newNode[next] = top.
4. top = newNode.

Algorithm for Pop

- Let,
  - TOP is the pointer that points to the top most element of the stack.
  - temp points to the element popped from the top of the stack.

1. If (TOP == NULL)
2.   Print [Underflow condition].
3. Else
4.   initialize a node pointer (temp) with TOP.
5.   TOP = TOP[next]
6.   Release the memory location pointed by temp.
7. end if

Dynamic Linked List Implementation

```
struct Node
{
  int data;
  struct Node *next;
}*top;

void init()
{ top = NULL;}
```

```
void push(int value)
{
  struct Node *newNode;
  newNode = (struct
     Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  if(top == NULL)
    newNode->next = NULL;
  else
    newNode->next = top;
  top = newNode;
}
```

Contd...

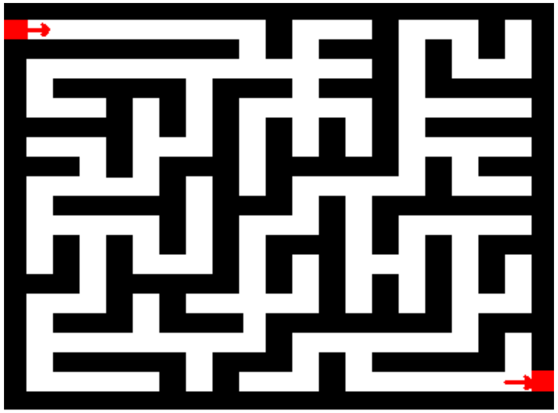
```
void pop()
{
   if(top == NULL)
      printf("\nUnderflow\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
```

Contd...

```
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
        printf("%d--->",temp->data);
        temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

Applications

- Reverse a word.
  - Push a word letter by letter and then pop letters from the stack.
- "UNDO" mechanism in text editors.
- Backtracking.
  - Game playing, finding paths, exhaustive searching.
- Parsing.
- Recursive function calls.
- Calling a function.
- Expression Evaluation.
- Expression Conversion.



Expression Representation

- Infix – Operator is in-between the operands.
- Prefix – Operator is before the operands. Also known as polish notation.
- Postfix – Operator is after the operands. Also known as suffix or reverse polish notation.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *

Example (Infix to Postfix)

- 5 + 3 * 2
  - 5 3 2 * +
- 3 + 4 * 5 / 6
  - 3 4 5 * 6 / +
- (300 + 23) * (43 – 21) / (84 + 7)
  - 300 23 + 43 21 - * 84 7 + /
- (4 + 8) * (6 – 5) / ((3 – 2) * (2 + 2))
  - 4 8 + 6 5 - * 3 2 – 2 2 + * /

Infix to Postfix Conversion Algorithm

- Let Q be any infix expression to be converted in to a postfix expression P.

1. Push left parenthesis onto STACK and add right parenthesis at the end of Q.
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered add it to P.
4. If a left parenthesis is encountered push it onto the STACK.
5. If an operator is encountered, then
   i. Repeatedly pop from STACK and add to P each operator which has same precedence as or higher precedence than the operator encountered.
   ii. Push the encountered operator onto the STACK.
6. If a right parenthesis is encountered, then
   i. Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered.
   ii. Remove the left parenthesis; do not add it to P.
7. Exit

Input	Stack	Output
A + (B * (C – D) / E)		
A + (B * (C – D) / E))	(	
+ (B * (C – D) / E))	(	A
(B * (C – D) / E))	(+	A
B * (C – D) / E))	(+ (	A
* (C – D) / E))	(+ (	A B
(C – D) / E))	(+ (*	A B
C – D) / E))	(+ (* (	A B

Input	Stack	Output
– D) / E))	(+ (* (	A B C
D) / E))	(+ (* (–	A B C
) / E))	(+ (* (–	A B C D
/ E))	(+ (*	A B C D –
E))	(+ (/	A B C D – *
))	(+ (/	A B C D – * E
)	(+	A B C D – * E /
		A B C D – * E / +

Infix to Prefix Conversion Algorithm

- Reverse the given expression.
- Apply algorithm for infix to postfix conversion.
- Reverse the expression.

Example

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

1. Reverse the infix expression.

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

2. Make Every '(' as ')' and every ')' as '('

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

3. Convert expression to postfix…

Input	Stack	Output
5 ^ E + D * (C ^ B + A)		
5 ^ E + D * (C ^ B + A))	(	
^ E + D * (C ^ B + A))	(	5
E + D * (C ^ B + A))	(^	5
D * (C ^ B + A))	(+	5 E ^
* (C ^ B + A))	(+	5 E ^ D
(C ^ B + A))	(+ *	5 E ^ D
C ^ B + A))	(+ * (	5 E ^ D
^ B + A))	(+ * (	5 E ^ D C
B + A))	(+ * (^	5 E ^ D C
+ A))	(+ * (^	5 E ^ D C B

Input	Stack	Output
A))	(+ * (+	5 E ^ D C B ^
))	(+ * (+	5 E ^ D C B ^ A
)	(+ *	5 E ^ D C B ^ A +
		5 E ^ D C B ^ A + * +

4. Reverse the expression.

+ * + A ^ B C D ^ E 5

Example (Infix to Prefix)

- 5 + 3 * 2
    - + 5 * 3 2
- 3 + 4 * 5 / 6
    - + 3 * 4 / 5 6
- (300 + 23) * (43 – 21) / (84 + 7)
    - * + 300 23 / – 43 21 + 84 7
- (4 + 8) * (6 – 5) / ((3 – 2) * (2 + 2))
    - * + 4 8 / – 6 5 * – 3 2 + 2 2

Postfix Evaluation Algorithm

1. Initialize empty stack
2. For every token in the postfix expression (scanned from left to right):
   a. If the token is an operand (number), push it on the stack
   b. Otherwise, if the token is an operator (or function):
      i. Check if the stack contains the sufficient number of values (usually two) for given operator
      ii. If there are not enough values, finish the algorithm with an error
      iii. Pop the appropriate number of values from the stack
      iv. Evaluate the operator using the popped values and push the single result on the stack
3. If the stack contains only one value, return it as a final result of the calculation
4. Otherwise, finish the algorithm with an error

Example

- Evaluate 2 3 4 + * 6 –
- 2 * (3 + 4) - 6 = 8.

	Input token	Operation	Stack contents (top on the right)	Details
2 3 4 + * 6 –	2	Push on the stack	2	
3 4 + * 6 –	3	Push on the stack	2, 3	
4 + * 6 –	4	Push on the stack	2, 3, 4	
+ * 6 –	+	Add	2, 7	Pop two values: 3 and 4 and push the result 7 on the stack
* 6 –	*	Multiply	14	Pop two values: 2 and 7 and push the result 14 on the stack
6 -	6	Push on the stack	14, 6	
-	-	Subtract	8	Pop two values: 14 and 6 and push the result 8 on the stack
	(End of tokens)	(Return the result)	8	Pop the only value 8 and return it

Solve

- Consider the following Stack , where STK is allocated N=6 memory cells:

  STK: PPP,QQQ, RRR,SSS,TTT

  Describe the stack as the following operations take place:

  1. PUSH(STK, UUU)
  2. POP(STK)
  3. PUSH(STK, VVV)
  4. PUSH(STK, WWW)
  5. POP(STK)

Solve

- Convert to equivalent postfix expression:

1. (A-B)*(D/E)
2. A*(B+D)/E-F*(G+H/K)
3. ((p+q)*s^(t-u))

- Evaluate the expression 5 6 2+*12 4 /- in tabular form showing stack after every step.

Postfix to Infix Conversion Algorithm

1. Accept a postfix string from the user (say P).
2. Start scanning the string one character at a time.
3. If it is an operand, push it in stack.
4. If it is an operator, pop opnd1, opnd2 and concatenate them in the order "(opnd2, optr, opnd1)".
5. Push the result in the stack.
6. Repeat these steps until input postfix string P ends.
7. Pop the remaining element of the stack, which is the required Infix notation equivalent to a given Postfix notation.

Solve: abc-+de-fg-h+/*

Expression	Input token	Stack contents as strings (top on the right)
abc-+de-fg-h+/*		
bc-+de-fg-h+/*	a	a
c-+de-fg-h+/*	b	a, b
-+de-fg-h+/*	c	a, b, c
+de-fg-h+/*	–	a, (b – c)
de-fg-h+/*	+	(a + (b – c))
e-fg-h+/*	d	(a + (b – c)), d
-fg-h+/*	e	(a + (b – c)), d, e
fg-h+/*	–	(a + (b – c)), (d – e)
g-h+/*	f	(a + (b – c)), (d – e), f
-h+/*	g	(a + (b – c)), (d – e), f, g
h+/*	–	(a + (b – c)), (d – e), (f – g)
+/*	h	(a + (b – c)), (d – e), (f – g), h
/*	+	(a + (b – c)), (d – e), ((f – g) + h)
*	/	(a + (b – c)), ((d – e) / ((f – g) + h))
	*	((a + (b – c)) * ((d – e) / ((f – g) + h)))

Prefix to Infix Conversion Algorithm

1. Accept a prefix string from the user (say P).
2. Start scanning the string from right one character at a time.
3. If it is an operand, push it in stack.
4. If it is an operator, pop opnd1, opnd2 and concatenate them in the order "(opnd1, optr, opnd2)".
5. Push the result in the stack.
6. Repeat these steps until input prefix string P ends.
7. Pop the remaining element of the stack, which is the required Infix notation equivalent to a given Prefix notation.

Example

- Solve: *+a-bc/-de+-fgh
- We can either start scanning from right or reverse the expression first and then scan from left to right.
- The reversed expression is

h g f – + e d – / c b – a + *

Expression	Input token	Stack contents as strings (top on the right)
hgf–+ed–/cb–a+*		
gf–+ed–/cb–a+*	h	h
f–+ed–/cb–a+*	g	h, g
–+ed–/cb–a+*	f	h, g, f
+ed–/cb–a+*	–	h, (f – g)
ed–/cb–a+*	+	((f – g) + h)
d–/cb–a+*	e	((f – g) + h), e
–/cb–a+*	d	((f – g) + h), e, d
/cb–a+*	–	((f – g) + h), (d – e)
/cb–a+*	/	((d – e) / ((f – g) + h))
b–a+*	c	((d – e) / ((f – g) + h)), c
–a+*	b	((d – e) / ((f – g) + h)), c, b
a+*	–	((d – e) / ((f – g) + h)), (b – c)
+*	a	((d – e) / ((f – g) + h)), (b – c), a
*	+	((d – e) / ((f – g) + h)), (a + (b – c))
	*	((a + (b – c)) * ((d – e) / ((f – g) + h)))