Stack & its Applications

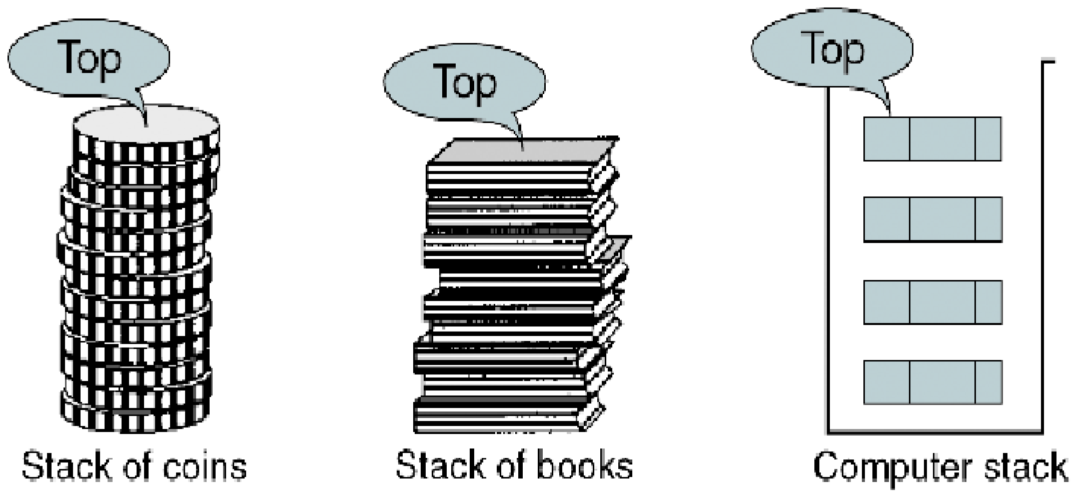
Dr. Pradeep Kumar Mallick
Associate Professor-II

What is a stack?

- A stack is a linear data structure that stores a set of homogeneous elements in a particular order
- Stack principle: LAST IN FIRST OUT (LIFO)
- Means: the last element inserted is the first one to be removed
- Example:

- Elements are removed in the reverse order in which they were inserted

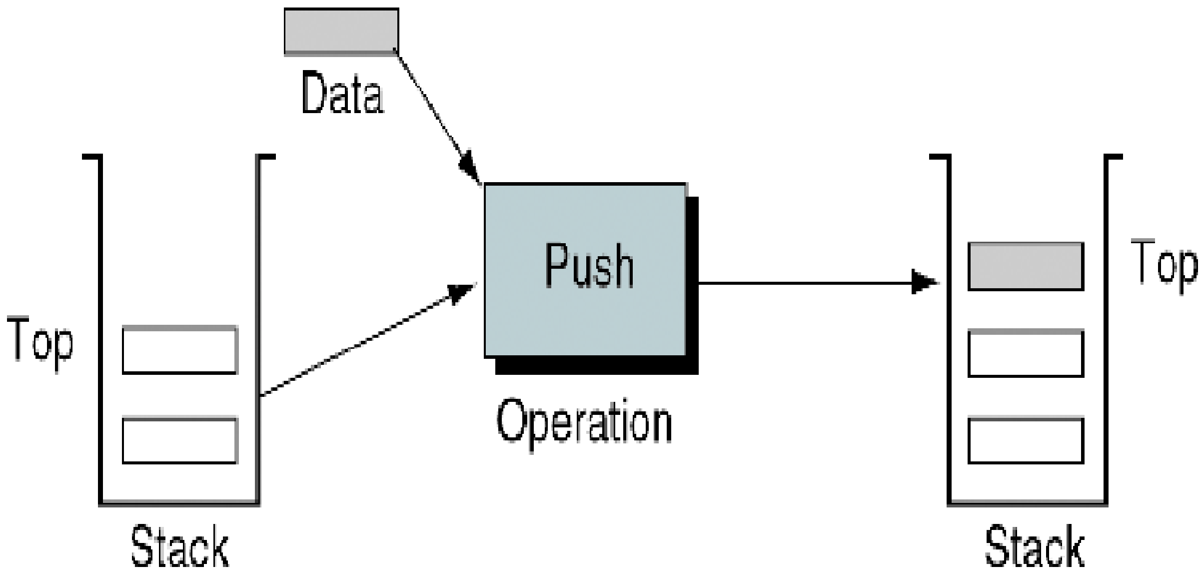
Examples of Stack



Stack

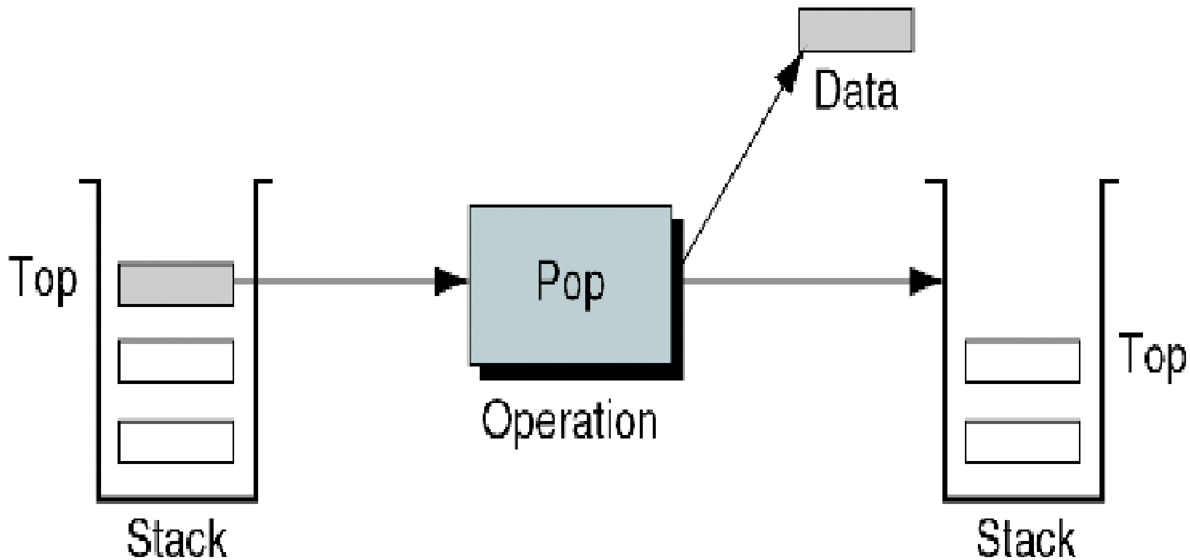
Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).

Operations on Stack



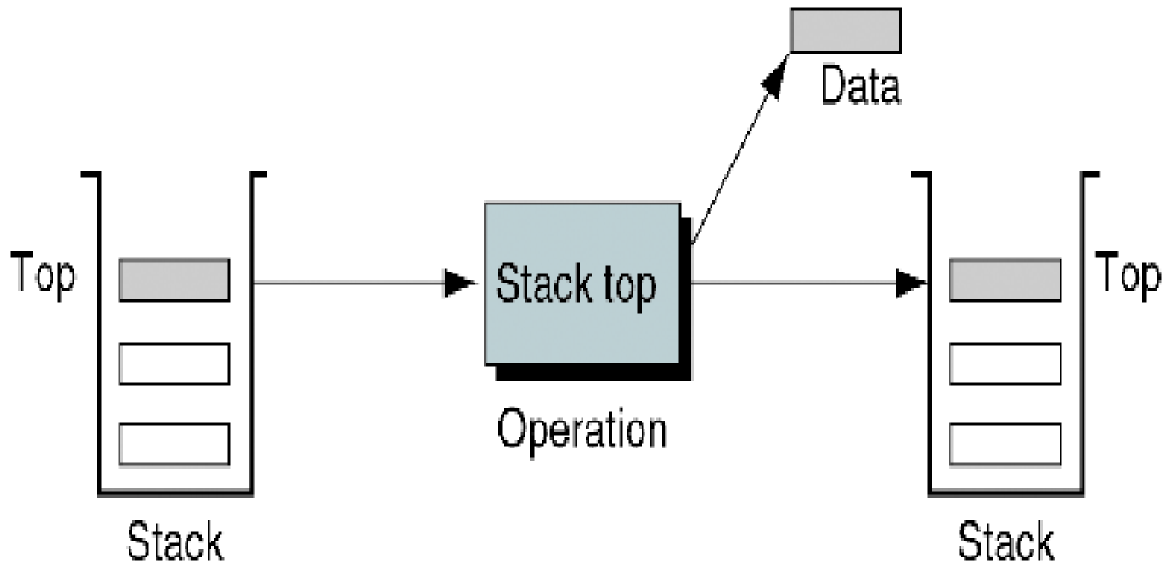
Push Stack Operation

Operations on Stack



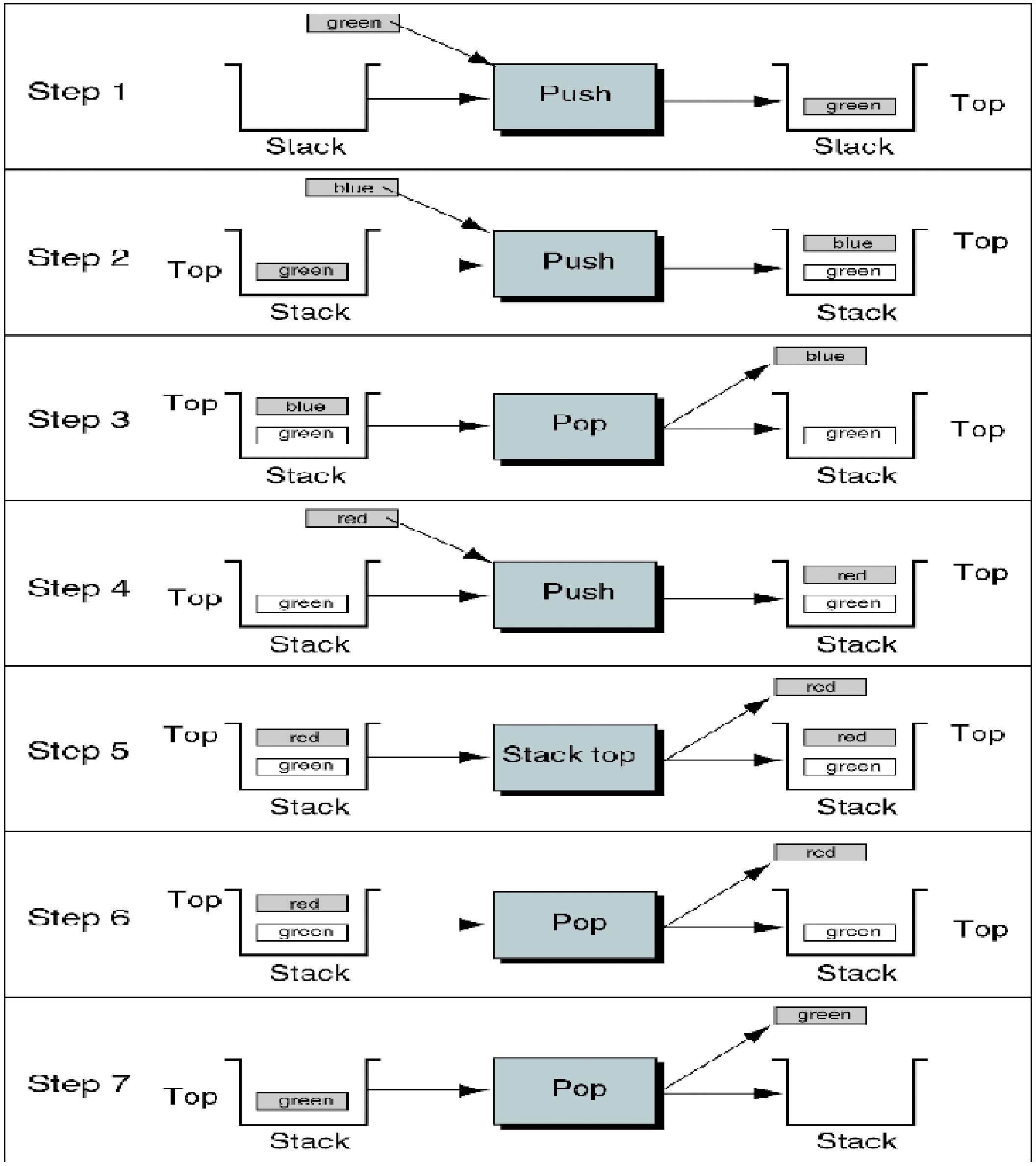
Pop Stack Operation

Operations on Stack



Stack Top Operation

Examples of Stack



Stack Example

Stack Applications

- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack
  - Evaluating expressions

Stack Implementation using Array

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at 0th position of array
- last element in the array is at the *top*
- Increment *top* when one element is pushed, decrement after pop

CreateS, isEmpty, isFull

Stack createS(stack_size) =

#define STACK_SIZE 100   /* maximum stack size */

element stack[STACK_SIZE];
int top = -1;

Boolean isEmpty(Stack) = top= -1;

Boolean isFull(Stack) = top = STACK_SIZE-1;

Push

Push(Stack,MAXSTK,TOP,item)

1.[Check for stack overflow]

    if(Top>=MAXSTK-1 then

        print stack over flow and exit

2.Set TOP:= TOP+1

3. [Perform insertion]

        Stack[TOP]=item

4.Exit

Pop

Pop(STACK,TOP, TEMP)

1.[Check for stack underflow]
  if TOP<0 or Top= -1 then
    print stack under flow and exit
  else [Remove Item]
    Set TEMP:= STACK[TOP]
2.[Decrement Stack TOP]
    Set TOP:= TOP-1
3. Return the deleted item from Stack
4.Exit

Implement stacks in an array

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
```

Implement stacks in an array

```
do
 {
   printf("\n Enter the Choice:");
   scanf("%d",&choice);
   switch(choice)
   {
     case 1:
       push();
       break;
     case 2:
       pop();
       break;
case 3:
     display();
      break;
 case 4:
       printf("\n\t EXIT POINT ");
       break;
  default:
 printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
     }
   } while(choice!=4);
   return 0;
}
```

Implement of stacks in an array

```
void push()
{
  if(top>=n-1)
  {
    printf("\n\tSTACK is over flow");
    exit(0);
  }
  else
  {
    printf(" Enter a value to be pushed:");
    scanf("%d",&x);
    top++;
    stack[top]=x;
  }}
```

Implement of stacks in an array

```
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
        exit(0);
    }
    else
    {
        printf("\n\t The popped elements is %d", stack[top]);
        top--;
    }
}
```

Implement of stacks in an array

```
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);
      printf("\n Press Next Choice");
   }
   else
         printf("\n The STACK is empty");


}
```

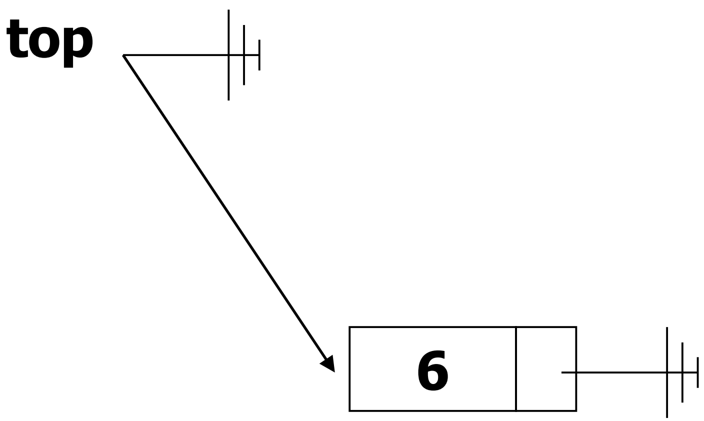
Stack: Linked List Implementation

- Push and pop at the head of the list
  - New nodes should be inserted at the front of the list, so that they become the top of the stack
  - Nodes are removed from the front (top) of the list
- Straight-forward linked list implementation
  - push and pop can be implemented fairly easily, e.g. assuming that head is a reference to the node at the front of the list

Stack: Example

C Code

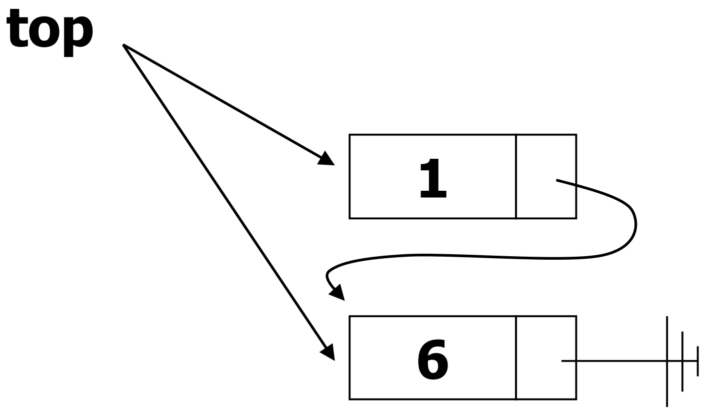
```
Stack s;
s.push(6);
```



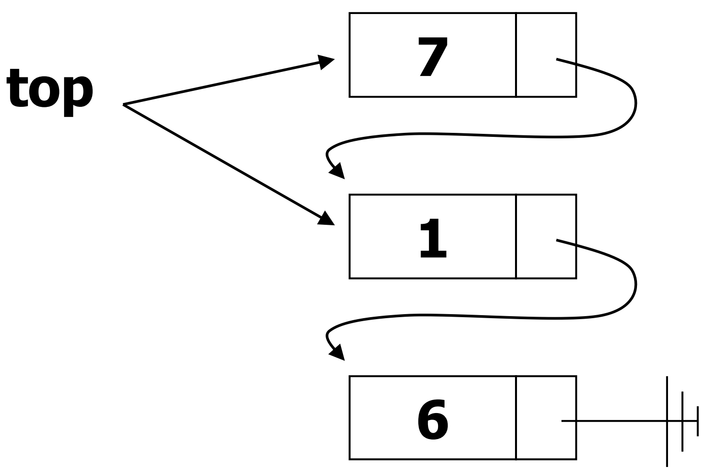
Stack: Example

C Code

```
Stack s;
s.push(6);
s.push(1);
```



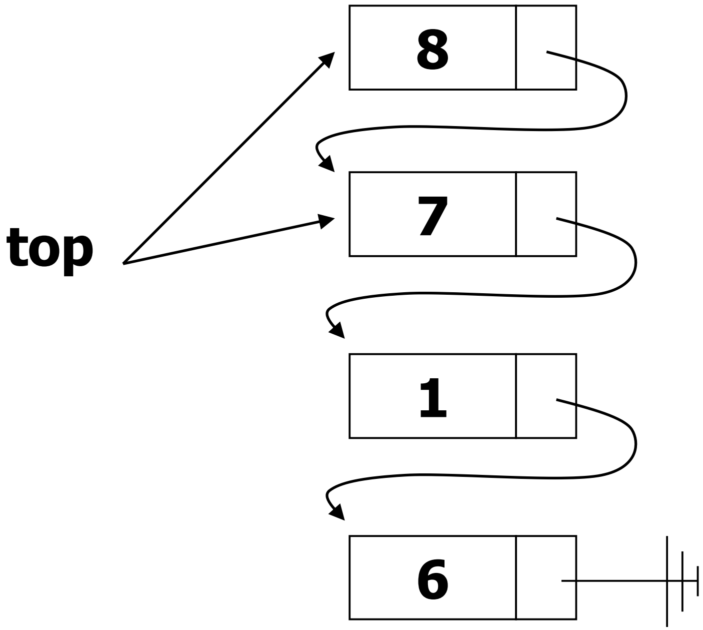
Stack: Example



C Code

```
Stack s;
s.push(6);
s.push(1);
s.push(7);
```

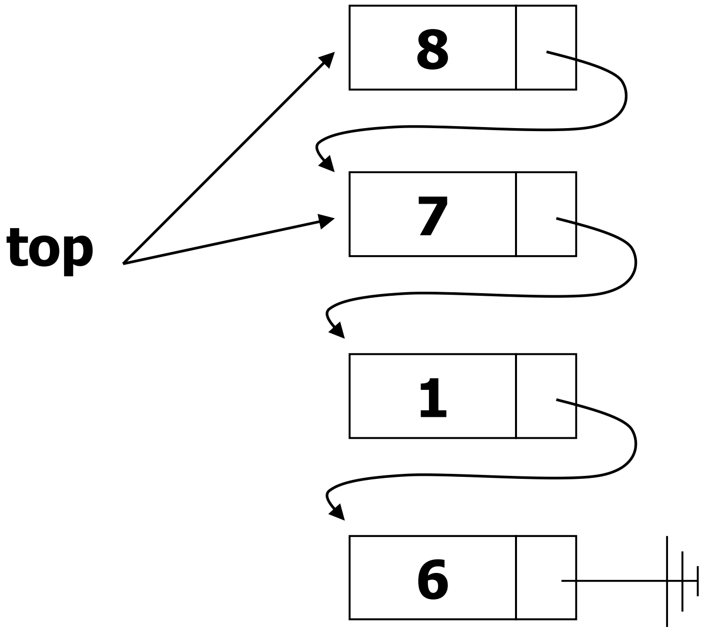
Stack: Example



C Code

```
Stack s;
s.push(6);
s.push(1);
s.push(7);
s.push(8);
```

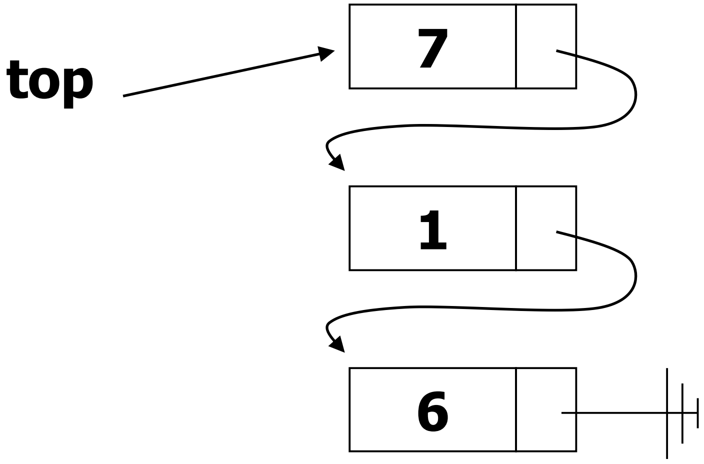
Stack: Example



C Code

```
Stack s;
s.push(6);
s.push(1);
s.push(7);
s.push(8);
s.pop();
```

Stack: Example



C Code

```
Stack s;
s.push(6);
s.push(1);
s.push(7);
s.push(8);
s.pop();
```

Stack Implementation

```
typedef struct stack {
   int data;
   struct stack *next;
} stack;
```

Menu Driven

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct stack
{
      int info;
      struct stack * next;
};
typedef struct stack   stack;
//Prototype Declaration
stack* push (stack*);
stack* pop(stack*);
void display(stack*);
```

```
void main()
{
                int flag=1, choice;
                stack *top=NULL;
                while(flag==1)
                {
                        printf("Press 1 to push node \n");
                        printf("Press 2 to for pop  \n");
                         printf("Press 3 to for display  \n");
                        printf("Enter your Choice:");
                        scanf("%d", &choice);
                        switch(choice)
                        {
                                    case 1:
                                        top= push(top);
                                        break;
                                    case 2:
                                        top=pop(top);
                                    break;
```

Menu Driven

```
case 3:
            display(top);
            break;
        default:
             printf("Wrong Choice");
             break;
      }
    }
}
```

stack* push(stack* top)

```
{
  stack* newnode;
  int newinfo;
printf("Enter the new information to be push on
stack top\n");
  scanf("%d",&newinfo);
  newnode=(stack*)malloc(sizeof(stack));
  newnode->info=newinfo;
  if (top == NULL)
  {
    top=newnode;
    top->next=NULL;
  }
  else
  {
    newnode->next=top;
    top=newnode;
  }
  return(top);
}
```

Menu Driven

```
stack* pop(stack* top)
{
    int item;  stack *temp;
    if( top == NULL)
    {
        printf(" Empty Stack …");
        return(NULL) ;
    }
    else
    {
        temp=top;   top=top->next;
        temp->next=NULL;
        free(temp);
    }
        return(top);
}
```

```
void display(stack *top)
{
    stack *temp;
    temp = top;
    if(top == NULL)
    {
      printf("Empty stack …");
      exit(0);
    }
    else
    {
      printf("Top->");
      while(temp != NULL)
      {
        printf("%d ->", temp->info);
        temp = temp->next;
      }
    }
}
```

Performance and Limitations (Implementation of stack ADT using Array)

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Implement two stacks in an array

- Create a data structure that represents two stacks using only one array.
- Following functions must be supported by twoStacks:
  - push1(int x): pushes x to first stack
  - push2(int x): pushes x to second stack
  - pop1(): pops an element from first stack and return the popped element
  - pop2(): pops an element from second stack and return the popped element

Implement two stacks in an array

Implementation of twoStack should be space efficient.

- Method 1 (Divide the space in two halves)
  - A simple way to implement two stacks is to divide the array in two halves and assign the half space to two each stack, i.e., use arr[0] to arr[n/2-1] for stack1, and arr[n/2] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be n.

- The problem with this method is inefficient use of array space.
- A stack push operation may result in stack overflow even if there is space available in arr[]

Implement two stacks in an array

```
top1 = -1;
top2 = n/2 - 1;
```

```
// Method to push an element x to stack1
void push1(int x) {
  if(top1 == n/2-1) {
    printf("Stack Overflow...");
    return;
   }
   top1++;
   stack[top1] = x;
}
```

```
// Method to push an element x to stack2
void push2(int x) {
  if(top2 == n-1) {
    printf("Stack Overflow...");
    return;
   }
   top2++;
   stack[top2] = x;
}
```

Implement two stacks in an array

```
// Method to pop an element from first stack
int pop1() {
   int x;
   if(top1 == -1)  {
      printf("Stack Underflow...");
      return -9999;
    }
    x = stack[top1];
    top1--;
    return(x);
}
```

```
// Method to pop an element from second stack
int pop2() {
   int x;
   if(top2 == n/2 - 1)  {
      printf("Stack Underflow...");
      return -9999;
    }
    x = stack[top2];
    top2--;
    return(x);
}
```

Implement two stacks in an array

- Method 2: (A space efficient implementation)
- This method efficiently utilizes the available space.
- It doesn't cause an overflow if there is space available in arr[].
- The idea is to start two stacks from two extreme ends of arr[].
  - stack1 starts from starting of the array, the first element in stack1 is pushed at index 0.
  - The stack2 starts from end of the array, the first element in stack2 is pushed at index (n-1).
- Both stacks grow (or shrink) in opposite direction.
- To check for overflow, it needs to check for space between top elements of both stacks.

Implement two stacks in an array

```
top1 = -1;
top2 = n;
```

```
// Method to push an element x to stack1
void push1(int x) {
  if(top1 == top2-1) {
    printf("Stack Overflow...");
    return;
   }
   top1++;
   stack[top1] = x;
}
```

```
// Method to push an element x to stack2
void push2(int x) {
  if(top1 == top2-1) {
    printf("Stack Overflow...");
    return;
   }
   top2--;
   stack[top2] = x;
}
```

Implement two stacks in an array

```
// Method to pop an element from first stack
int pop1() {
   int x;
   if(top1 == -1)  {
      printf("Stack Underflow...");
      return -999;
    }
    x = stack[top1];
    top1--;
    return(x);
}
```

```
// Method to pop an element from second stack
int pop2() {
   int x;
   if(top2 == n)  {
      printf("Stack Underflow...");
      return -999;
    }
    x = stack[top2];
    top2++;
    return(x);
}
```

Reverse a String using Stack

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
char str[MAX];
int top = -1;
int main() {
    char str[MAX];
    int i;
    printf("Input a string: ");
    scanf("%[^\n]", str);
    for(i=0; i<strlen(str); i++) pushChar(str[i]);
    for(i=0; i<strlen(str); i++) str[i]=popChar();
    str[i] = '\0';
    printf("Reversed String is: %s\n", str);
    return 0;
}
```

```
void pushChar(char item) {
    if(top != MAX) {
      top=top+1;
      strr[top]=item;
    }
}

char popChar() {
    int item;
    if(top != -1) {
      item = str[top];
      top=top-1;
      return item;
    }
}
```

Infix, Prefix, & Postfix Expressions

Infix Notation

- Usually the algebraic expressions are written like this: a + b
- This is called infix notation, because the operator ("+") is in between operands in the expression
- A problem is that it needs parentheses or precedence rules to handle more complicated expressions:

For Example :

$$a + b * c = (a + b) * c \ ?$$
$$= a + (b * c) \ ?$$

Infix, Postfix, & Prefix notation

- There is no reason to place the operator somewhere else.
- How ?
  - Infix notation : a + b
  - Prefix notation : + a b
  - Postfix notation: a b +

Other Names

- **Prefix** notation was introduced by the Polish logician **Lukasiewicz**, and is sometimes called "Polish Notation".

- **Postfix** notation is sometimes called "Reverse Polish Notation" or **RPN.**

Why ?

- **Question:** Why would anyone ever want to use anything so "unnatural," when infix seems to work just fine?
- **Answer:** With postfix and prefix notations, parentheses are no longer needed!
- Advantages of postfix:
  - Don't need rules of precedence
  - Don't need rules for right and left associativity
  - Don't need parentheses to override the above rules

Example

infix	***postfix***	***prefix***
(a + b) * c	a b + c *	* + a b c
a + (b * c)	a b c * +	+ a * b c

Infix form : <*identifier*> <*operator*> <*identifier*>

Postfix form : <*identifier*> <*identifier*> <*operator*>

Prefix form : <*operator*> <*identifier*> <*identifier*>

Conclusion

- Infix is the only notation that requires parentheses in order to change the order in which the operations are done.

Infix to Postfix conversion (Intuitive Algorithm)

- An Infix to Postfix manual conversion algorithm is:
  1. Completely parenthesize the infix expression according to order of priority you want.
  2. Move each operator to its corresponding right parenthesis.
  3. Remove all parentheses.

- Examples:

3 + 4 * 5 $\longrightarrow$ (3 + (4 * 5)) $\longrightarrow$ 3 4 5 * +

a / b ^ c – d * e – a * c ^ 3 ^ 4 a b c ^ / d e * a c 3 4 ^ ^ * - -

((a / (b ^ c)) – ((d * e) – (a * (c ^ (3 ^ 4)))))

Infix to Postfix Conversation

ALGORITHM: INFIX_TO_POSTFIX (I, P)

Input: I is an arithmetic expression written in infix notation.

Output: This algorithm converts the infix expression to its equivalent postfix expression P.

Step 1: Add ')' to the end of the infix expression I and push '('on to the STACK.

Step 2: Scan I from left to right and repeat Step 3 to Step 6 for each element of I until the STACK is empty.

Step 3: If an operand is encountered, add it to P.

Step 4: If a left parentheses '('is encountered, push it on to the STACK.

Infix to Postfix Conversation

Step 5: If an operator **OP** is encountered, then

(a)Repeatedly pop from the STACK and add each operator (on the top of the STACK) to the Postfix expression **‘P’** which has same precedence as or higher precedence than **OP**.

(b)Push the operator **OP** on to the STACK.

[End of if]

Step 6: If a right parentheses ‘)’ is encountered, then

(a) Repeatedly pop from the STACK and add each operator (on the top of the STACK) to the Postfix expression ‘P’ until a left parentheses ‘(‘ is encountered.

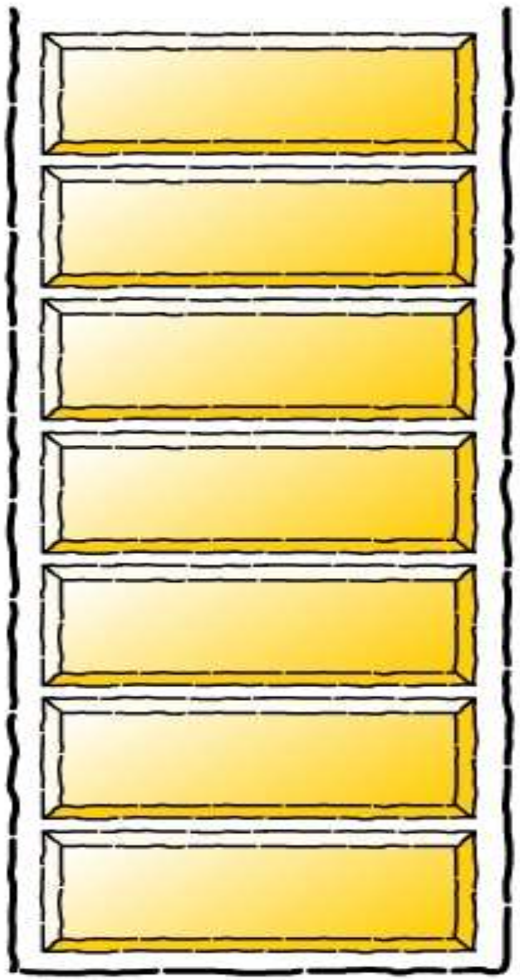
(b) Remove the left parentheses ‘(‘. [Don’t add it to the postfix expression P.]

[End of if]

[End of Step 2 loop]

Step 7: Exit

Infix to Postfix Conversion

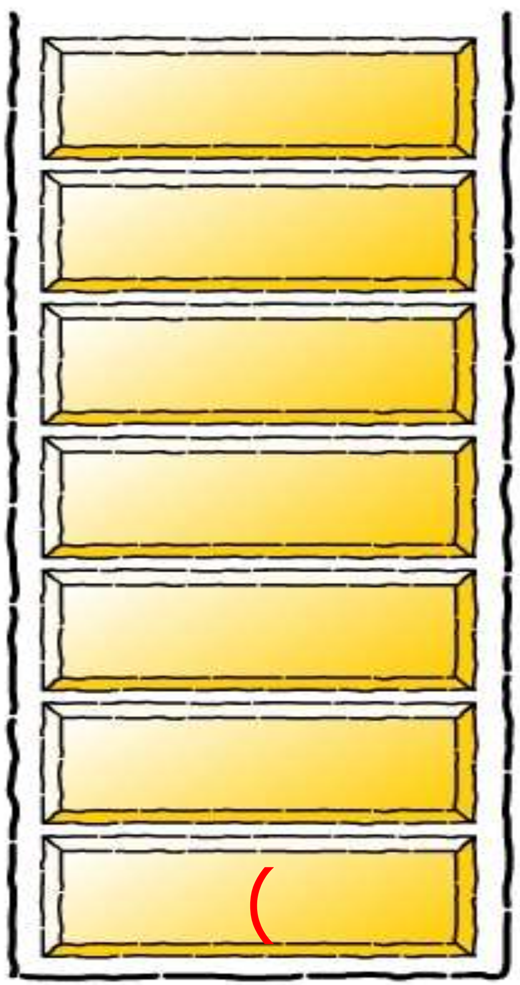


Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

Infix to Postfix Conversion

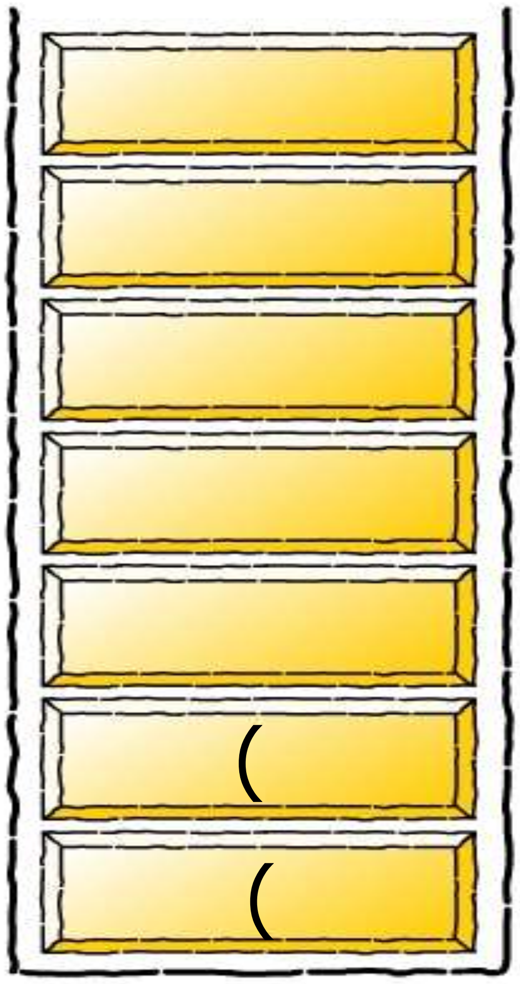


Infix Expression

(a + b - c) * d – (e + f))

Postfix Expression

Infix to Postfix Conversion

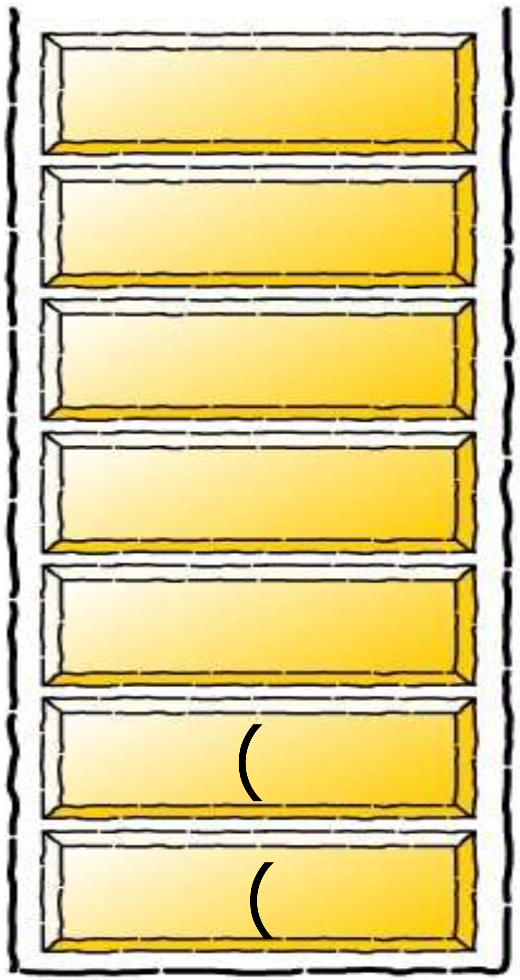


Infix Expression

a + b - c) * d – (e + f))

Postfix Expression

Infix to Postfix Conversion



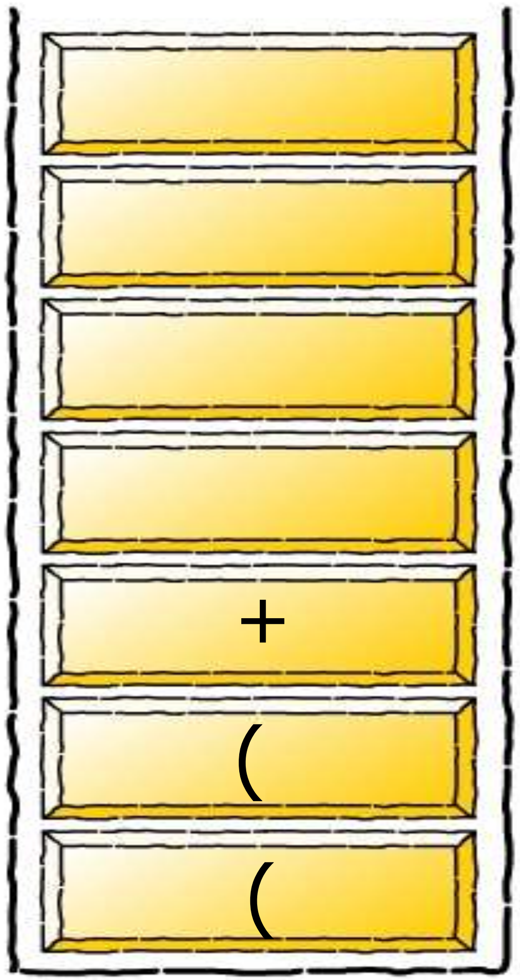
Infix Expression

+ b - c) * d – (e + f))

Postfix Expression

a

Infix to Postfix Conversion



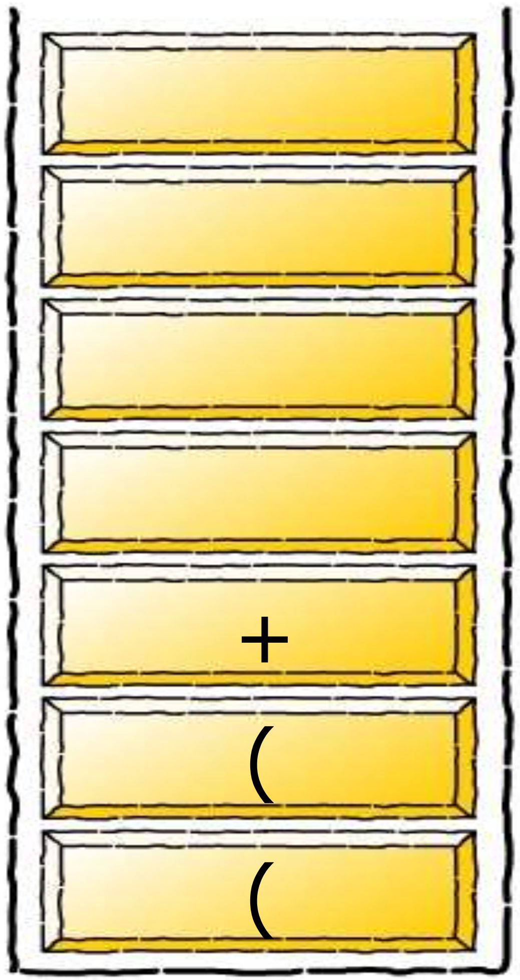
Infix Expression

b - c) * d – (e + f))

Postfix Expression

a

Infix to Postfix Conversion



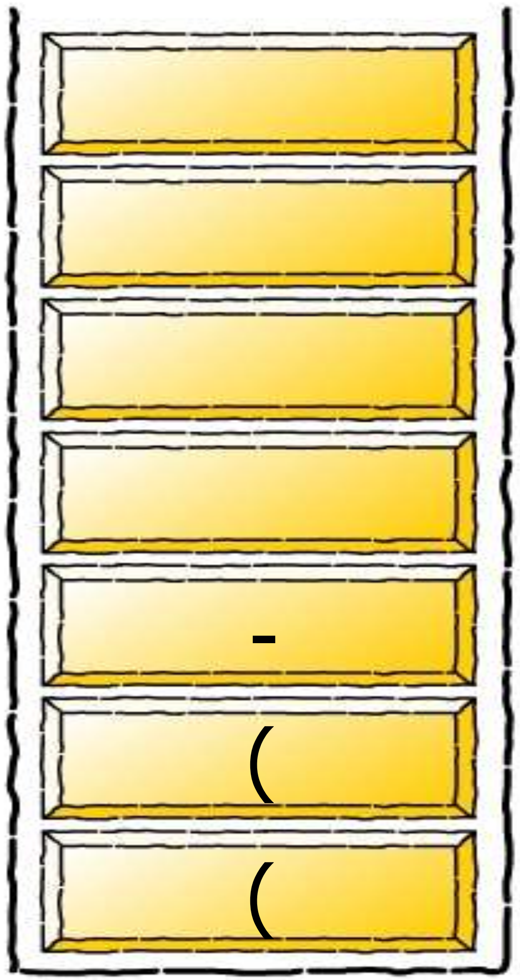
Infix Expression

- c) * d – (e + f))

Postfix Expression

a b

Infix to Postfix Conversion



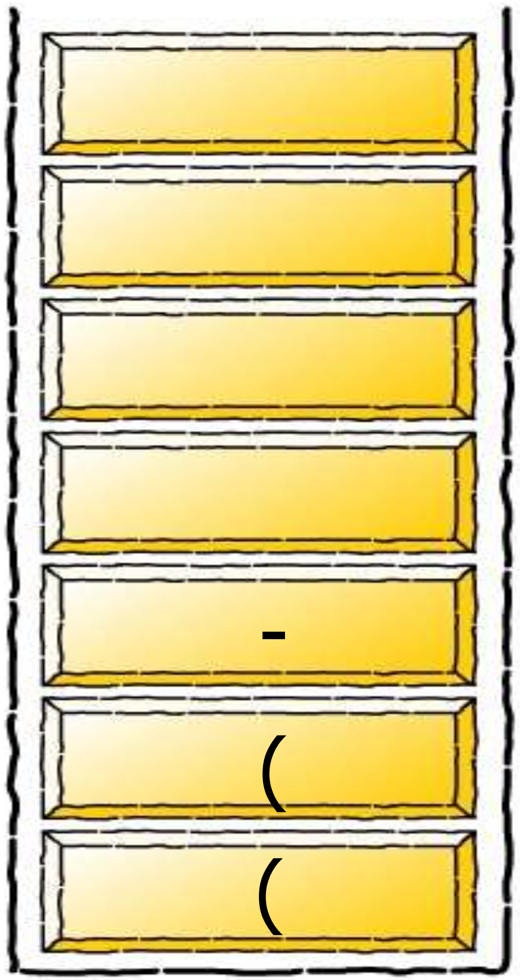
Infix Expression

c) * d – (e + f))

Postfix Expression

a b +

Infix to Postfix Conversion



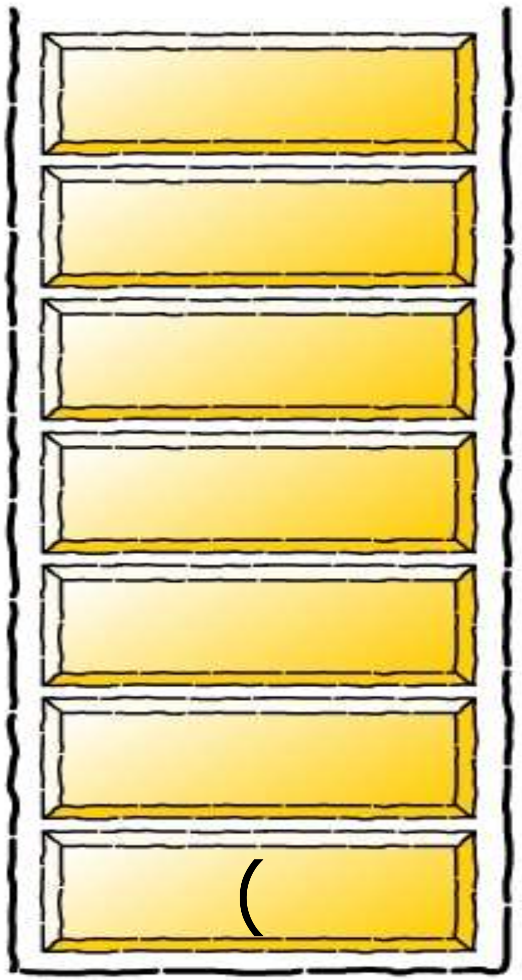
Infix Expression

) * d – (e + f))

Postfix Expression

a b + c

Infix to Postfix Conversion



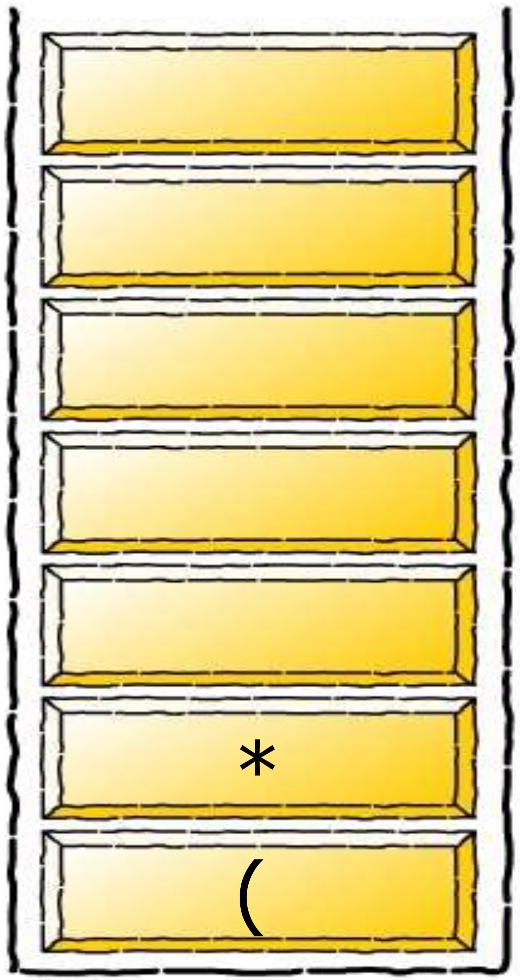
Infix Expression

* d – (e + f))

Postfix Expression

a b + c -

Infix to Postfix Conversion



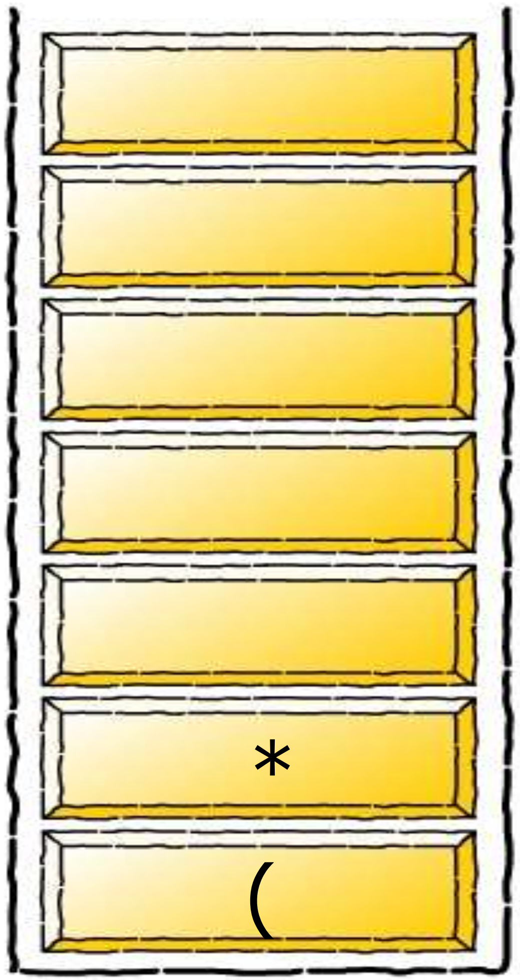
Infix Expression

d – (e + f))

Postfix Expression

a b + c -

Infix to Postfix Conversion



Stack

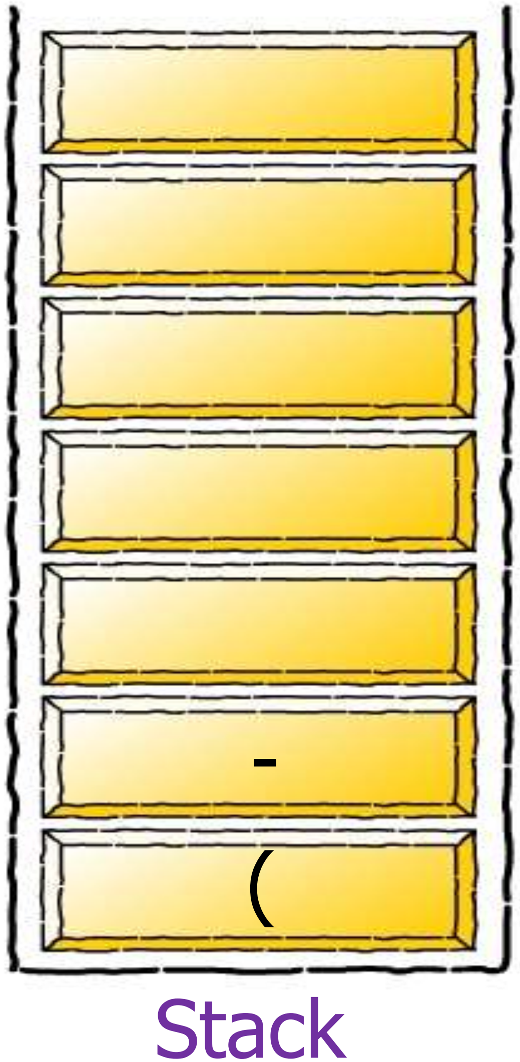
Infix Expression

− (e + f))

Postfix Expression

a b + c − d

Infix to Postfix Conversion



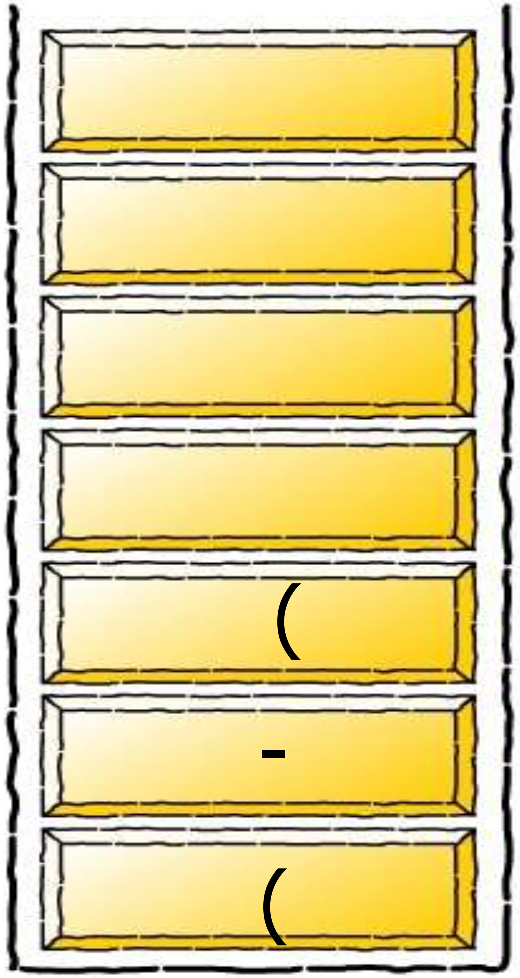
Infix Expression

(e + f))

Postfix Expression

a b + c – d *

Infix to Postfix Conversion



Stack

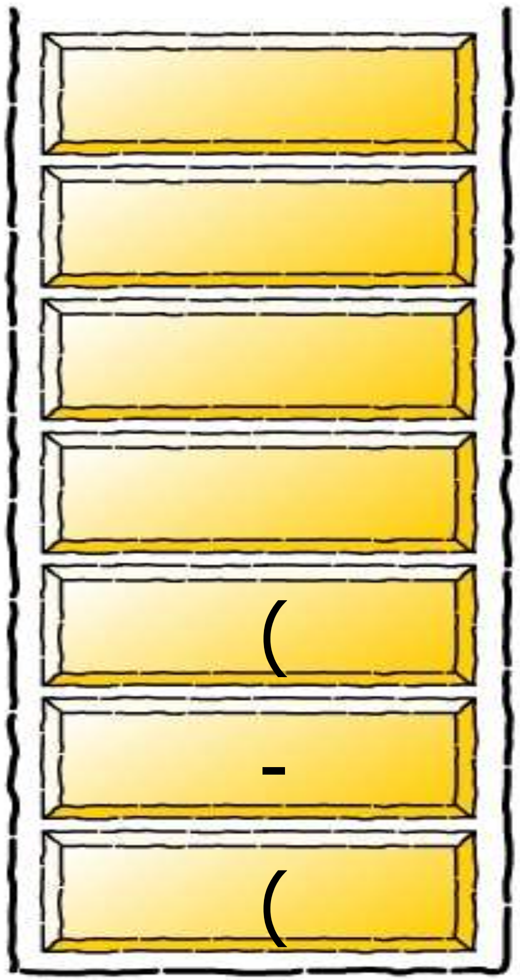
Infix Expression

e + f))

Postfix Expression

a b + c – d *

Infix to Postfix Conversion



Stack

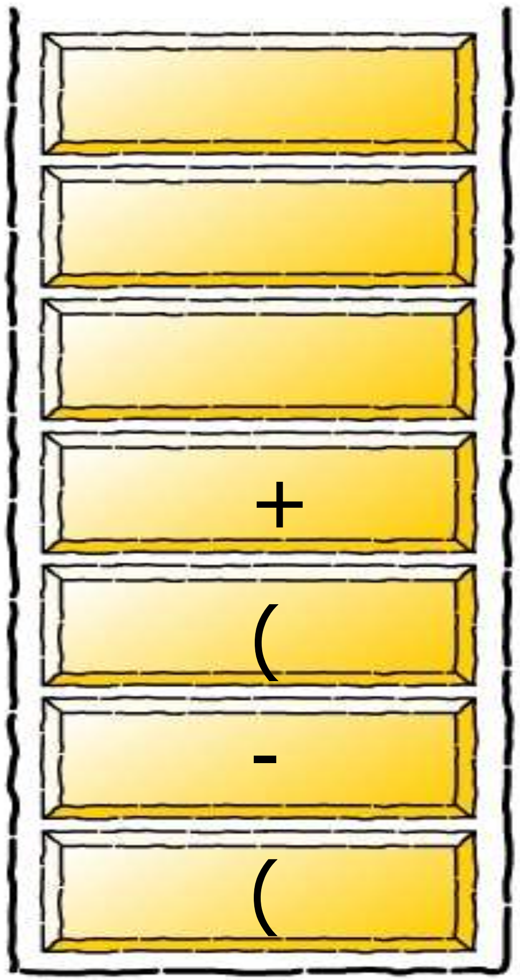
Infix Expression

+ f))

Postfix Expression

a b + c – d * e

Infix to Postfix Conversion



Stack

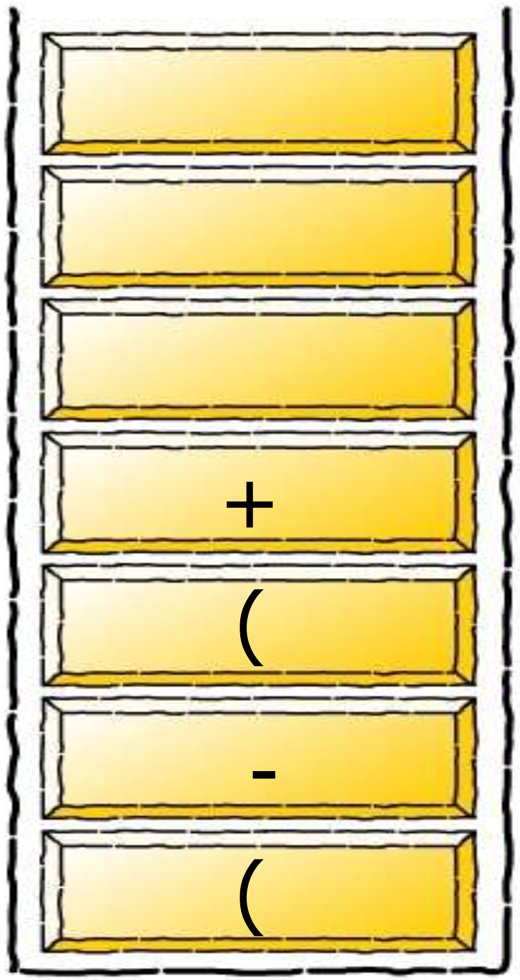
Infix Expression

f))

Postfix Expression

a b + c – d * e

Infix to Postfix Conversion



Stack

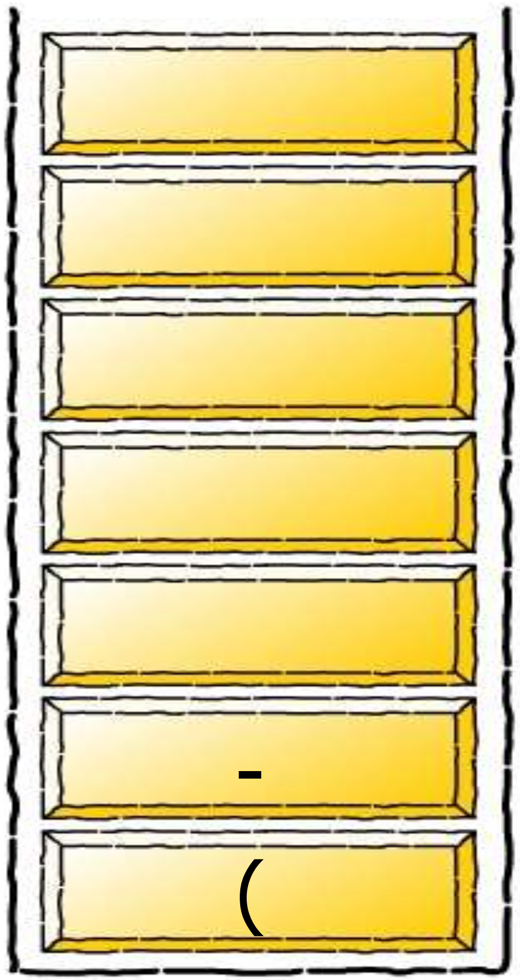
Infix Expression

))

Postfix Expression

a b + c – d * e f

Infix to Postfix Conversion



Stack

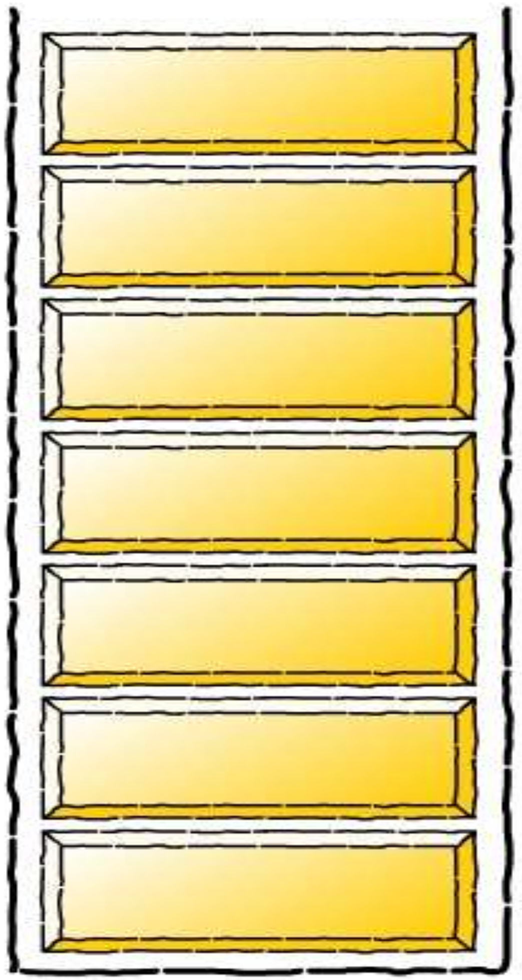
Infix Expression

)

Postfix Expression

a b + c – d * e f +

Infix to Postfix Conversion



Stack

Infix Expression

Postfix Expression

a b + c – d * e f + -

Convert 2*3/(2-1)+5*3 into Postfix form

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Postfix Expression is 23*21-/53*+

Exmpale

(a+(b+c)*d/e)+(f+g)

POSTFIX EVALUATION

POSTFIX EVALUATION(P)

[Suppose P is an arithmetic Expression written in postfix expression]

1. Add a right parenthesis at the end of the P postfix expression.
2. Scan **P** from left to right until a right parenthesis occur.

   Repeat step 3 to 4
3. If an operand comes then add it to the stack. If operator comes then or encounter
   1. POP top two elements from the stack ,where A is the first top element and B is the second top element.
   2. Evaluate A and B.
   3. Store the result at the top of the stack again.
4. Exit

Exmpale

P:12,7,3,-,/,2,1,5,+,*,+

12,7,3,-,/,2,1,5,+,*,+)

Symbol	Stack
12	12
7	12,7
3	12,7,3
-	12,4 (7-3)
/	3 (12/4)
2	3,2
1	3,2,1
5	3,2,1,5
+	3,2,6 (1+5)
*	3,12 (2*6)
+	15 (3+15)

Precedence of Operators

Token	Operator	Precedence	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
! - - + & * sizeof	logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

Precedence of Operators

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right

Precedence of Operators

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^=	assignment	2	right-to-left
,	comma	1	left-to-right

Postfix conversion

user compiler

Infix	Postfix
2+3*4 a*b+5 (1+2)*7 a*b/c (a/(b-c+d))*(e-a)*c a/b-c+d*e-a*c	234*+ ab*5+ 12+7* ab*c/ abc-d+/ea-*c* ab/c-de*+ac*-

Postfix: no parentheses, no precedence

Infix to Prefix Conversion

Q is an arithmetic expression written in infix notation. This algorithm **InfixToPrefix(Q, P)** finds the equivalent postfix notation where P is the equivalent prefix notation –

1. Start
2. Validate the Infix expression for correctness
   (a) Operator is between the operands [Binary operators are considered only]
   (b) Brackets '(' and ')' are properly matched.
3. Reverse the infix expression Q
4. **CALL** InfixToPostfix (Q, P)
5. Reverse the expression P
6. Stop

Example: Infix notation: A+B*C

Step 1 – Infix expression is correct

Step 2 – Reversing the expression resulting to C*B+A

Step 3 – Postfix expression produced in step 2 is CB*A+

Step 4 – Reversing the postfix expression produced in step 3 is **+ A * B C**

Prefix or P: + A * B C

Algorithm for Conversion of infix to prefix expression

ALGORITHM: INFIX_TO_PREFIX (I, P)

Input: I is an arithmetic expression written in infix notation.

Output: This algorithm converts the infix expression to its equivalent prefix expression P.

Step 1: Add '(' to the beginning of the infix expression I and push ')' on to the STACK.

Step 2: Scan I from right to left and repeat Step 4 to Step 6 for each element of I until the STACK is empty.

Step 3: If an operand is encountered, PUSH it on to the Output Stack(Stack2).

Step 4: If a right parentheses ')' is encountered, push it on to the STACK.

Algorithm for Conversion of infix to prefix expression

Step 5: If an operator OP is encountered, then

(a)Repeatedly pop from the STACK and add each operator (on the top of the STACK) to the Output Stack(Stack2) which has higher precedence than OP.

(b)Push the operator OP on to the STACK.

[End of if]

Step 6: If a left parentheses '(' is encountered, then

(a) Repeatedly pop from the STACK and add each operator (on the top of the STACK) to the Output Stack(Stack2) until a right parentheses ')' is encountered.

(b) Remove the right parentheses ')'. [Don't add it to the Output Stack(Stack2).]

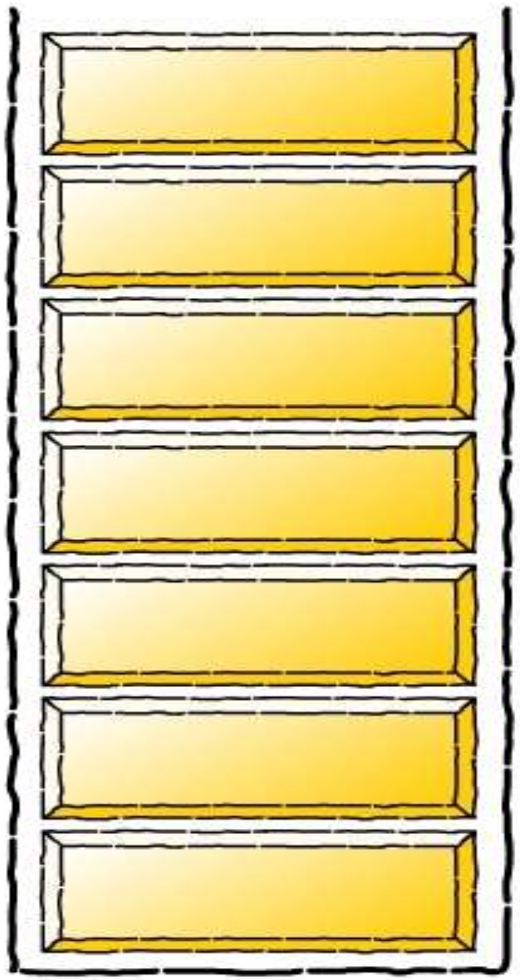
[End of if]

[End of Step 2 loop]

Step 7: Pop the elements from the Output Stack(Stack2).

Step 8: Exit

Infix to Prefix Conversion



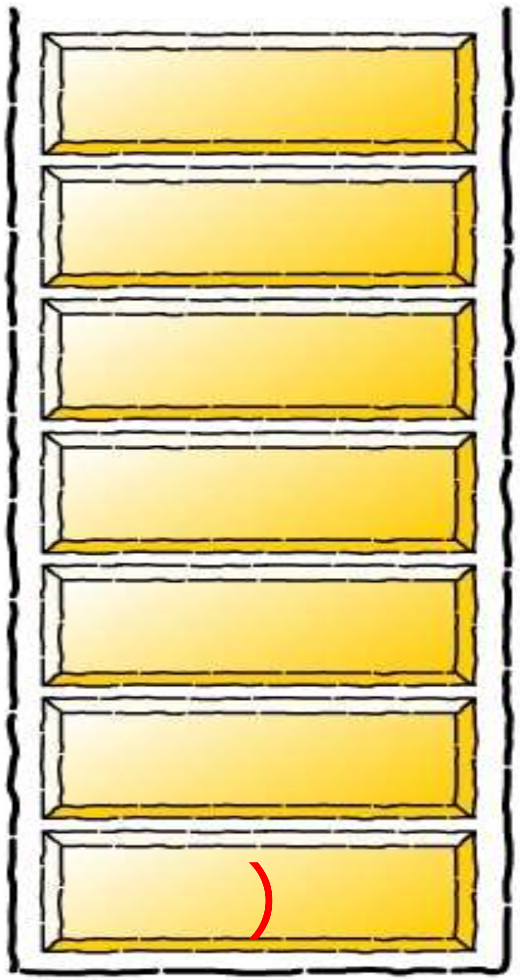
Stack

Infix Expression

(a + b - c) * d – (e + f)

Output Stack(Stack2)

Infix to Prefix Conversion



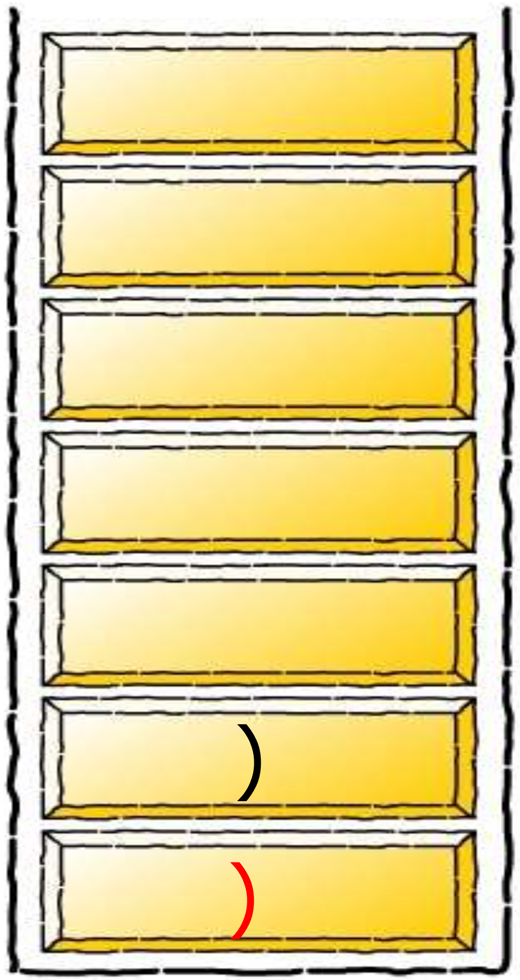
Stack

Infix Expression

((a + b - c) * d – (e + f)

Output Stack(Stack2)

Infix to Prefix Conversion



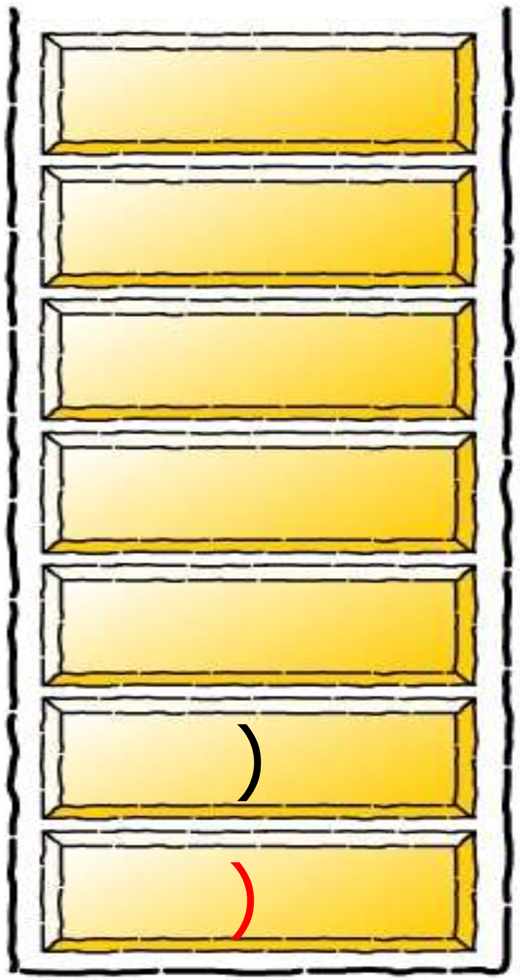
Stack

Infix Expression

((a + b - c) * d – (e + f

Output Stack(Stack2)

Infix to Prefix Conversion



Stack

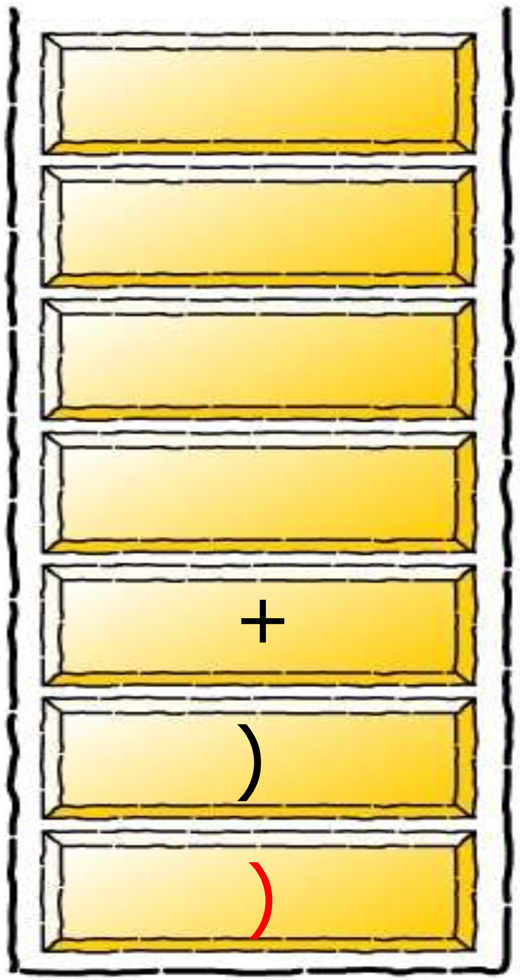
Infix Expression

((a + b - c) * d – (e
+

Output Stack(Stack2)

f

Infix to Prefix Conversion



Stack

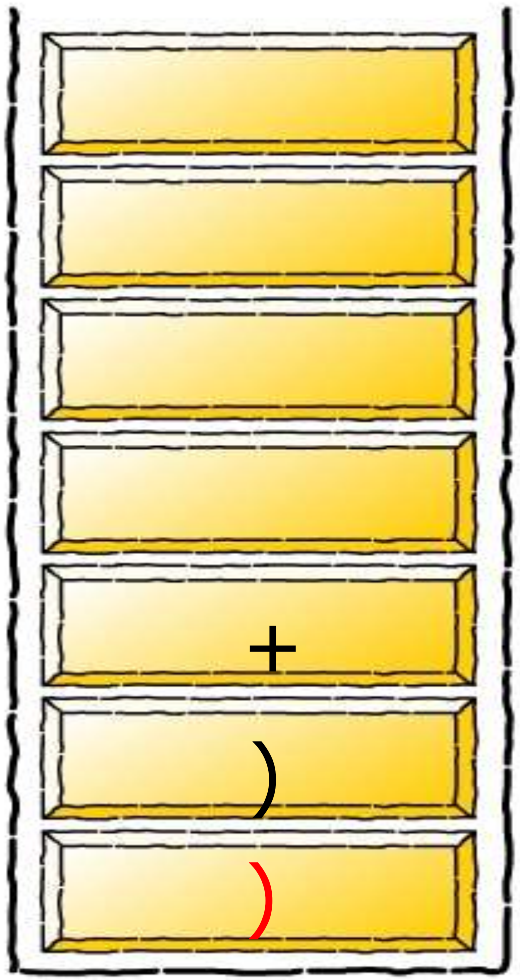
Infix Expression

((a + b - c) * d – (e

Output Stack(Stack2)

f

Infix to Prefix Conversion



Stack

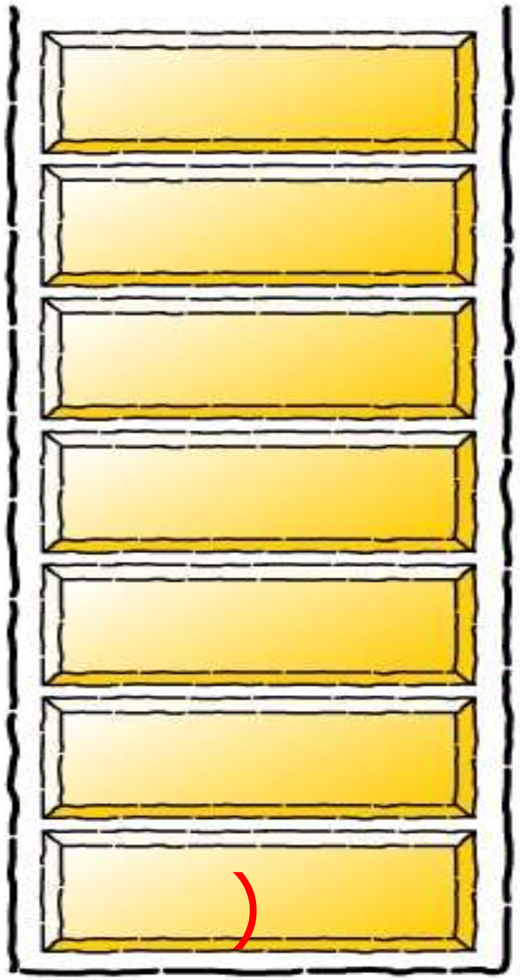
Infix Expression

((a + b - c) * d –(

Output Stack(Stack2)

f e

Infix to Prefix Conversion



Stack

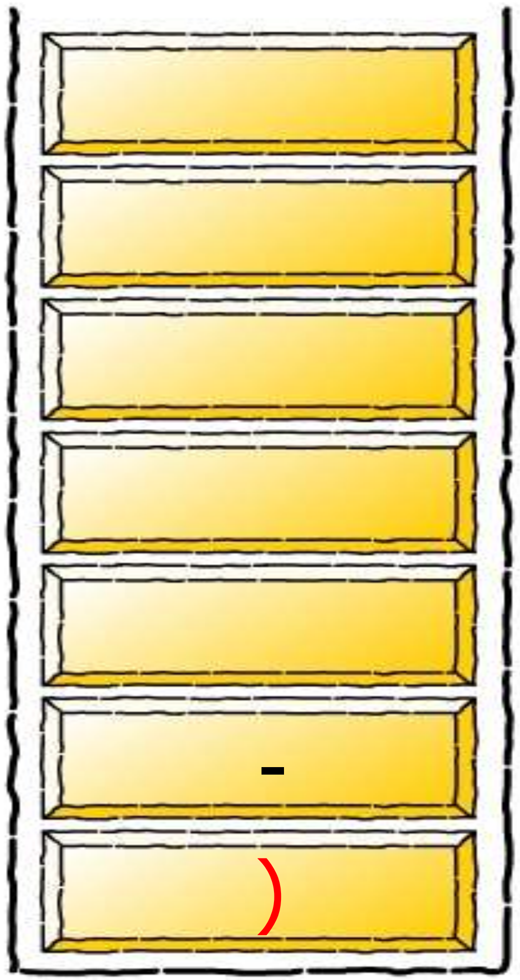
Infix Expression

((a + b - c) * d –

Output Stack(Stack2)

f e +

Infix to Prefix Conversion



Stack

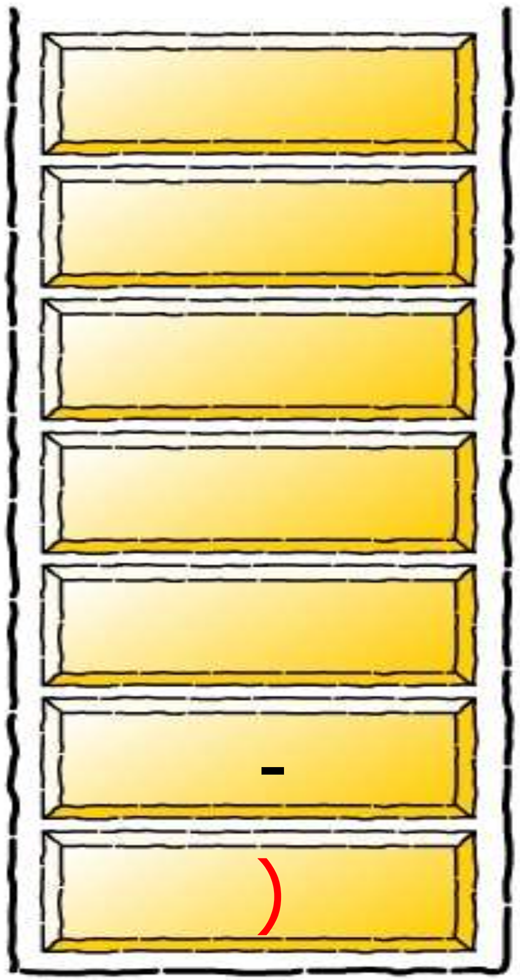
Infix Expression

((a + b - c) * d

Output Stack(Stack2)

f e +

Infix to Prefix Conversion



Stack

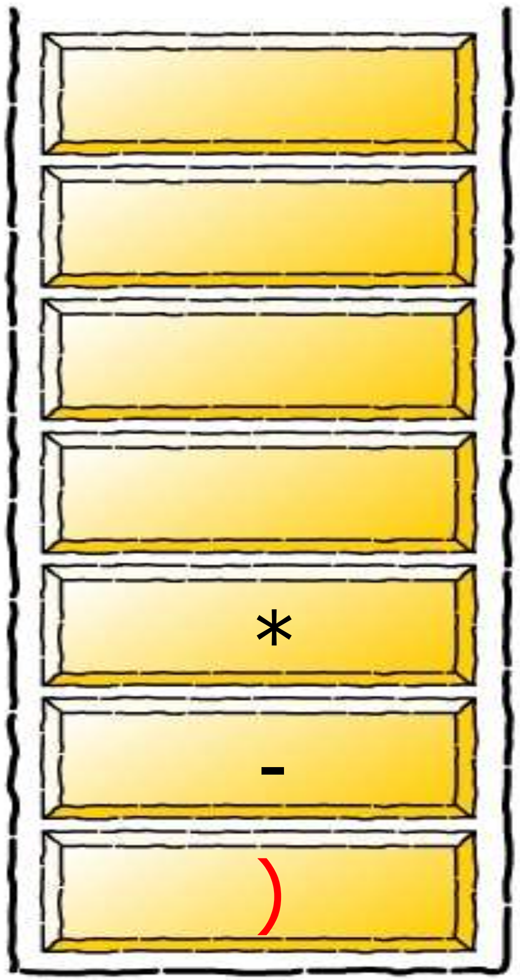
Infix Expression

((a + b - c) *

Output Stack(Stack2)

f e + d

Infix to Prefix Conversion



Stack

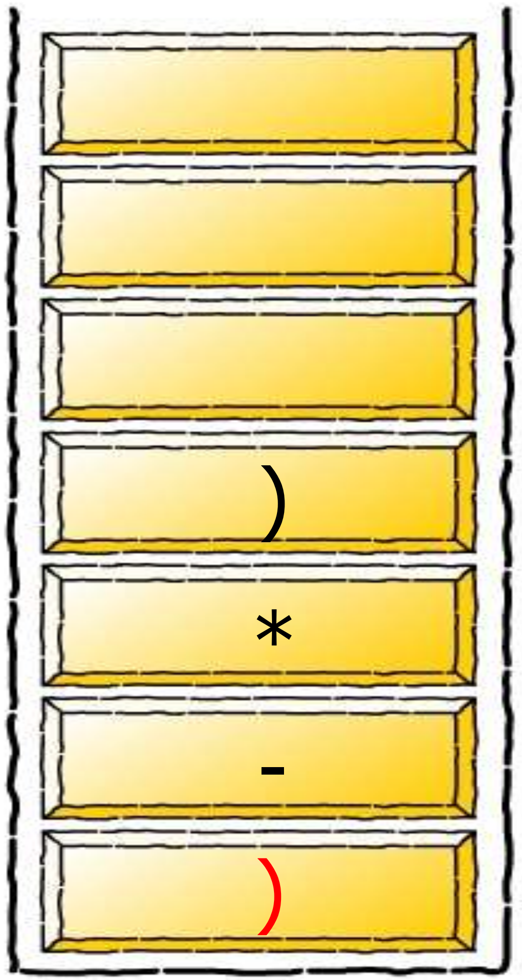
Infix Expression

((a + b - c)

Output Stack(Stack2)

f e +
d

Infix to Prefix Conversion



Stack

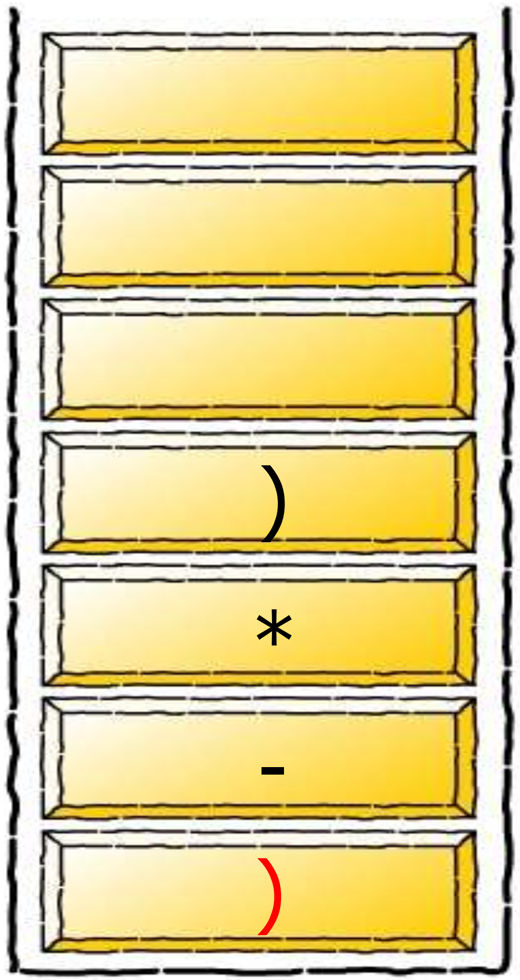
Infix Expression

((a + b - c

Output Stack(Stack2)

f e +
d

Infix to Prefix Conversion



Stack

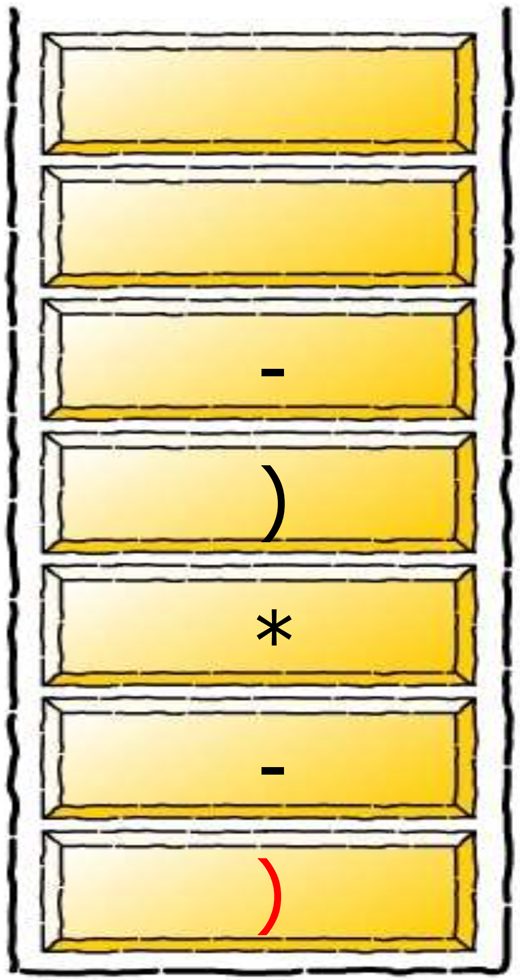
Infix Expression

((a + b -

Output Stack(Stack2)

f e + d c

Infix to Prefix Conversion



Stack

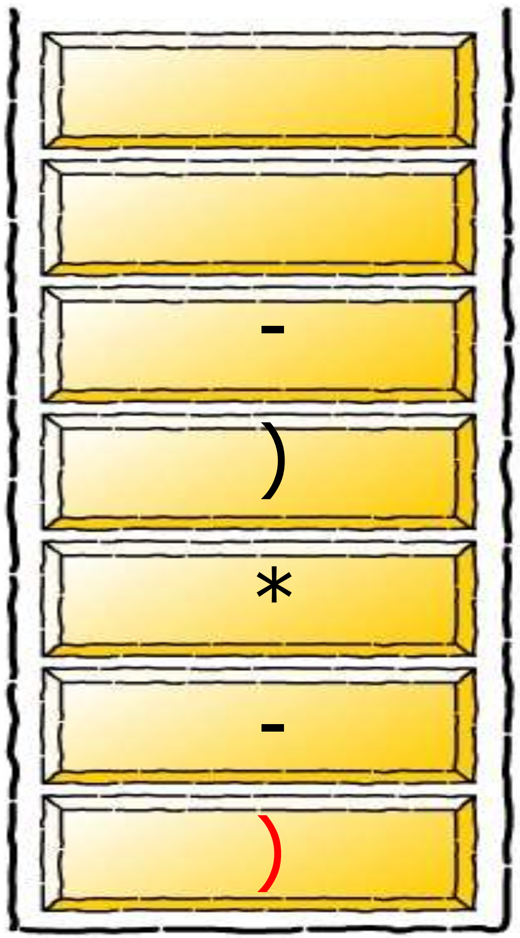
Infix Expression

((a + b

Output Stack(Stack2)

f e + d c

Infix to Prefix Conversion



Stack

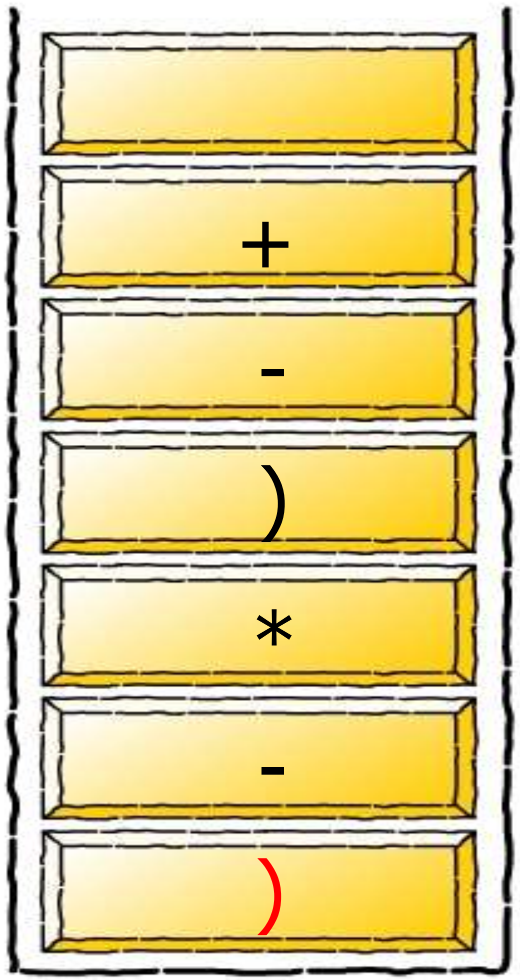
Infix Expression

((a +

Output Stack(Stack2)

f e + d c b

Infix to Prefix Conversion



Stack

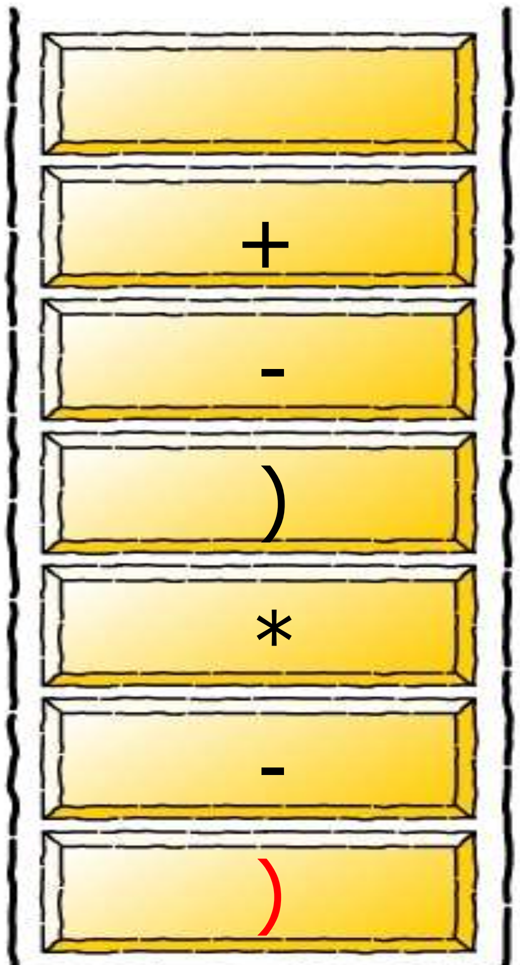
Infix Expression

((a

Output Stack(Stack2)

f e + d c b

Infix to Prefix Conversion



Stack

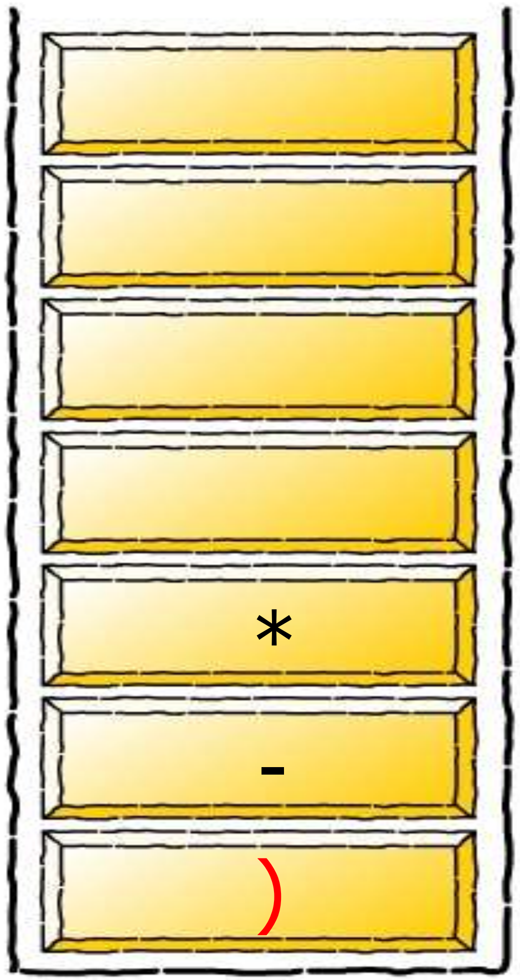
Infix Expression

((

Output Stack(Stack2)

f e + d c b a

Infix to Prefix Conversion



Stack

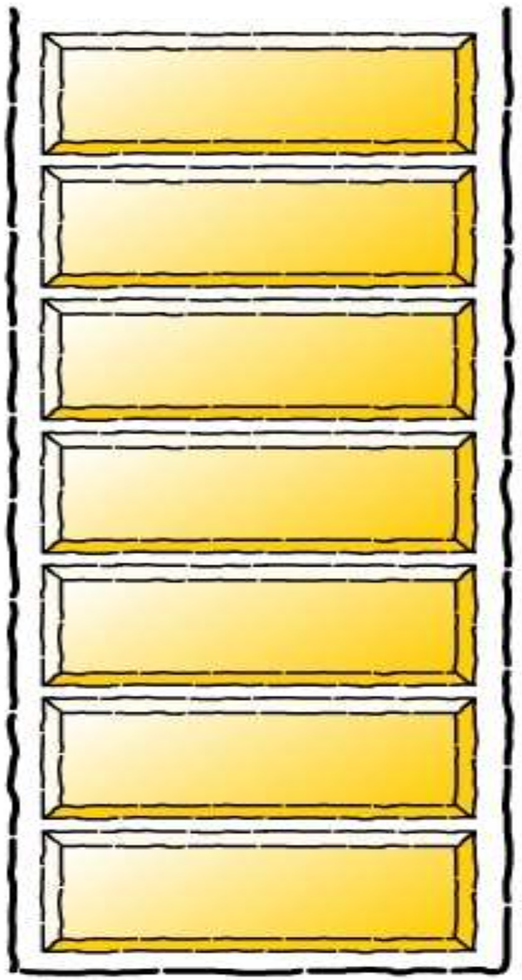
Infix Expression

(

Output Stack(Stack2)

f e + d c b a + -

Infix to Prefix Conversion



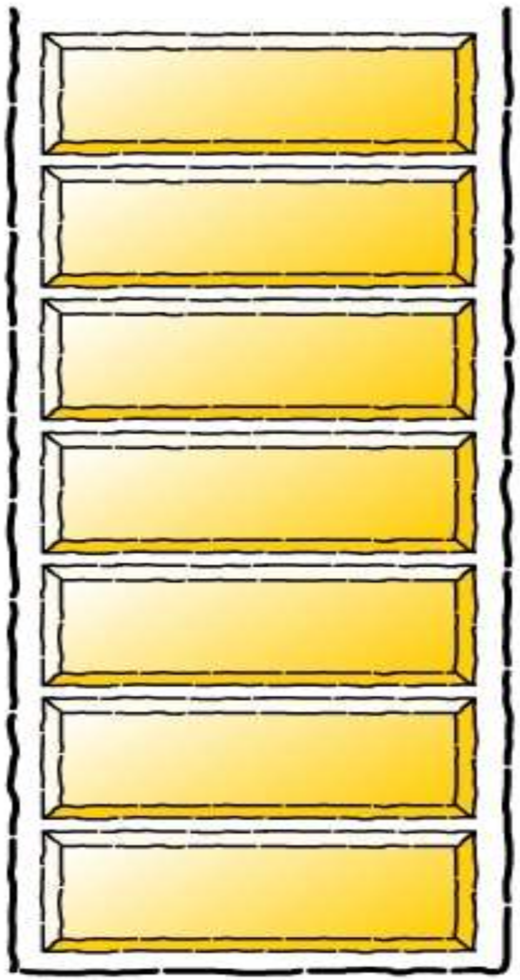
Stack

Infix Expression

Output Stack(Stack2)

f e + d c b a + - * -

Infix to Prefix Conversion



Stack

Infix Expression

Output Stack(Stack2)

f e + d c b a + - * -

Prefix Expression

- * - + a b c d + e f

Evaluation of Prefix Expression

ALGORITHM: EVALUATE_PREFIX (P)

Input: P is an arithmetic expression in prefix notation.

Output: This algorithm evaluates the prefix expression P and assigns the result to the variable VALUE.

Step 1: Add a left parentheses '(' at the beginning of P.

Step 2: Scan P from right to left and repeat Step 3 to Step 4 for each element of P until the '(' is encountered.

Step 3: If an operand is encountered, push it on to the STACK.

Evaluation of Prefix Expression

Step 4: If an operator OP is encountered, then

(a)Pop the two top elements of the STACK, where A is the top element and B is the next to top element.

(b)Evaluate A OP B.

(c)Push the result of the evaluation(b) on to the STACK.

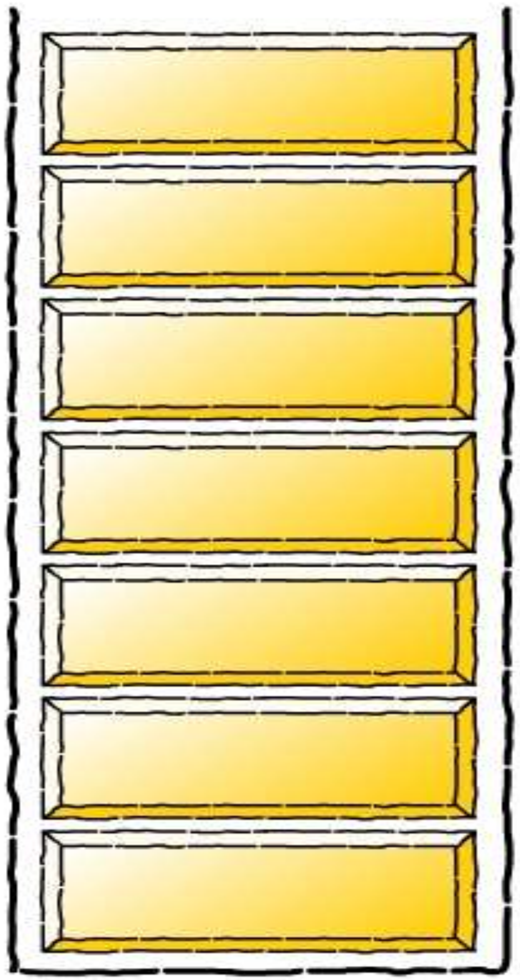
[End of If]

[End of Step 2 loop]

Step 5: Set VALUE equal to the top most element of the STACK.

Step 6: EXIT

Evaluation of Prefix Expression



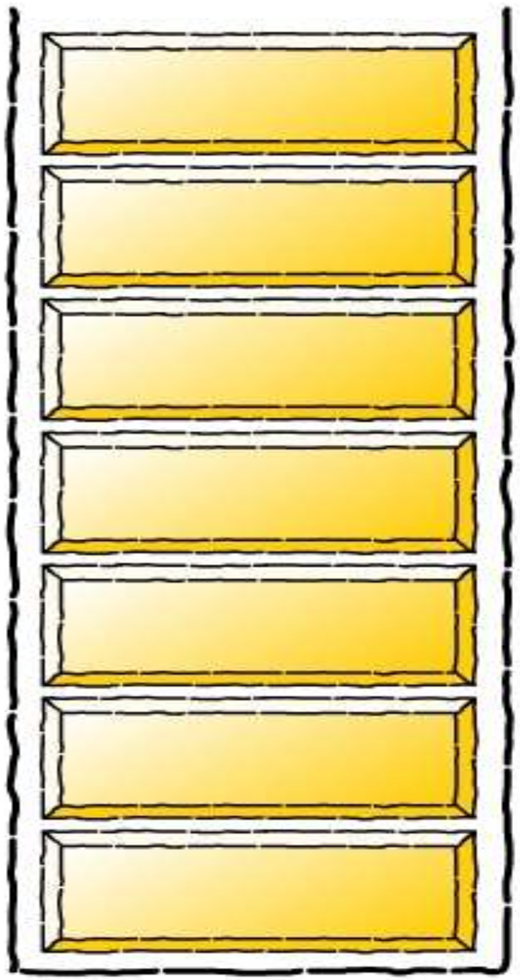
Stack

Prefix Expression

- * 5 + 6 2 / 12 4

Result

Evaluation of Prefix Expression

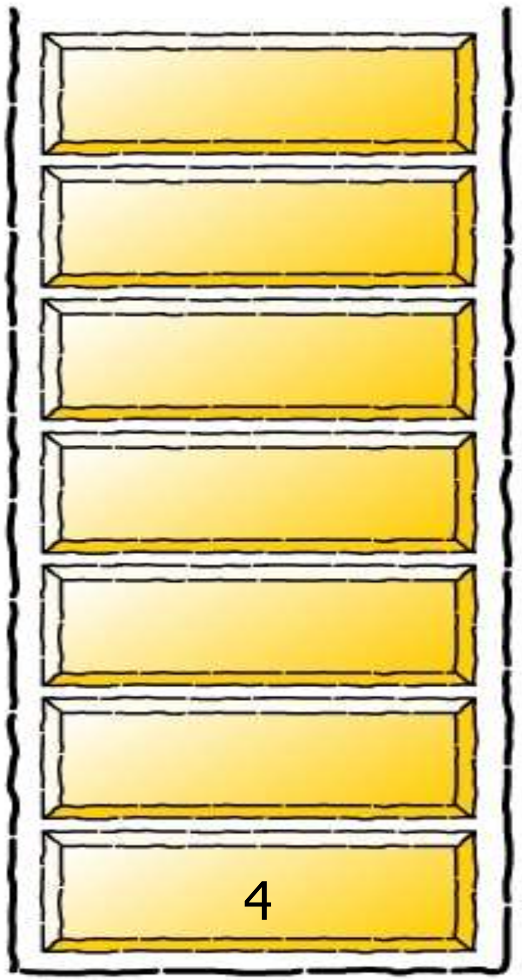


Prefix Expression

(- * 5 + 6 2 / 12 4

Result

Evaluation of Prefix Expression



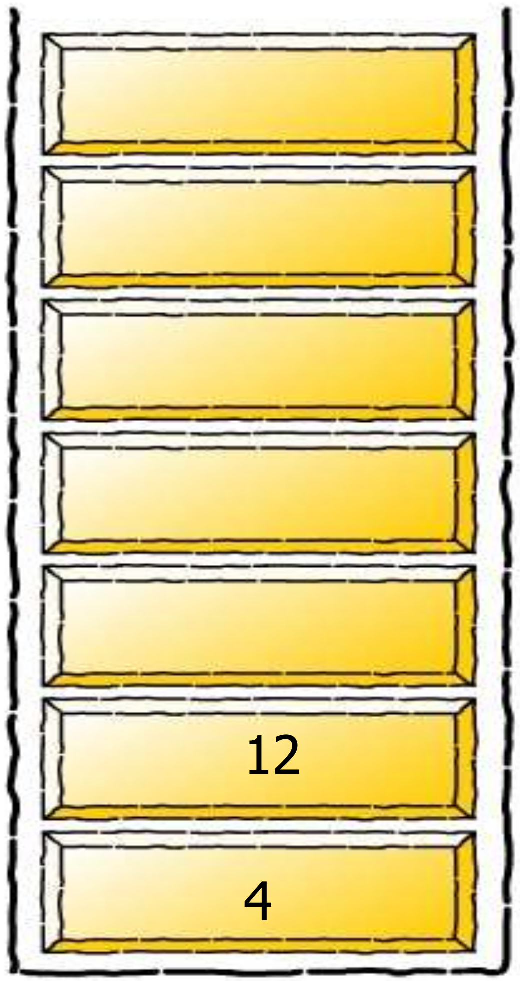
Stack

Prefix Expression

(- * 5 + 6 2 / 12

Result

Evaluation of Prefix Expression



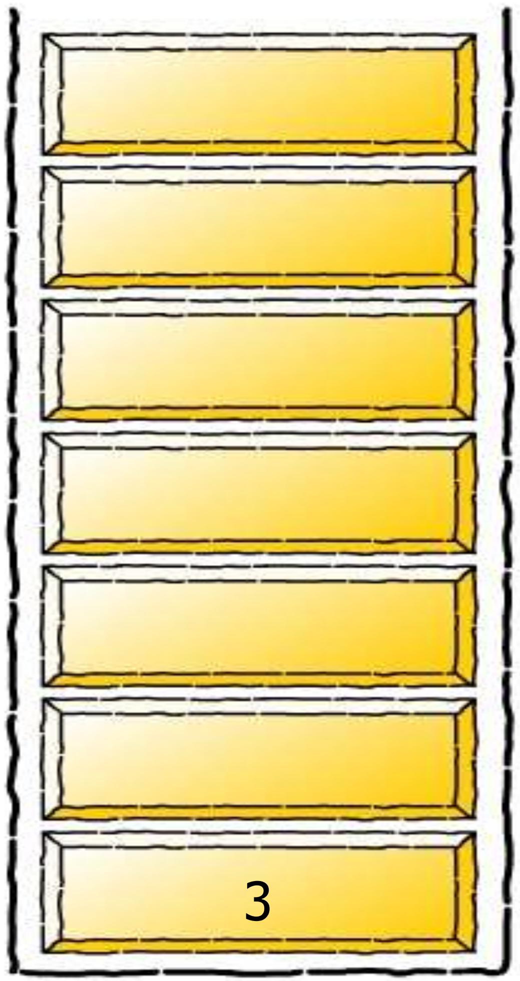
Stack

Prefix Expression

(- * 5 + 6 2 /

Result

Evaluation of Prefix Expression



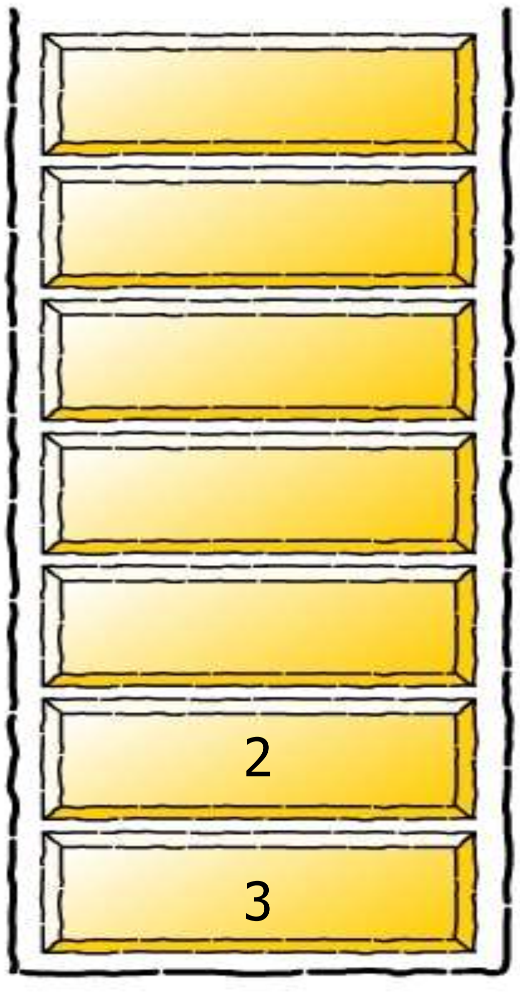
Stack

Prefix Expression

(- * 5 + 6 2

Result

Evaluation of Prefix Expression

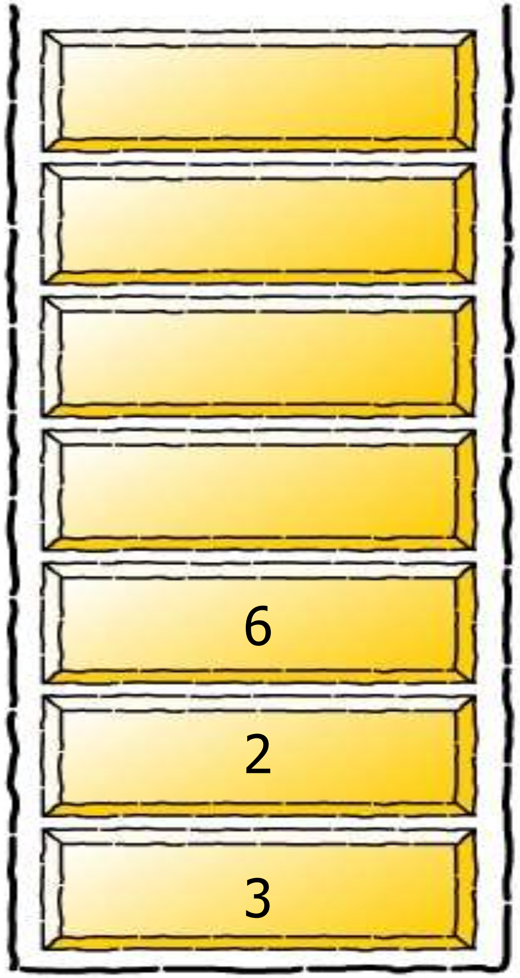


Prefix Expression

(- * 5 + 6

Result

Evaluation of Prefix Expression



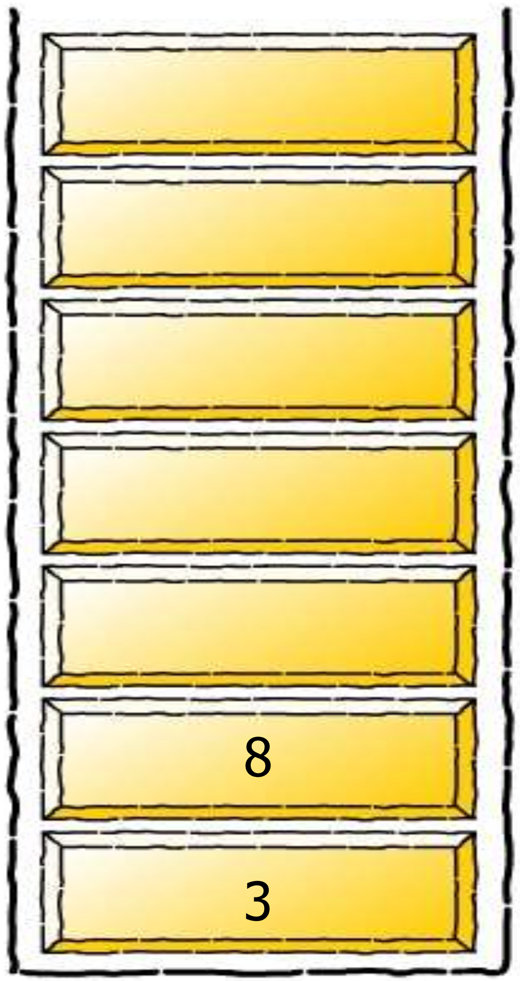
Stack

Prefix Expression

(- * 5 +

Result

Evaluation of Prefix Expression



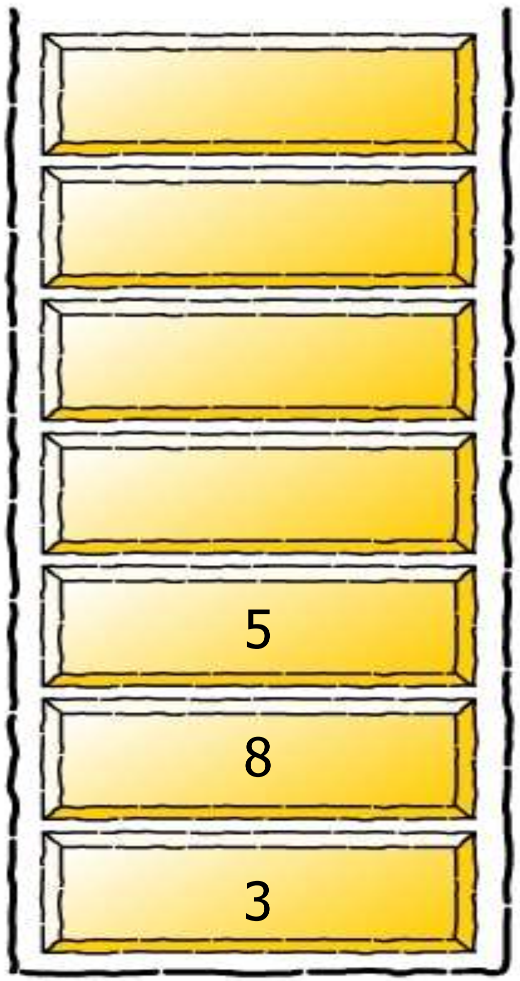
Stack

Prefix Expression

(- * 5

Result

Evaluation of Prefix Expression



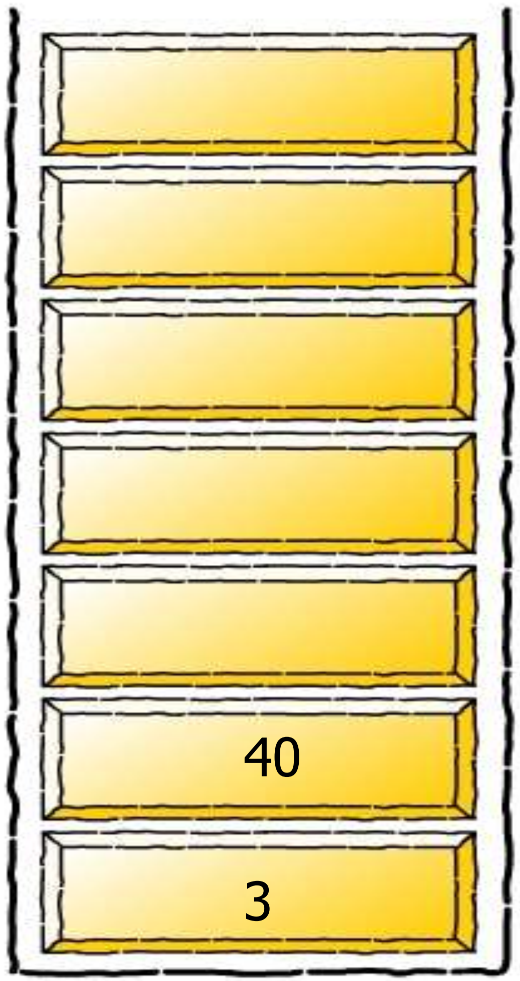
Stack

Prefix Expression

(- *

Result

Evaluation of Prefix Expression



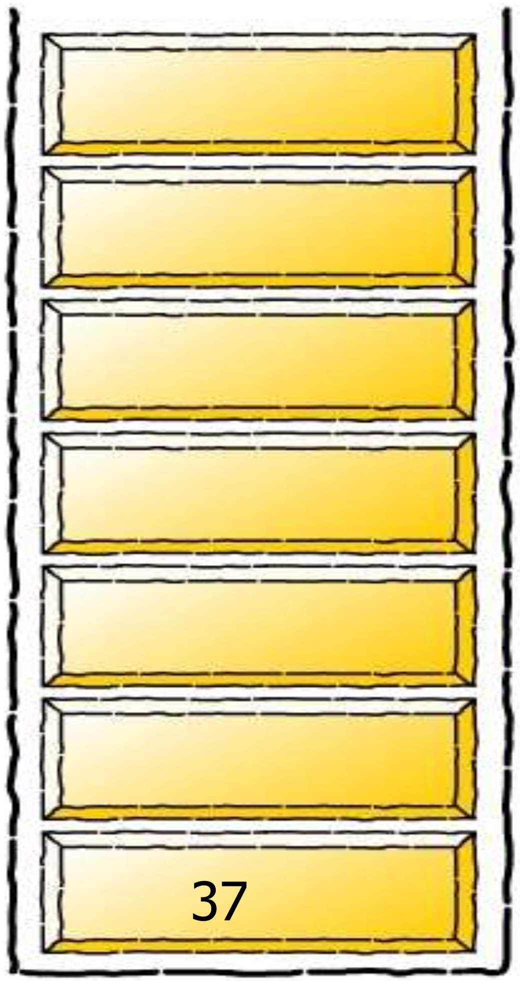
Stack

Prefix Expression

(-

Result

Evaluation of Prefix Expression



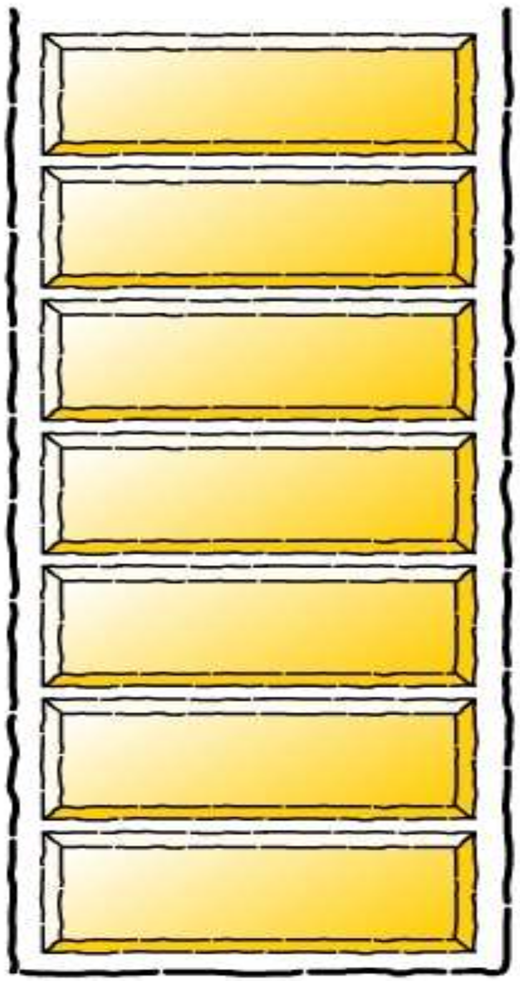
Stack

Prefix Expression

(

Result

Evaluation of Prefix Expression



Stack

Prefix Expression

Result

37

Example Infix to Prefix

TRACING THE ALGORITHM:

Infix string: A+B*C+D/E

Ch	prefix	stackop
E	E	
/	E	/
D	ED	/
+	ED/	+
C	ED/C	+
*	ED/C	+, *
B	ED/CB	+, *
+	ED/CB*	+, +
A	ED/CB*A	+, +
	ED/CB*A+	+
	ED/CB*A++	

Reverse of is ++A*BC/DE.

The prefix expression of A+B*C+D/E is ++A*BC/DE.

Recursion

- Process in which a function calls itself directly or indirectly is called recursion
- corresponding function is called as recursive function
- Using recursive algorithm, certain problems can be solved quite easily
- Example: Findout factorial of a number

```
int fact(int n) {
    if (n <= 1) return 1;
    else return n * fact(n - 1);
}
```

Demonstrate working of Recursion

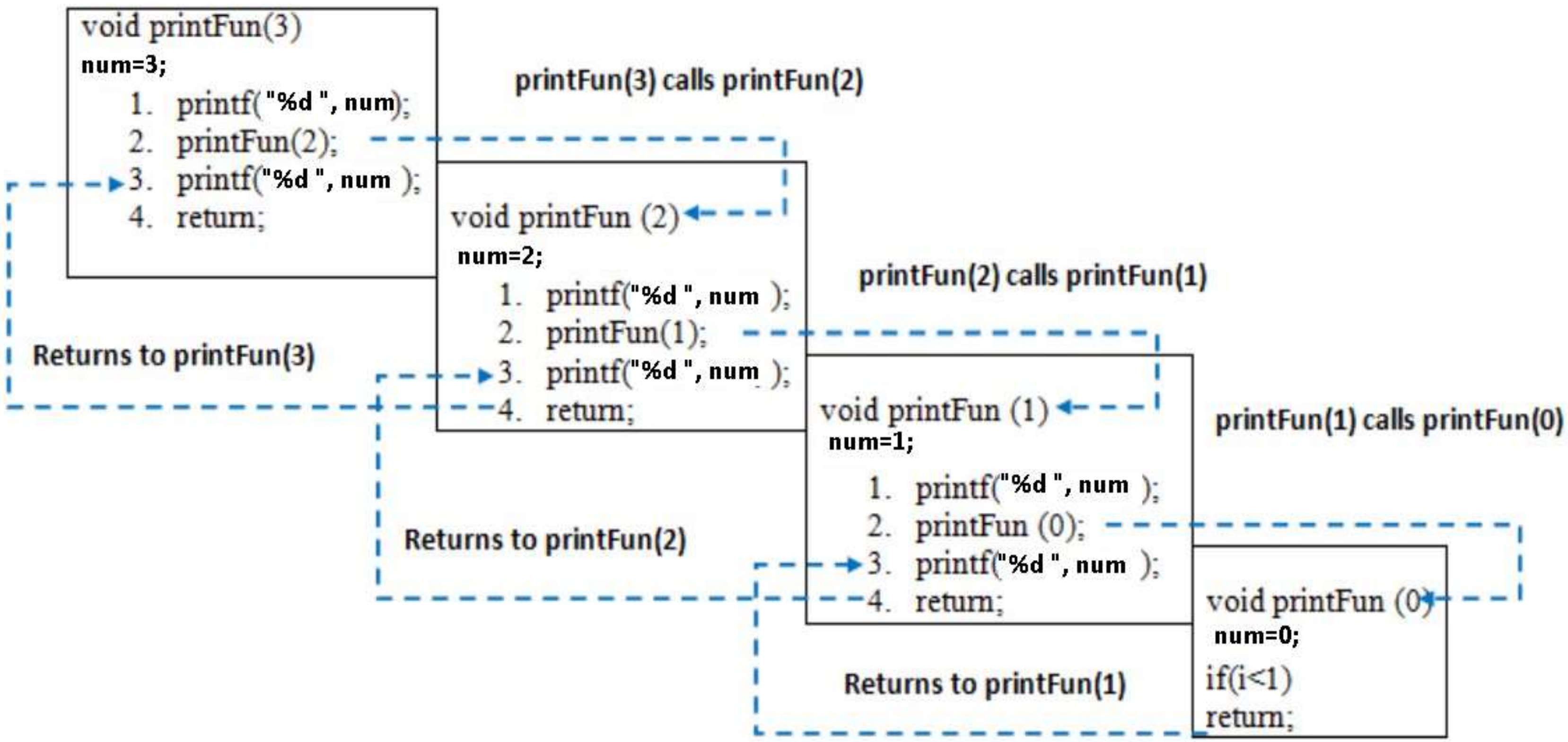
```
void printFun(int num) {
    if (num < 1)
        return;
    else {
        printf("%d ", num);
        printFun(num - 1); // statement 2
        printf("%d ", num);
        return;
    }
}
```

```
int main() {
    int n = 3;
    printFun(n);
}
```

Output :

3 2 1 1 2 3

Demonstrate working of Recursion



Recursion

```
int fact(int n) {
    if (n <= 1) return 1;
    else return n * fact(n - 1);
}
```

If call x = fact(3), stores n=3 on the stack

| fact calls itself, putting n=2 on the stack

| | fact calls itself, putting n=1 on the stack

| | fact returns 1

| fact has n=2, computes and returns 2*1 = 2

fact has n=3, computes and returns 3*2 = 6

Classic: Factorial

```
int  fact (int n)     {
    if (n<=1)
          return (1)   /*  base case  */
    else
          return (n * fact (n-1) );
            /* recursive case
}      // end factorial()
```

main calls argument n = 7.

fact(7)

value of n at this node: returned value

n=7
return (7*fact(6)) — return(7*720) = 5040 = answer!!

recursive call
n=6
return(6*fact(5)) — return(6*120) = 720

n=5
return(5*fact(4)) — return(5*24) = 120

n=4
return (4*fact(3)) — return(4*6) = 24

n=3
return (3*fact(2)) — return (3 *2) = 6

n=2
return (2*fact(1)) — return(2 * fact(1)) = 2 * 1 = 2 Thus fact (2) = 2.
return 2.
n=1 (return (1)) — 1 is substituted for the call (base case reached)

Program: Linear implementation of stack Using Structure

```
#include<stdio.h>
#define MAX 50
typedef struct {
   int stk[MAX];
   int top;
}Stack;

void push(Stack *s, int item) {
   if(s->top==MAX-1){
            printf("\nStack Overflow...\n");
            return;
      }
   s->stk[++s->top]=item;
}
```

```
void pop(Stack *s, int *item) {
   if(s->top==-1)
      {
            printf("\nStack Underflow...\n");
            return;
      }
   *item=s->stk[s->top];
   s->top--;
}
```

Program: Linear implementation of stack Using Structure

```
void display(Stack *s) {
    int i;
    if(s->top == -1) {
        printf("Stack is Empty...\n");
              return;
        }
    printf("\nThe elements in the stack
              are...\n");
    for(i=s->top; i>=1; i--)
        printf("%d->", s->stk[i]);
    printf("%d", s->stk[i]);
    printf("\n");
}
```

```
int main(){
    Stack s;
    int num;
    s.top=-1;
    int choice=0;
    do {
        printf("\nStack Options...\n");
        printf("\n1: Add item\n");
        printf("\n2: Remove item \n");
        printf("\n3: Display\n");
        printf("\n0: Exit\n");
        printf("\n\nEnter choice: ");
        scanf("%d",&choice);
```

Program: Linear implementation of stack Using Structure

```
switch(choice) {
    case 0:
        break;
    case 1:
        printf("\nEnter an item to be
                    inserted: ");
        scanf("%d", &num);
        push(&s, num);
        break;
    case 2:
        pop(&s, &num);
        printf("\nThe popped element
                    is %d\n", num);
        break;
    case 3:
        display(&s);
        break;
    default:
        printf("\nAn Invalid
                    Choice !!!\n");
    }
  }while(choice!=0);
  return 0;
}
```

Questions

1. The term "push" and "pop" is related to the

    a. Array
    b. Lists
    c. Stack
    d. All of above

c. Stack

Questions

2. Which of the following name does not relate to stacks?

a. FIFO lists
b. LIFO list
c. Piles
d. Push-down lists

a. FIFO Lists

Questions

4. The data structure which is one ended is ………………..

a) Queue
b) Stack
c) Tree
d) Graph

b) Stack

Questions

5.In the stack, if user try to remove element from the empty stack then it is called -----

a)Garbage Collection
b)Overflow Stack
c)Underflow Stack
d)Empty Collection

c) Underflow Stack

Questions

6. User push 1 element in the stack having already five elements and having stack size as 5 then stack becomes-----

a)User flow
b)Underflow
c)Crash
d)Overflow

d)Overflow

Questions

7.User perform the following operations on stack size 5,Then

Push(1), pop(), push(2), push(3), pop(), push(4), pop(), pop(), push(5)

At the end of last operation, total number of elements present in the stack is----

a)3

b)1

c)2

d)4

b) 1

Questions

8.For the following operations on stack of size 5 then

Push(1), pop(), push(2), push(3), pop(), push(4), pop(), pop(), push(5), pop(), pop(), push(6)

Which of the following statement is correct for stack

a)Underflow Occurs
b)Overflow Occurs
c)Stack operation will be performed smoothly
d)None of these

a)Underflow Occurs

Questions

9. What will be the postfix expression for the following infix expression

b*c+d/e

a. b*cde/+
b. bc*de/+
c. bcd*e/+
d. bc*de+/

b. bc*de/+

Questions

10. Expressions in which operator is written after the operand is called-------

a. Infix Expression
b. Prefix Expression
c. Postfix Expression

c. Postfix Expression

Questions

11.Stack can not be used to ------

a. Implementation of Recursion
b. Evaluation of postfix expression
c. Allocating resources and scheduling
d. Reversing String

c. Allocating resources and scheduling

Questions

12. Well formed parentheses can be implemented using

a. list
b. queue
c. hash map
d. stack

d. stack

Questions

13.Find the equivalent prefix expression from the following infix expression a+b-c/d*(e-f)/g

-+ab/*/cd-efg