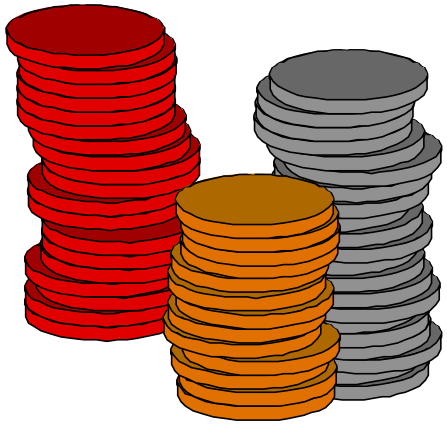
Stack
&
its Applications

Amiya Ranjan Panda

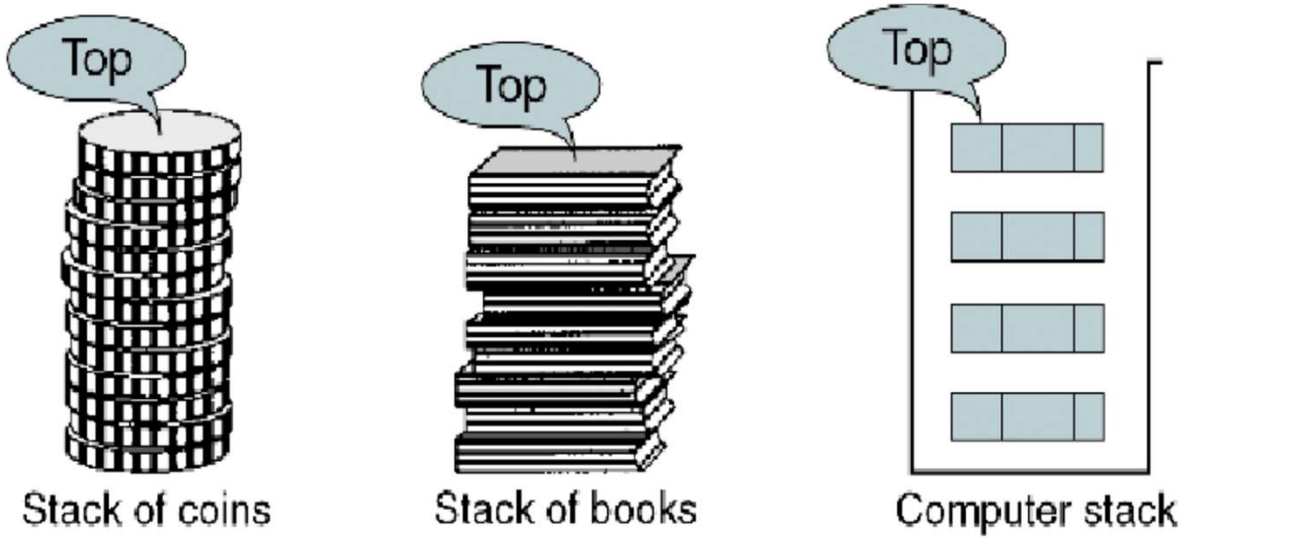
What is a stack?

- 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:



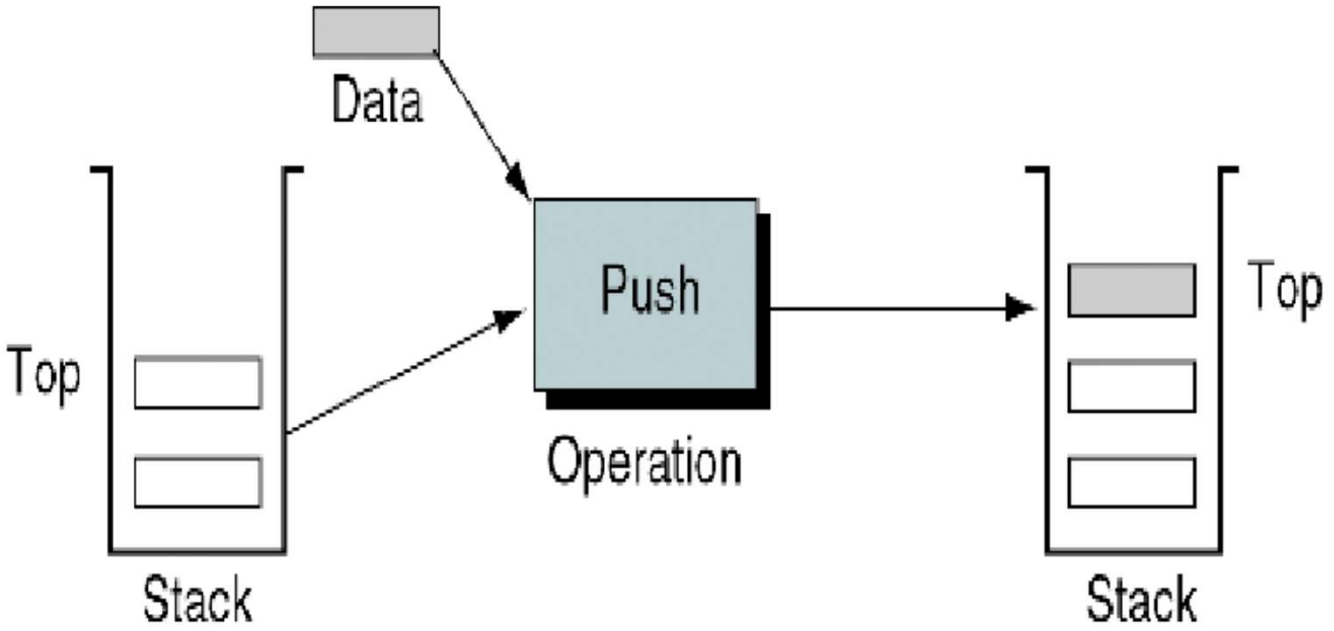
- Elements are removed in the reverse order in which they were inserted

Examples of Stack



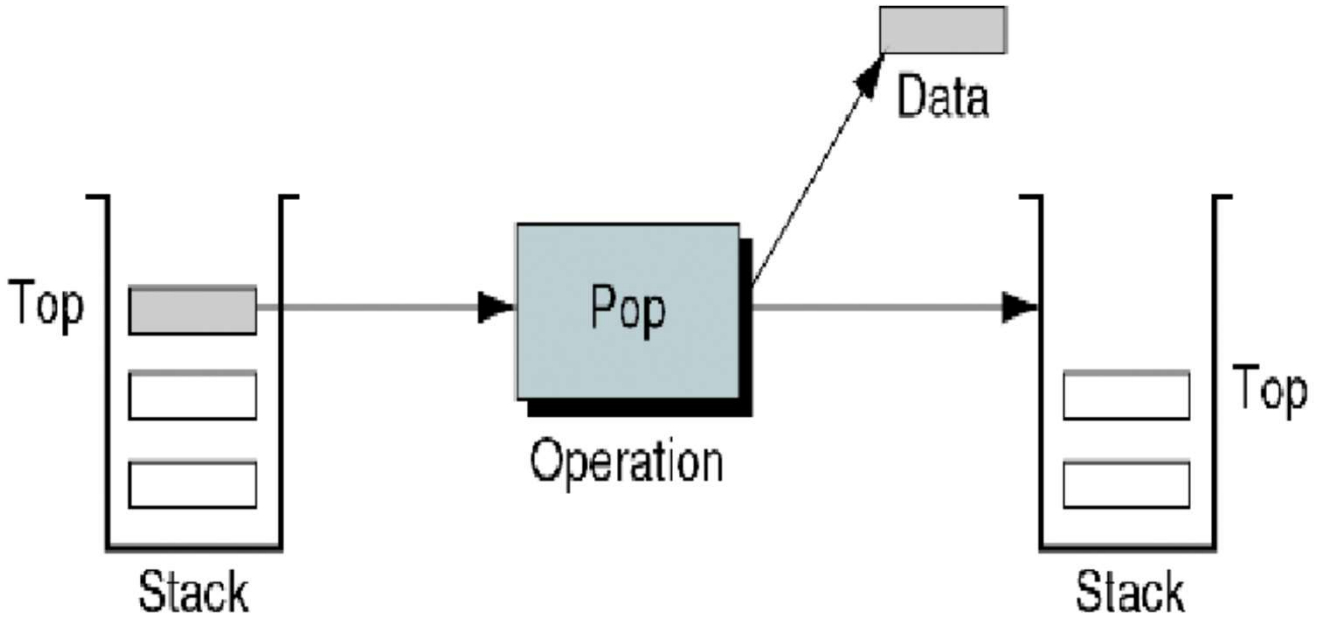
Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).

Operations on Stack



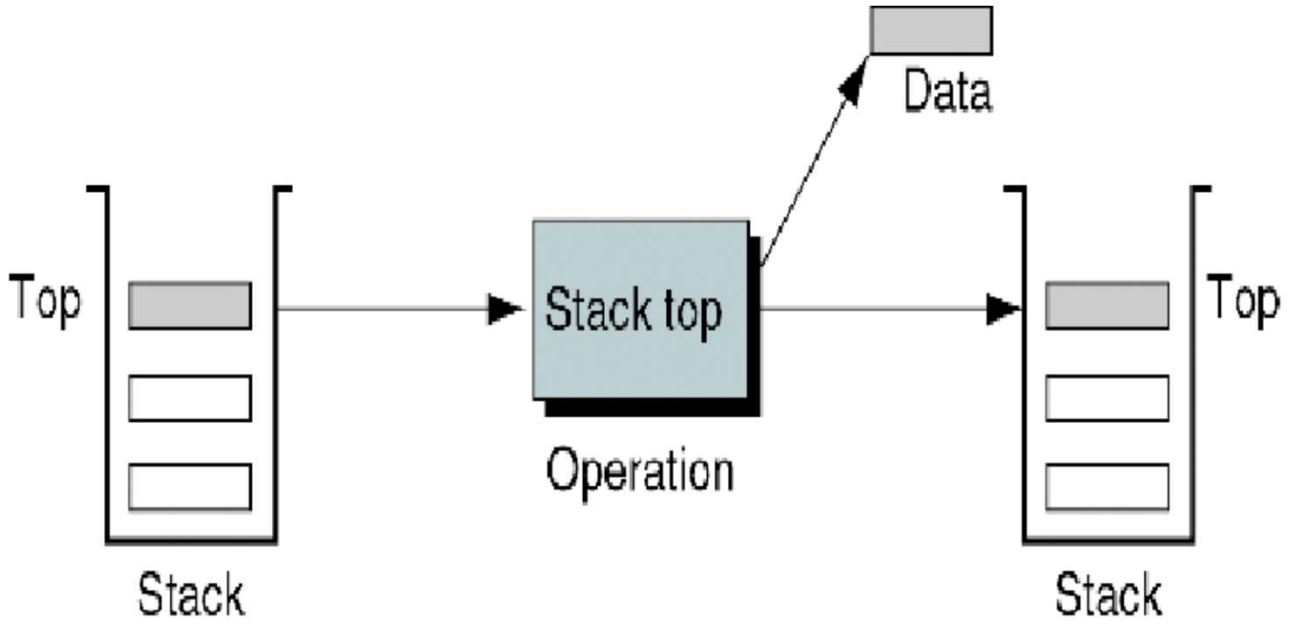
Push Stack Operation

Operations on Stack



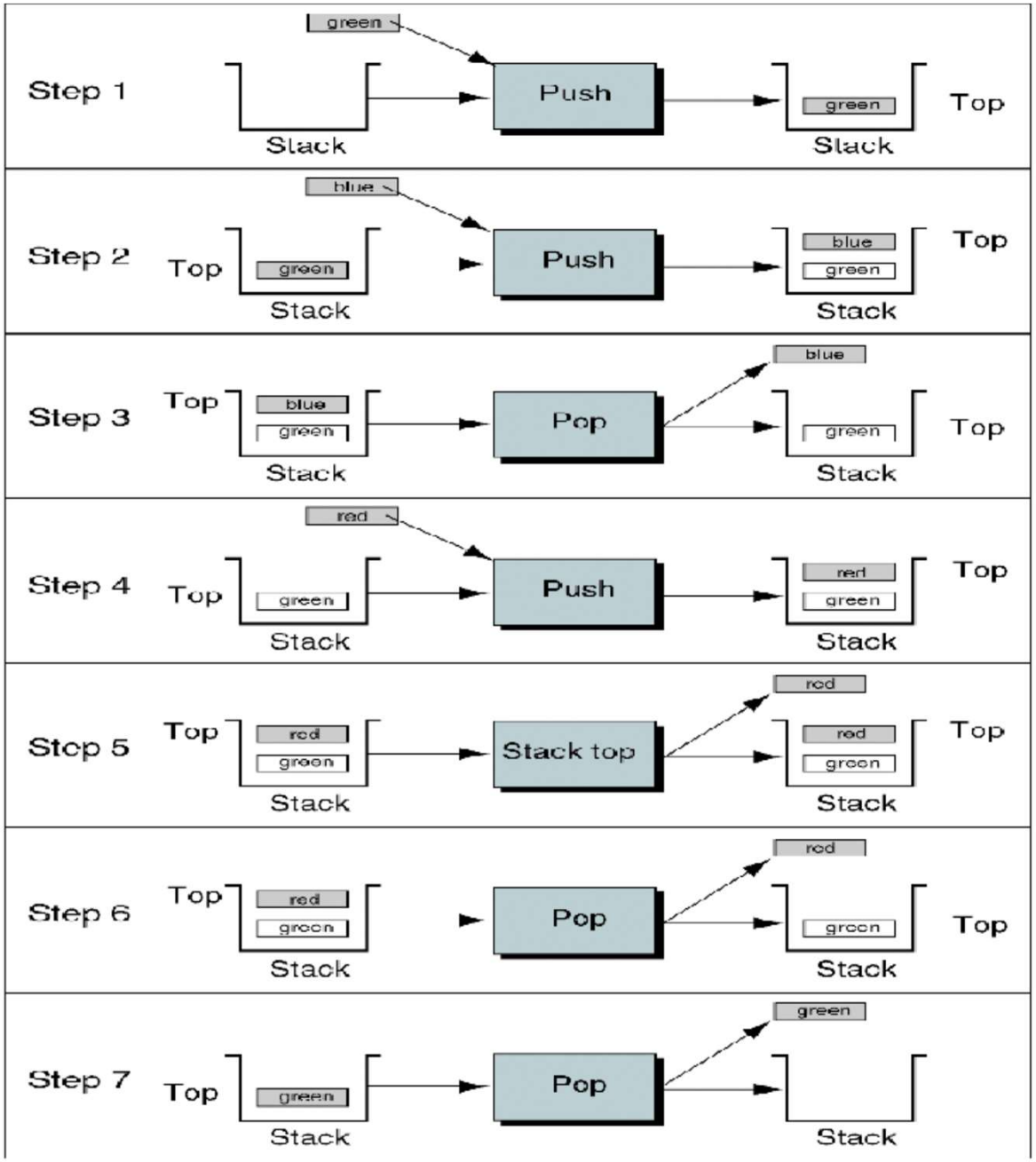
Pop Stack Operation

Operations on Stack

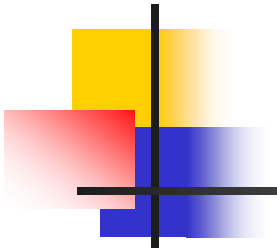


Stack Top Operation

Examples of Stack

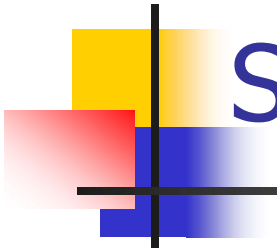


Stack Example



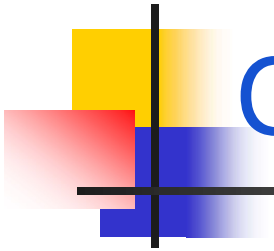
Stack Applications

- 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

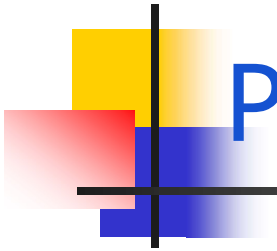
Stack createS(stack_size) =

```
#define STACK_SIZE 100          /* maximum stack size */

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

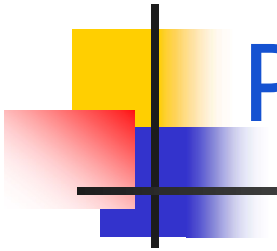
Boolean isEmpty(Stack) = top= -1;

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



Push

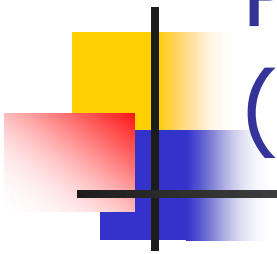
```
void push(element item)  {
    /* add an item to the global stack */
    if (top == STACK_SIZE-1)  {
        printf("Stack is Full");
        return;
    }
    stack[++top] = item;
}
```



Pop

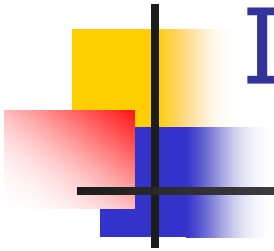
```
element pop()  {
    /* return the top element from the stack */
    if (top == -1)  {
            printf("Stack is Empty …");
            return;
     }
    return stack[top--];
 }
```

Performance and Limitations (Implementation of stack ADT using Array)



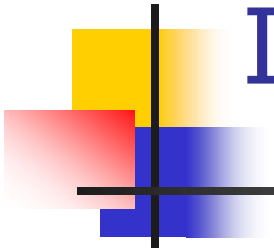
- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori* , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific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

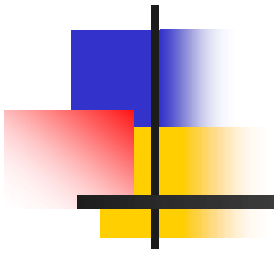
```
// Method to pop an element from second stack
int pop2() {
   int x;
   if(top2 == n)  {
      printf("Stack Underflow...");
      return -999;
    }
    x = stack[top2];
    top2++;
    return(x);
}
```

Reverse a String using Stack

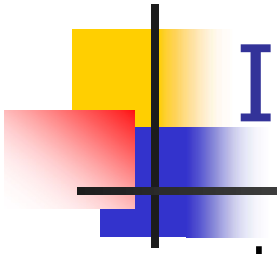
```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
char str[MAX];
int top = -1;
int main() {
    char str[MAX];
    int i;
    printf("Input a string: ");
    scanf("%[^\n]", str);
    for(i=0; i<strlen(str); i++) pushChar(str[i]);
    for(i=0; i<strlen(str); i++) str[i]=popChar();
    str[i] =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



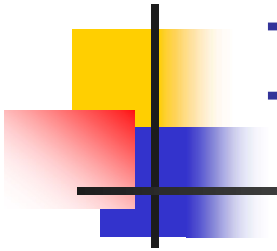
Infix Notation

- 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 :

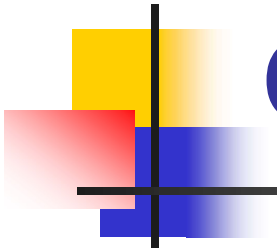
a + b * c = (a + b) * c ?

= a + (b * c) ?



Infix, Postfix, & Prefix notation

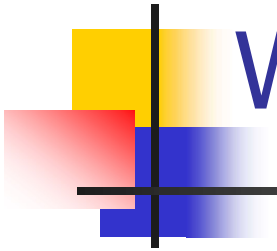
- There is no reason to place the operator somewhere else.
- How ?
  - Infix notation : a + b
  - Prefix notation : + a b
  - Postfix notation: a b +



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.**

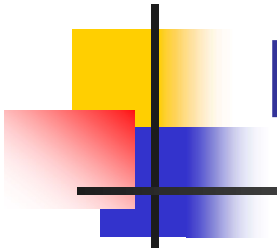


Why ?

- **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



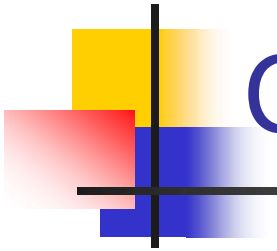
Example

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

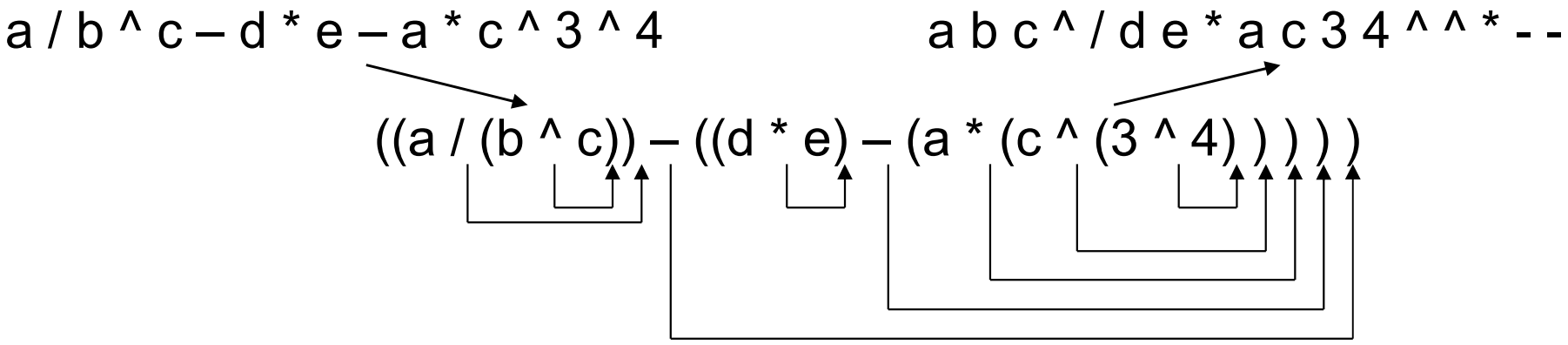
- Infix is the only notation that requires parentheses in order to change the order in which the operations are done.

Infix to Postfix conversion (Intuitive Algorithm)

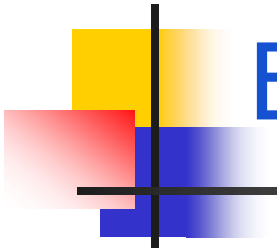
- An Infix to Postfix manual conversion algorithm is:
  1. Completely parenthesize the infix expression according to order of priority you want.
  2. Move each operator to its corresponding right parenthesis.
  3. Remove all parentheses.

- Examples:

3 + 4 * 5 ⟶ (3 + (4 * 5)) ⟶ 3 4 5 * +



Evaluation of Expressions



x = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1: ((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2: (4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Precedence of Operators

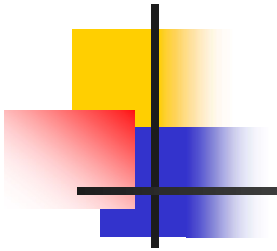
Token	Operator	Precedence	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
! - - + & * sizeof	logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type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



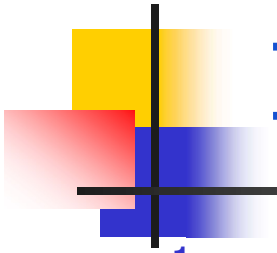
Postfix conversion

user compiler

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+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*-

Postfix: no parentheses, no precedence

Infix to postfix conversion

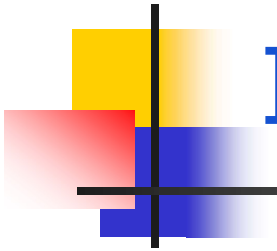


1. Read the tokens one by one from an infix expression using a loop.
2. For each token *x* do the following:
   i. When the *token* is an operand
      - Append to the end of the postfix expression
   ii. When the *token* is a left parenthesis "("
      - Push the token into the stack.
   iii. When the *token* is a right parenthesis ")"
      - Repeatedly pop a token from stack and append to the end of the postfix expression until "(" is encountered in the stack. Then pop "(" from stack.
      - If stack is empty before finding a "(", implies that expression is not a valid expression.

Infix to postfix conversion

iv. When the *token* is an operator

- Use a loop that checks the following conditions:
  a) The stack is not empty
  b) The token *y* currently at top of stack is an operator. In other words, it is not **not** a left parenthesis "(" .
  c) *y* is an operator of **higher or equal** precedence than that of *x*,
- As long as all these three conditions are true, repeatedly do the following:
  - Append the token *y* into postfix expression
  - Call pop to remove another token, named *y*, from stack



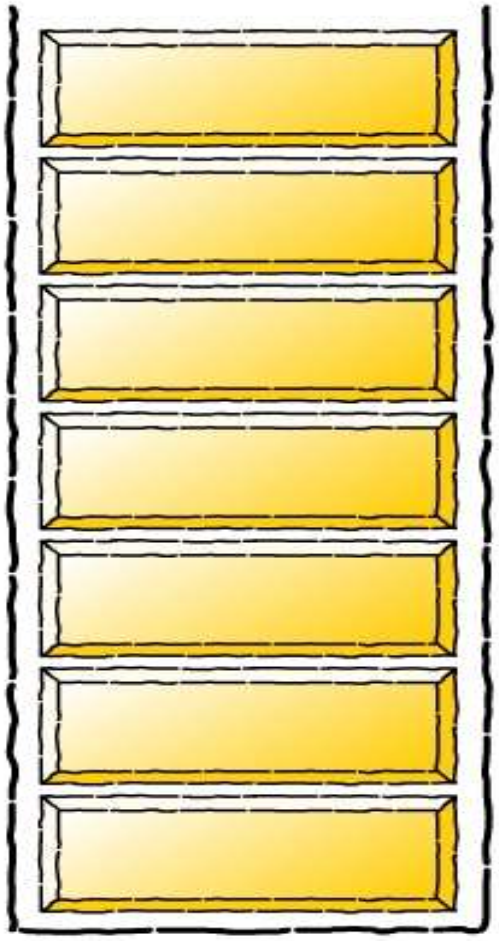
Infix to postfix conversion

- Note: The loop above will stops as soon as any of the three conditions is not true.
- Then, push the token *x* into stack.

- After all the tokens in infix expression are processed, then use another loop to repeatedly do the following as long as the stack is not empty:
  - Append the token from the top of the stack into postfixexpression.
  - Call pop to remove the top token *y* from the stack.

Infix to postfix conversion: Example

stack



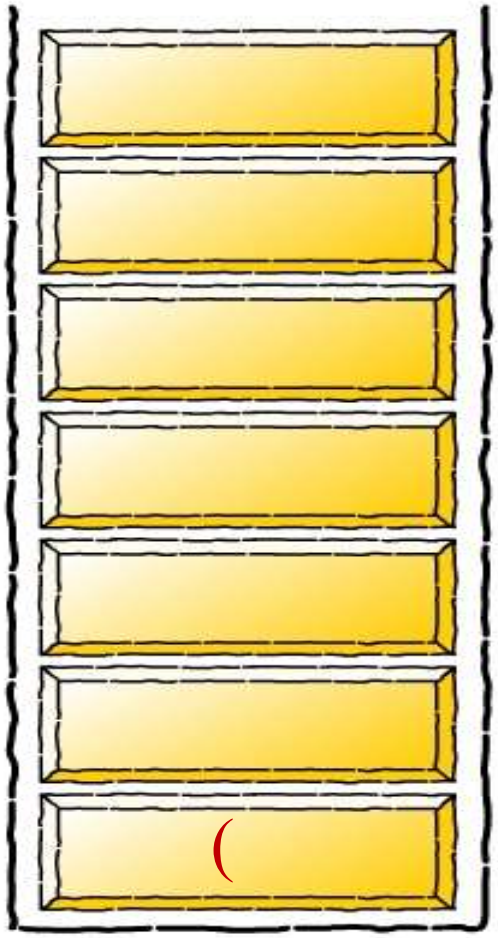
infix expression

(a + b - c) * d – (e + f)

postfix expression

Infix to postfix conversion

stack



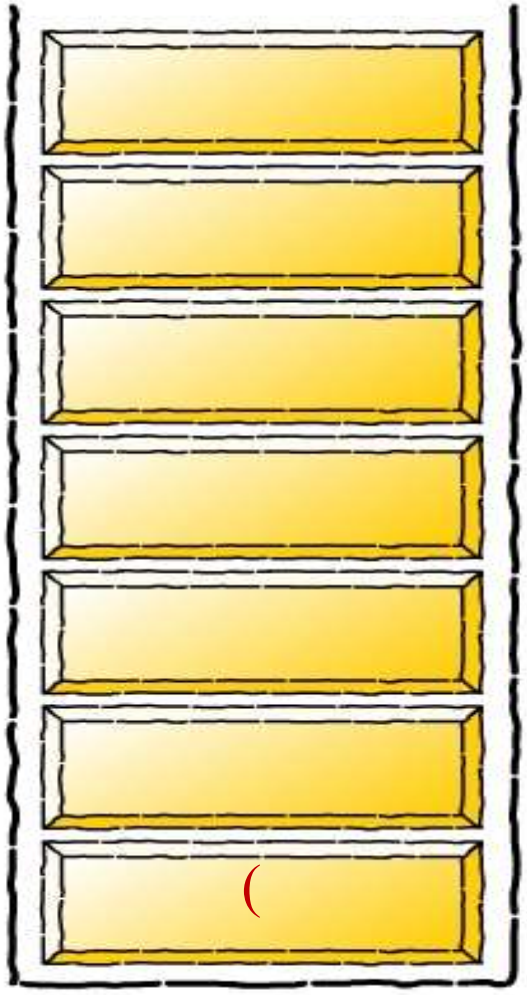
infix expression

a + b - c) * d – (e + f)

postfix expression

Infix to postfix conversion

stack



infix expression

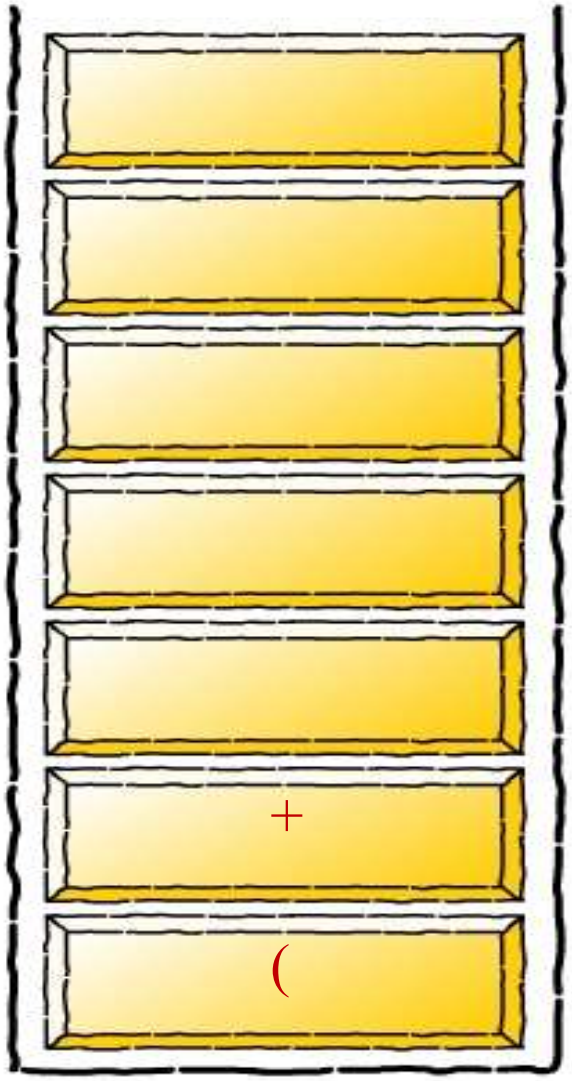
+ b - c) * d – (e + f)

postfix expression

a

Infix to postfix conversion

stack



infix expression

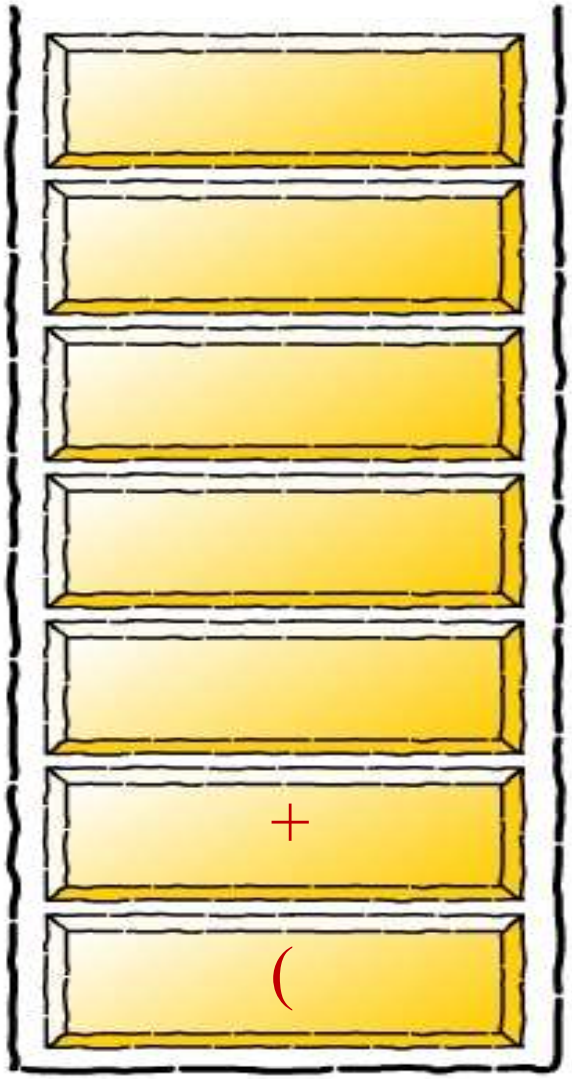
b - c) * d – (e + f)

postfix expression

a

Infix to postfix conversion

stack



infix expression

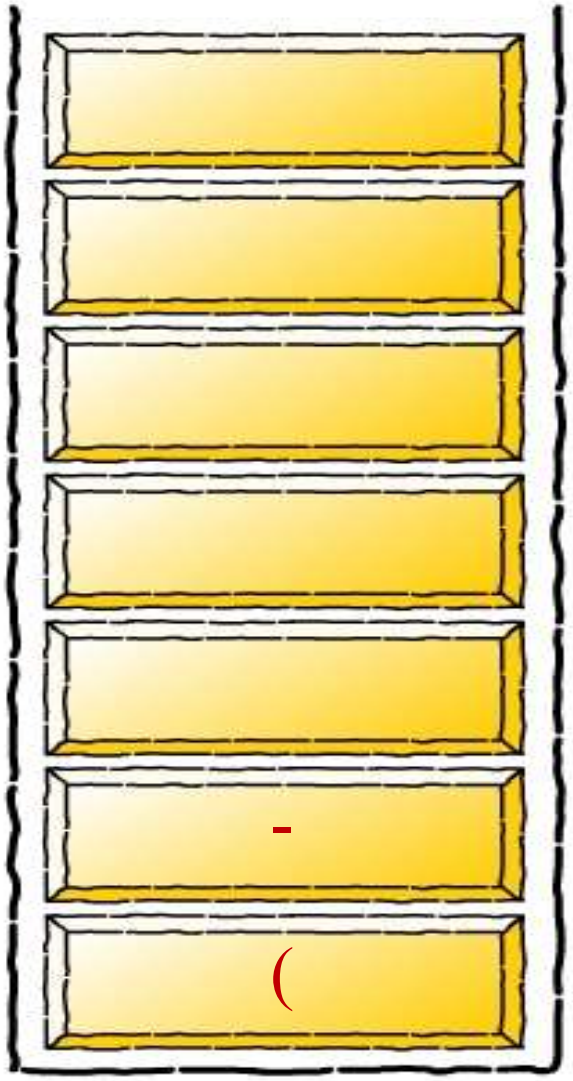
- c) * d – (e + f)

postfix expression

a b

Infix to postfix conversion

stack



infix expression

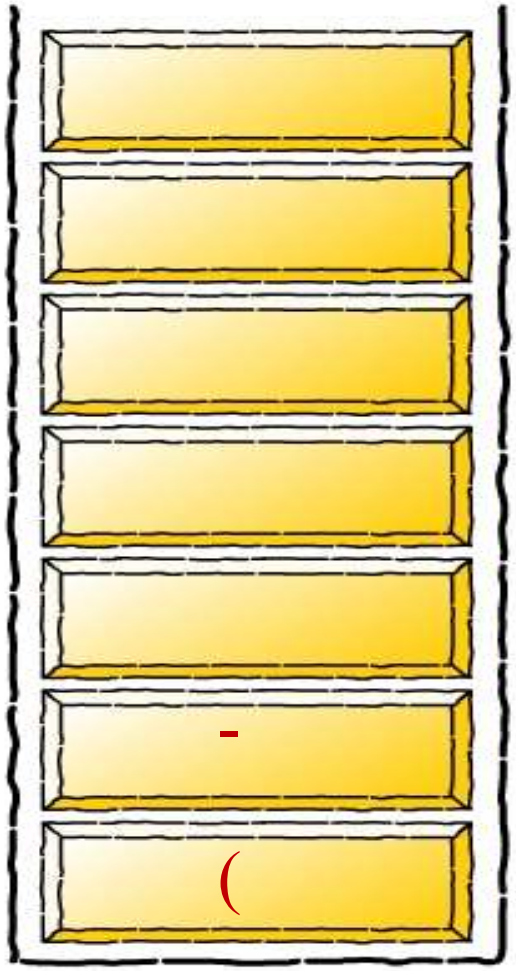
c) * d – (e + f)

postfix expression

a b +

Infix to postfix conversion

stack



infix expression

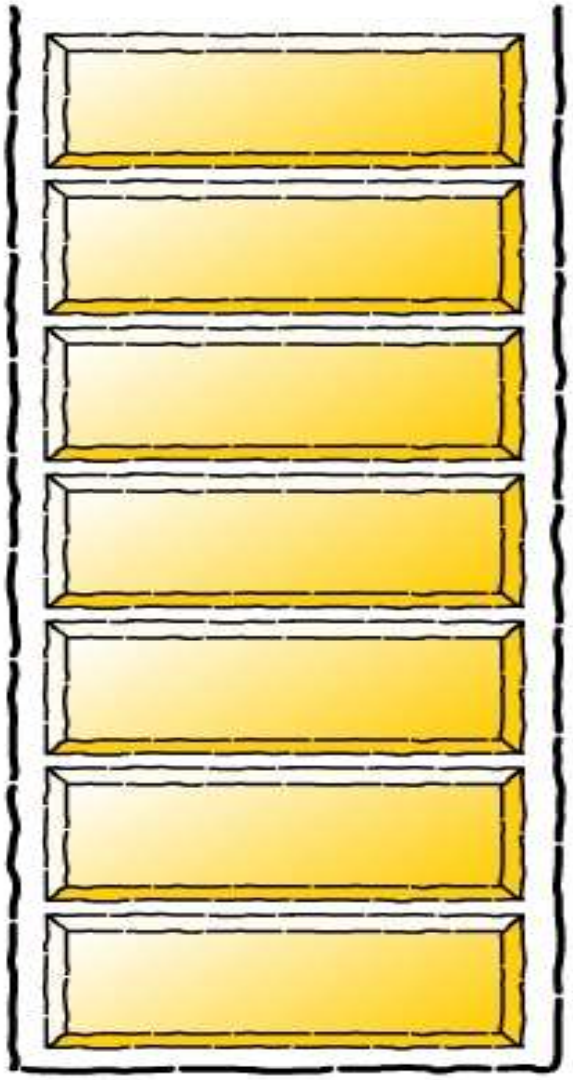
) * d – (e + f)

postfix expression

a b + c

Infix to postfix conversion

stack



infix expression

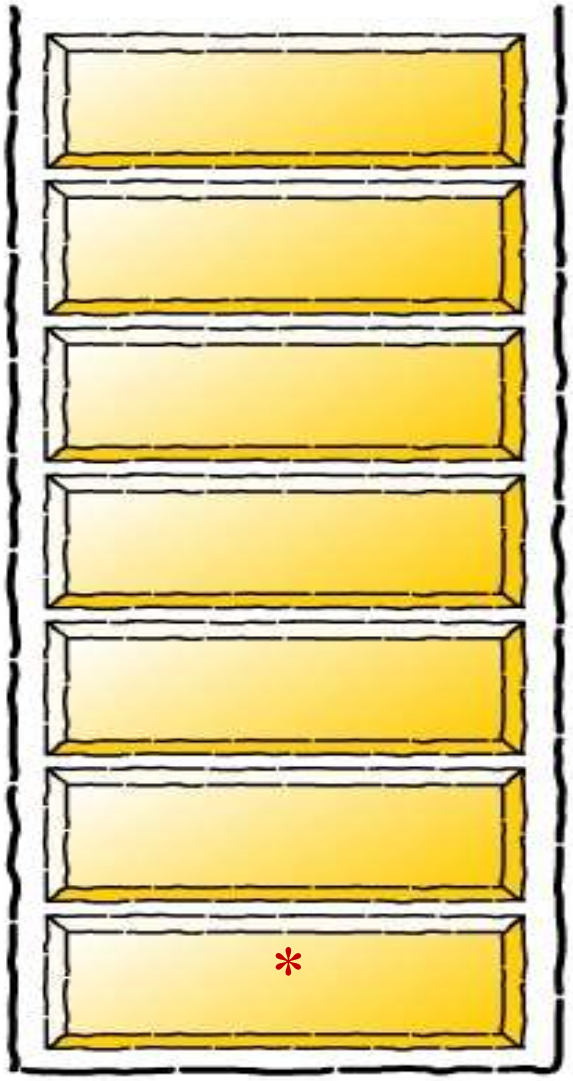
* d – (e + f)

postfix expression

a b + c -

Infix to postfix conversion

stack



infix expression

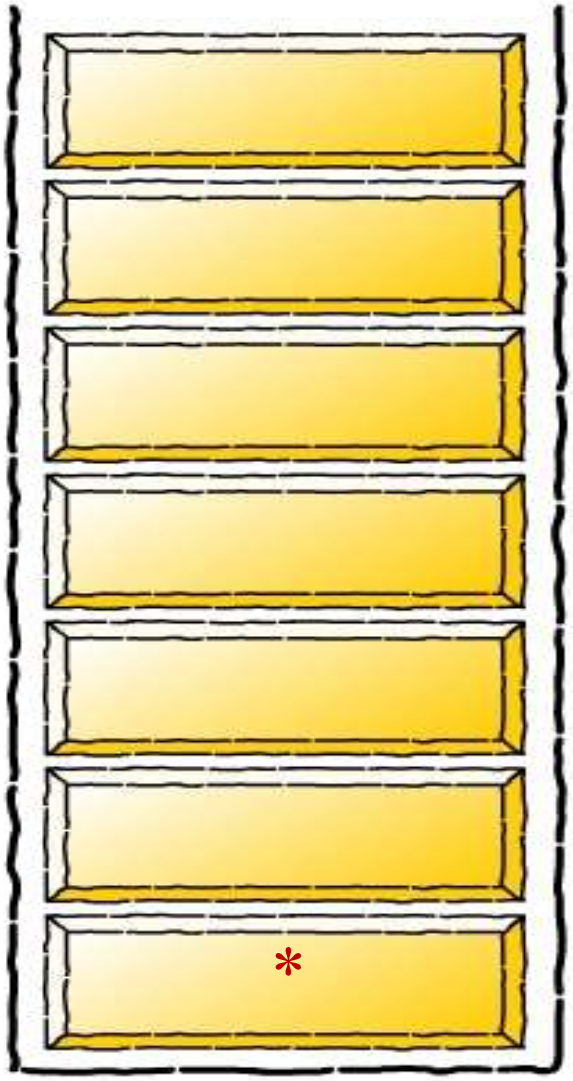
d – (e + f)

postfix expression

a b + c -

Infix to postfix conversion

stack



infix expression

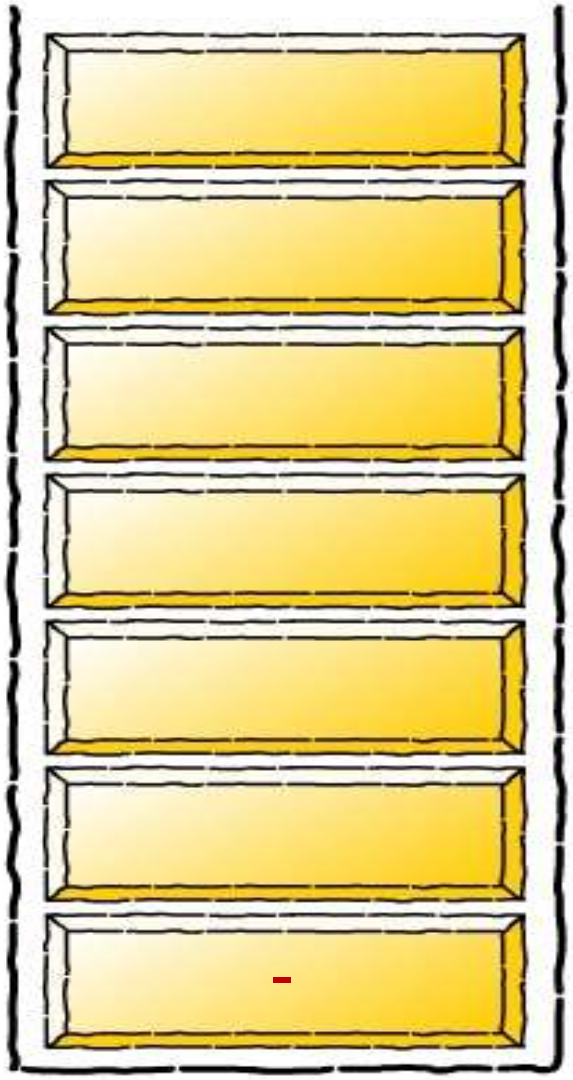
– (e + f)

postfix expression

a b + c - d

Infix to postfix conversion

stack



infix expression

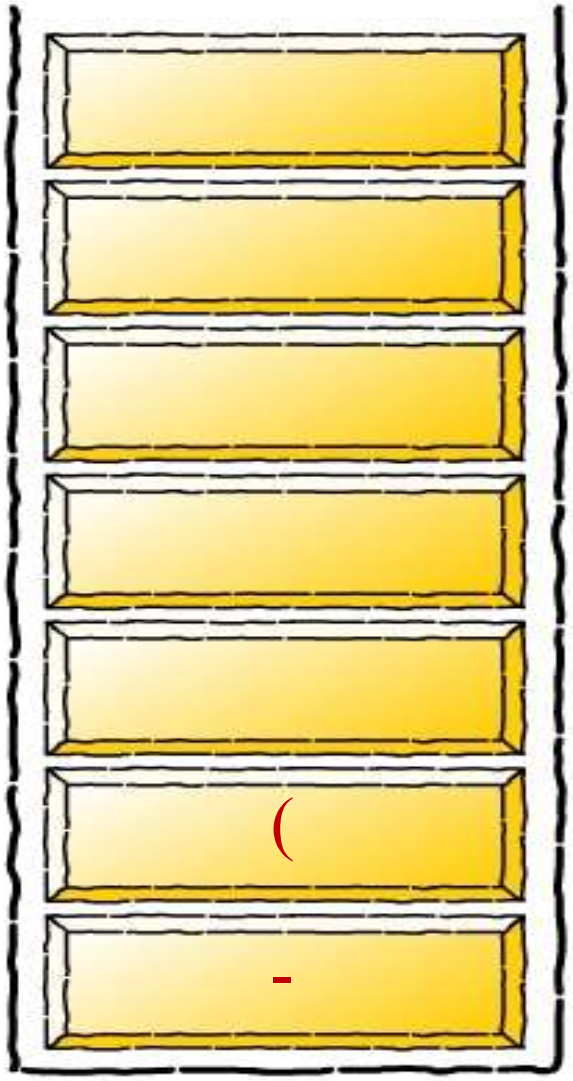
(e + f)

postfix expression

a b + c – d *

Infix to postfix conversion

stack



infix expression

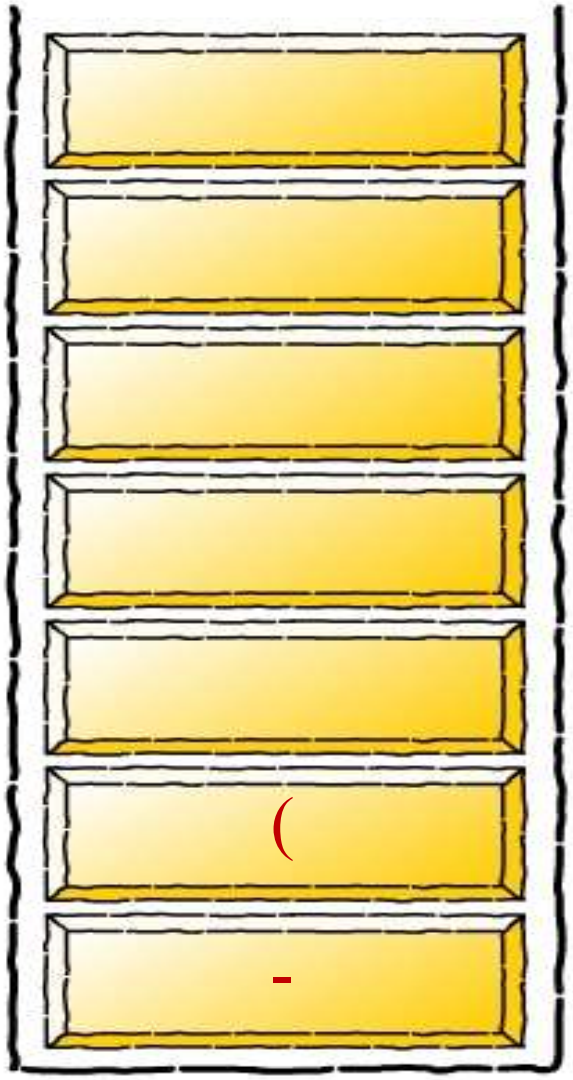
e + f)

postfix expression

a b + c – d *

Infix to postfix conversion

stack



infix expression

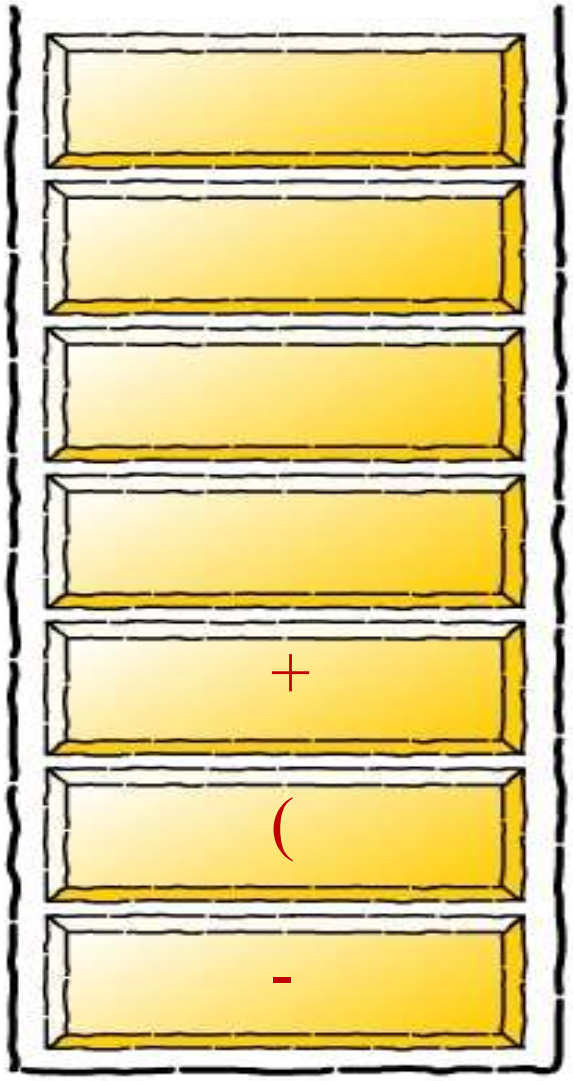
+ f)

postfix expression

a b + c – d * e

Infix to postfix conversion

stack



infix expression

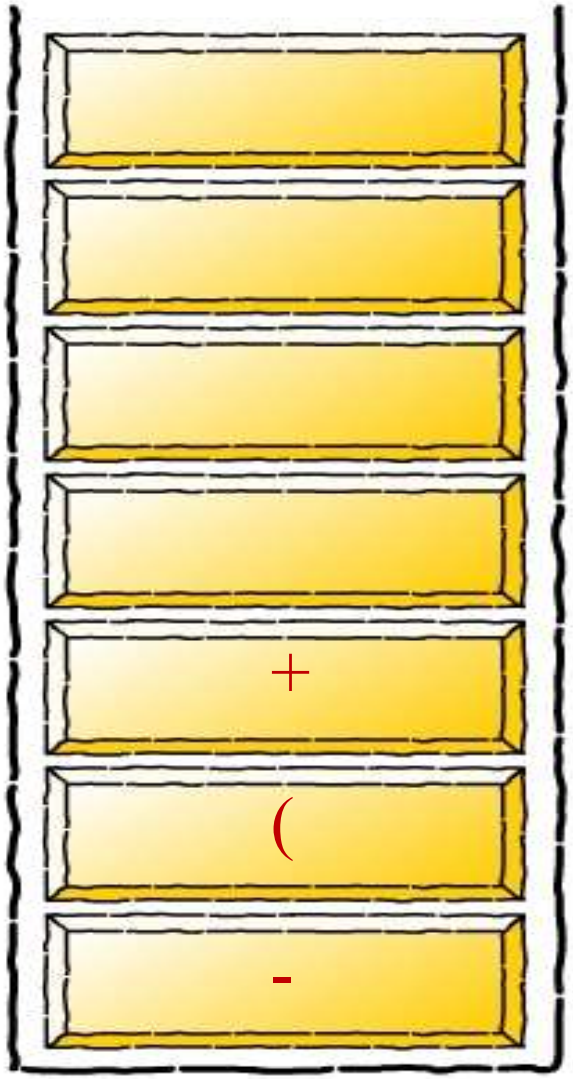
f)

postfix expression

a b + c – d * e

Infix to postfix conversion

stack



infix expression

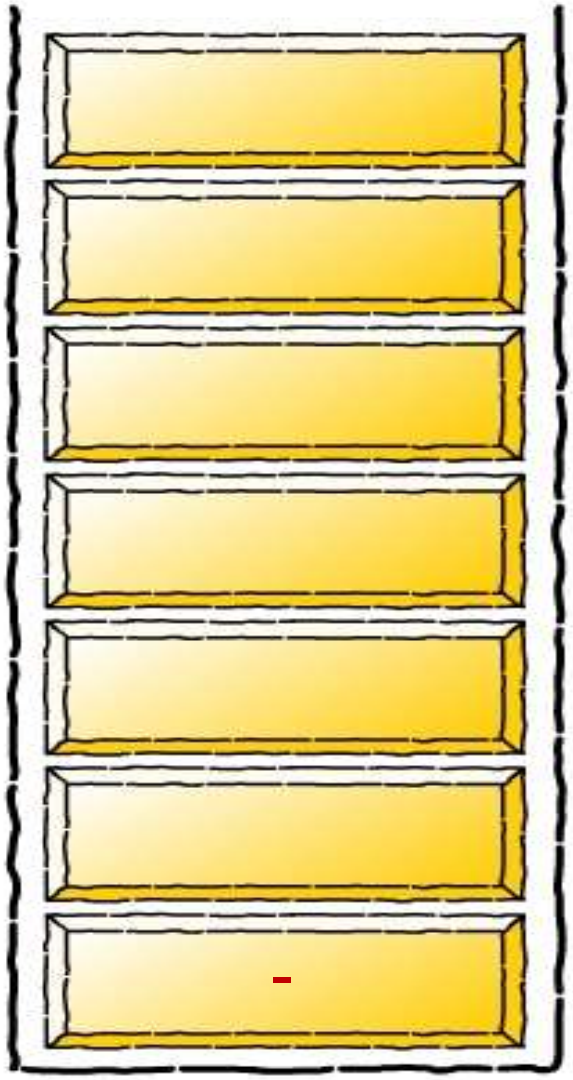
)

postfix expression

a b + c – d * e f

Infix to postfix conversion

stack



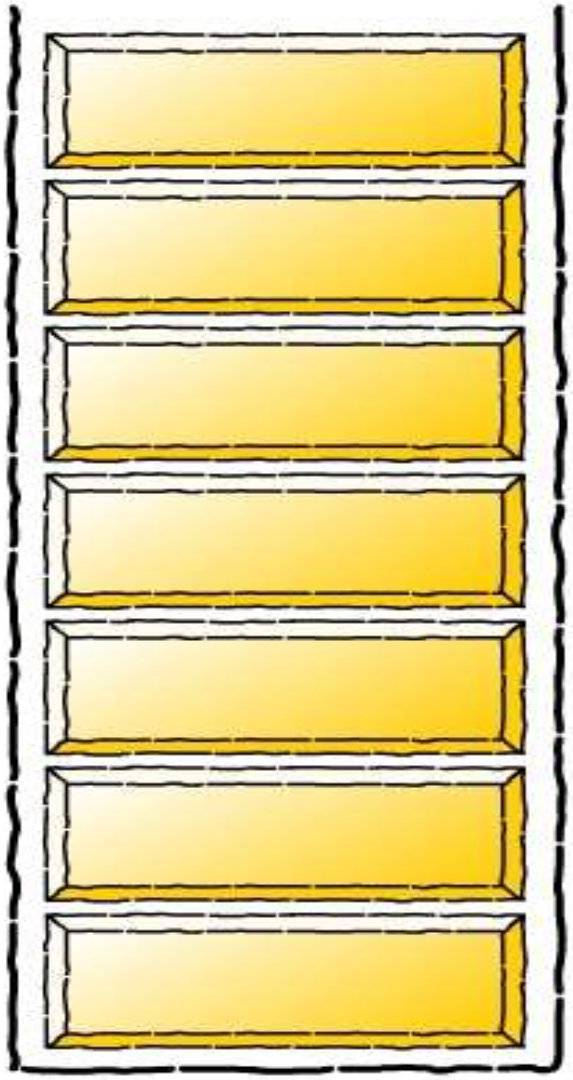
infix expression

postfix expression

a b + c – d * e f +

Infix to postfix conversion

stack



infix expression

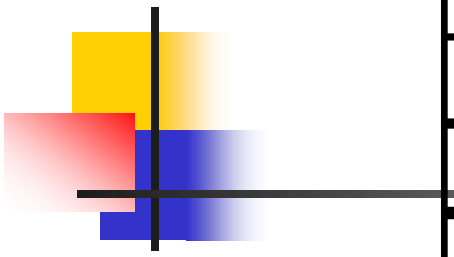
postfix expression

a b + c – d * e f + -

Algorithm of Infix to Postfix

```
infixToPostfix(infixexpr):
    postfixList = []
    tokenList = infixexpr
    for token in tokenList:
        if token is an operand
            append(token) to postfixList
        elif token == '(':
            push(token)
        elif token == ')':
            topToken = pop()
            while topToken != '(':
                append(topToken) to postfixList
                topToken = pop()
        else
         while (!Empty()) and  (prec[top()] >= prec[token])
            append(pop()) to postfixList
         push(token)
    while (!isEmpty())
            append(pop()) to postfixList
    return (postfixList)
```

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*+

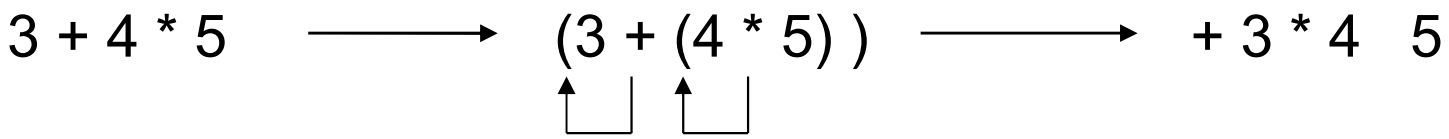
Postfix Expression is 23*21-/53*+

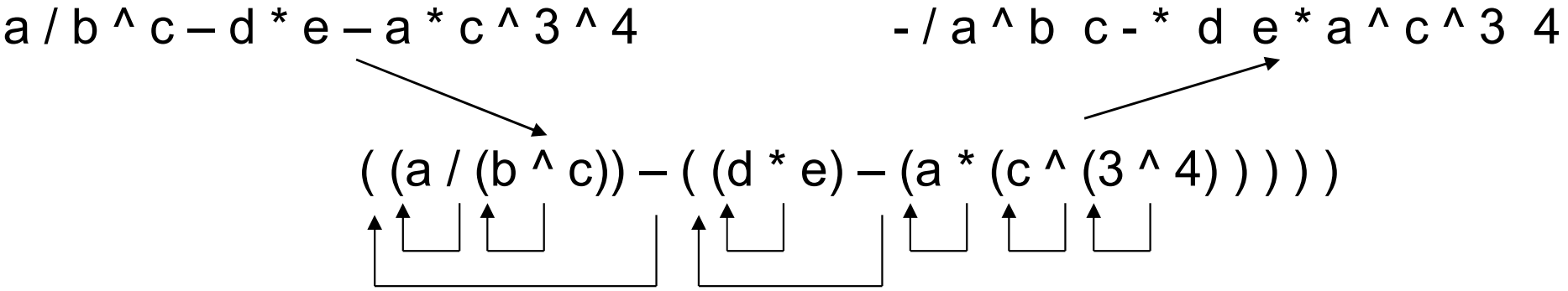
Examples of Prefix Expressions

Infix	Prefix
A+B	+AB
A+B-C	-+ABC
(A+B)*(C-D)	*+AB-CD
A^B*C-D+E/F/(G+H)	+-*^ABCD//EF+GH
((A+B)*C-(D-E))^(F+G)	^-*+ABC-DE+FG
A-B/(C*D^E)	-A/B*C^DE

Infix to Prefix conversion (Intuitive Algorithm)

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





Algorithm of Infix to Prefix

Input an infix expression in a string ‘infix’.
Reverse the string ‘infix’
Create an empty stack and also create an empty list for prefix expression
while(!end of string(infix))
    ch= a character from ‘infix’ string
    if(ch == ')')
        push(ch)
    if(ch == '(')
        while(top() != ')')
            append(top()) to prefixList
            pop()
    if(ch is an operand)
        append(ch) to prefixList
    if(ch is an operator)
        while(!empty() && prec (top()) > prec(ch))
            append(top()) to prefixList
            pop()
        push(ch);
while(!empty())
    append(top()) to prefixList
    pop()
reverse the ‘prefix’ string

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.

Algorithm to Evaluate Postfix Expression

scan each character ch in the postfix expression

if ch is an operator ⊙, then

    a = pop first element from stack

    b = pop second element from the stack

    res = b ⊙ a

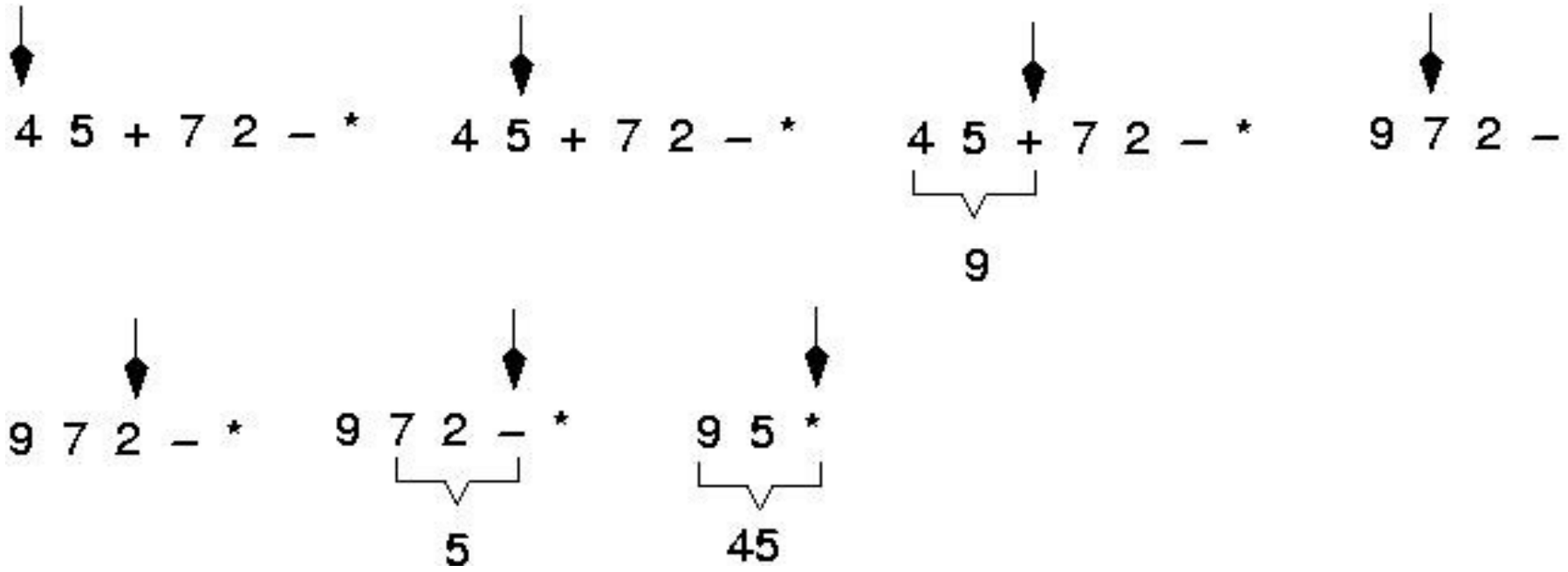
    push res into the stack

else if *ch* is an operand

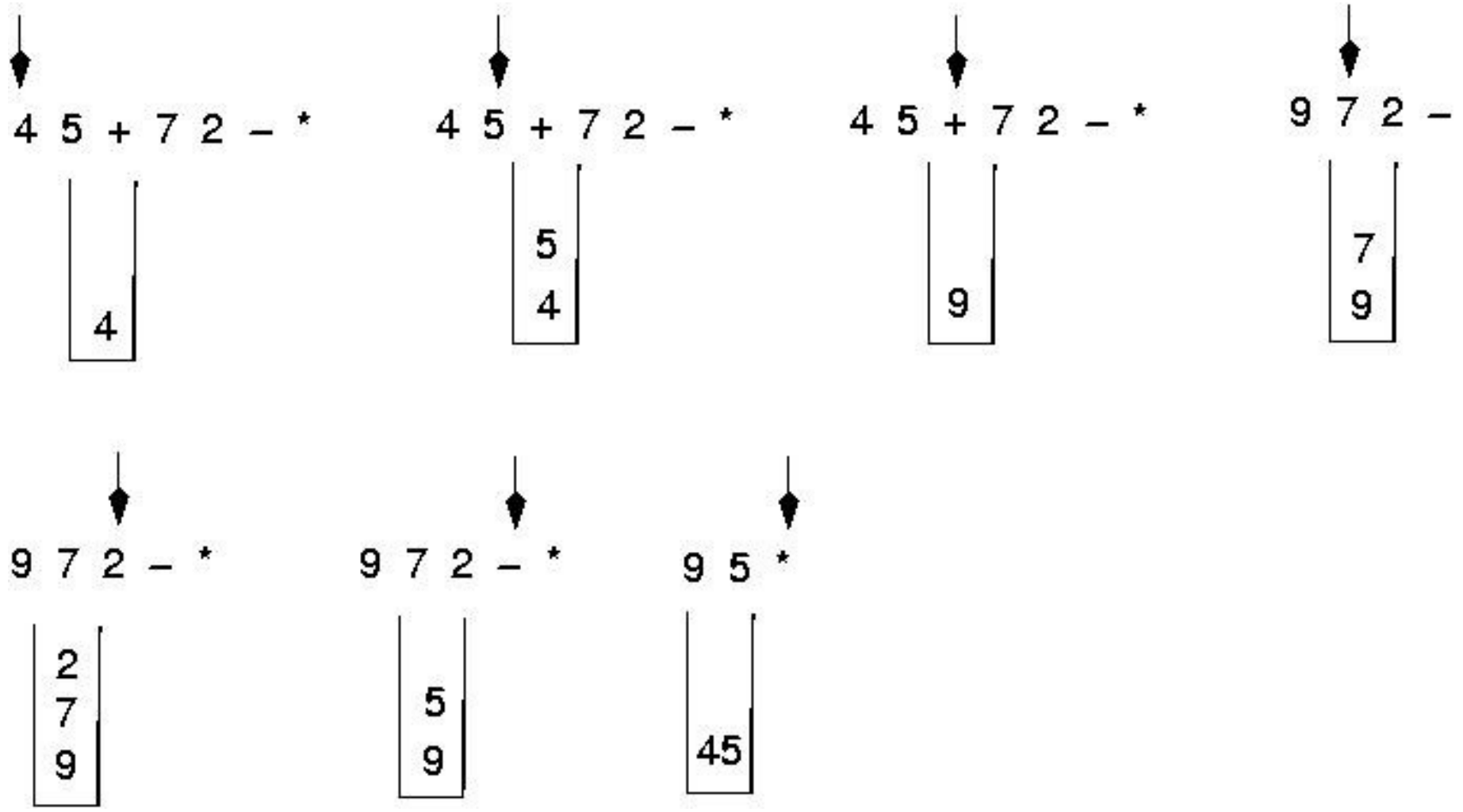
    push *ch* into the stack

return top(stack)

Example: postfix expressions

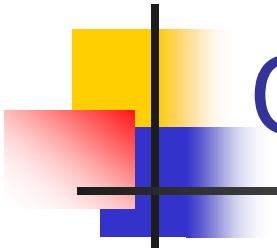


Postfix expressions: Algorithm using stacks



Algorithm for evaluating a postfix expression

```
while more input items exist {
    If symb is an operand
        then push (symb)
    else  {   //symbol is an operator
        opnd2=pop()
        opnd1=pop()
        val = opnd1 symb opnd2
        push(val)
    }
}
result = pop ()
```



Question

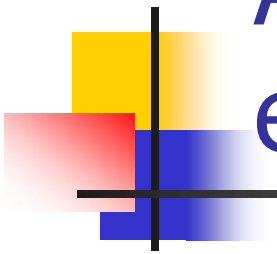
Evaluate the following expression in postfix : 623+-382/+*2^3+

A. 49
B. 51
C. 52
D. 7
E. None of these

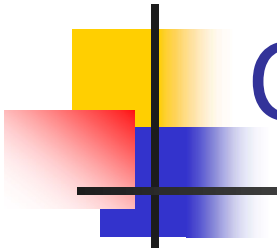
Evaluate: 623+-382/+*2^3+

Symbol	opnd1	opnd2	value	opndstk
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
^	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Algorithm for evaluating a prefix expression



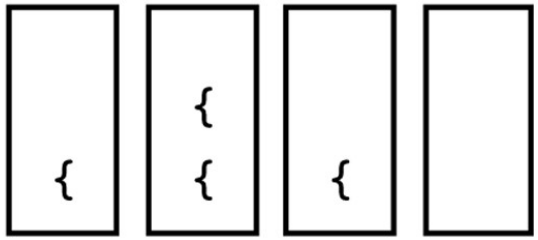
```
Reverse the prefix Expression
while more input items exist {
    If symb is an operand
        then push (symb)
    else  {   //symbol is an operator
        opnd1=pop()
        opnd2=pop()
        val = opnd1 symb opnd2
        push(val)
    }
}
result = pop ()
```



Checking for Balanced Braces

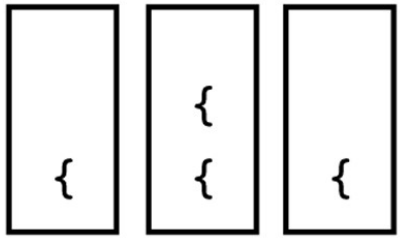
- Requirements for balanced braces
  - Each time you encounter a “}”, it matches an already encountered “{”
  - When you reach the end of the string, you have matched each “{”

Checking for Balanced Braces

Input string	Stack as algorithm executes	
	1.	2.	3.	4.
`{a{b}c}`				

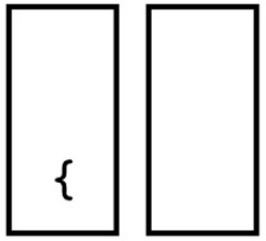
1. push "{"
2. push "{"
3. pop
4. pop

Stack empty $\Longrightarrow$ balanced

Input string	Stack as algorithm executes		
	1.	2.	3.
`{a{bc}`			

1. push "{"
2. push "{"
3. pop

Stack not empty $\Longrightarrow$ not balanced

Input string	Stack as algorithm executes	
	1.	2.
`{ab}c}`		

1. push "{"
2. pop

Stack empty when last "}" encountered $\Longrightarrow$ not balanced

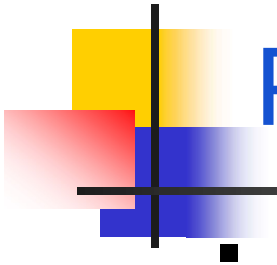
Checking for Balanced Braces

```
createStack()
balancedSoFar = true
i = 0
while ( balancedSoFar  and  i < lengthofaString )  {
      ch = character at position i in aString
      ++i
      if ( ch is '{' )      // push an open brace
            push( '{' )
      else if ( ch is '}' )            // close brace
            if ( !isEmpty() )
                  pop()       // pop a matching open brace
            else                      // no matching open brace
                  balancedSoFar = false
                              // ignore all characters other than braces
}
if ( balancedSoFar  and  isEmpty() )
      String has balanced braces
else
      String does not have balanced braces
```

Find Maximum depth of brackets

```
int maxDepth(char str[]) {
    int curr_max = 0; // current count
    int max = 0;        // overall maximum count
    int n = strlen(str);
    for (int i = 0; i< n; i++)  {
        if (str[i] == '(') {
            curr_max++;
            if (curr_max> max)
                max = curr_max;
        }
        else if (str[i] == ')')  {
            if (curr_max>0)
                curr_max--;
            else
                return -1;
        }
    }
    // finally check for unbalanced string
    if (current_max != 0)
        return -1;
    return max;
}

int main() {
    char str[] = "( ((X)) (((Y))) )";
    printf("%d\n", maxDepth(str));
    return 0;
}
```



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

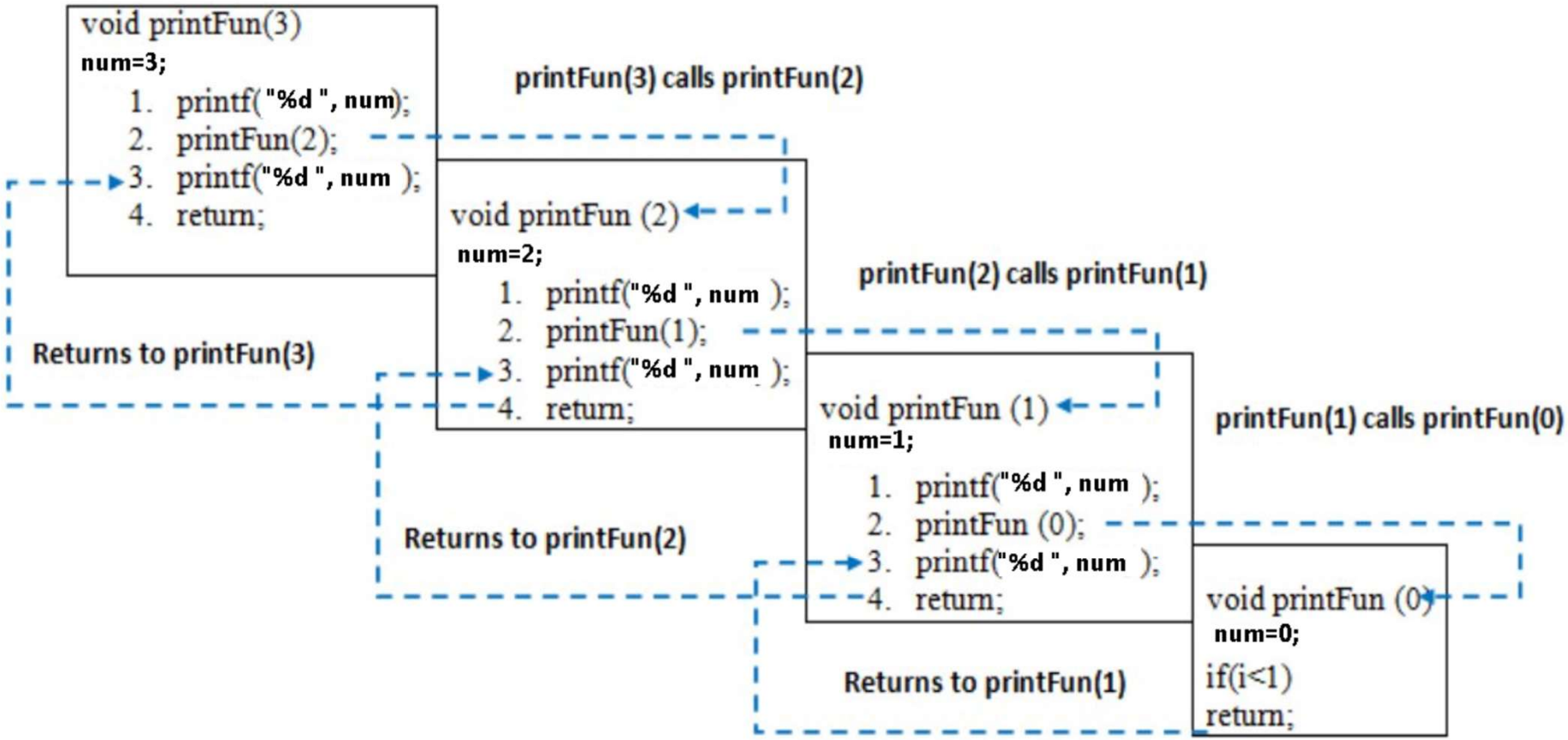
```
void printFun(int num) {
    if (num < 1)
        return;
    else {
        printf("%d ", num);
        printFun(num - 1); // statement 2
        printf("%d ", num);
        return;
    }
}
```

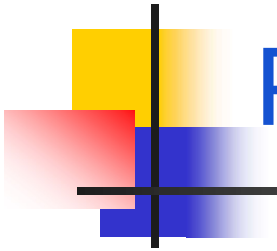
```
int main() {
    int n = 3;
    printFun(n);
}
```

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--;
}
```

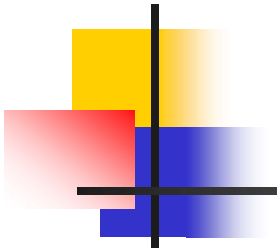
Program: Linear implementation of stack Using Structure

```
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;
}
```



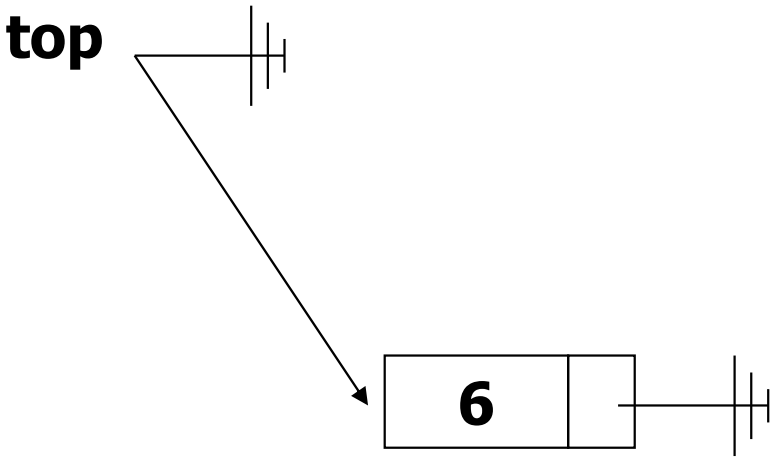
Stack: Linked List Implementation

- 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

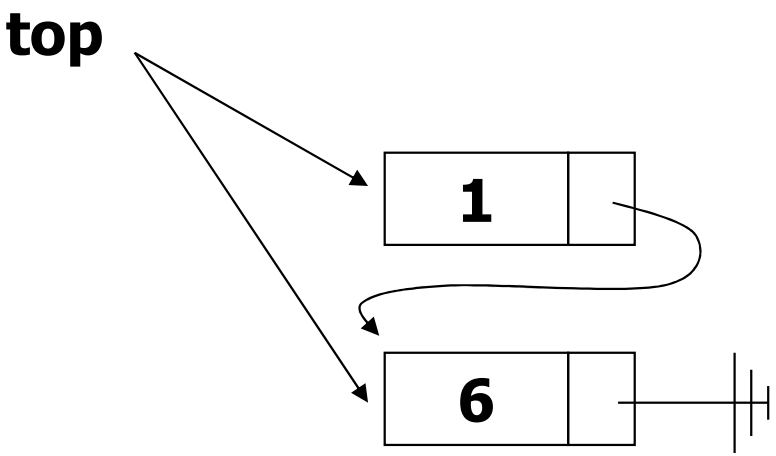
```
Stack s;
s.push(6);
```



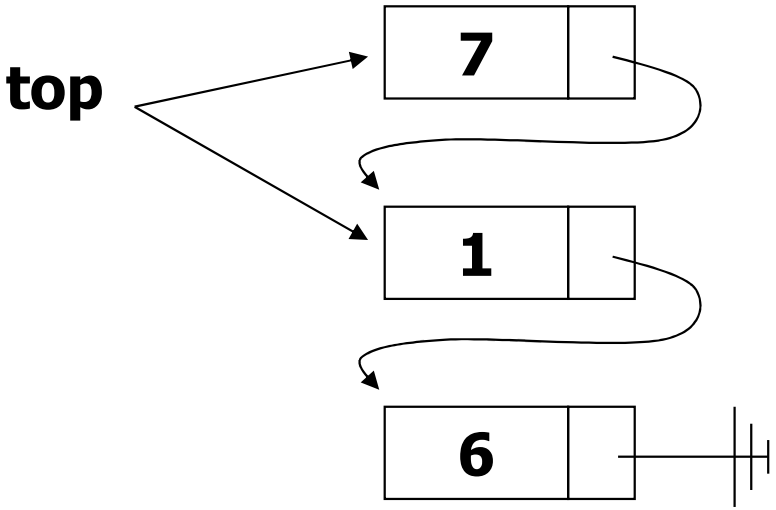
Stack: Example

C Code

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



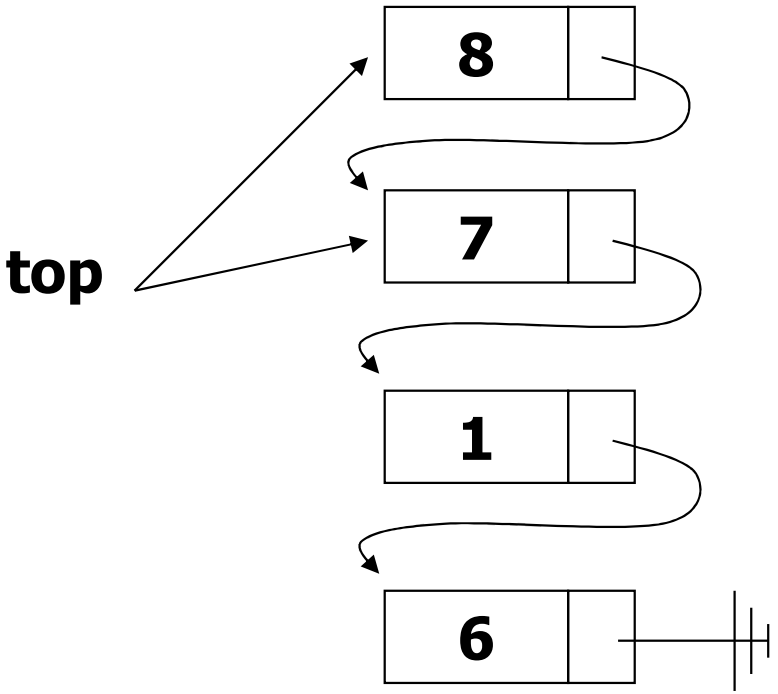
Stack: Example



C Code

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

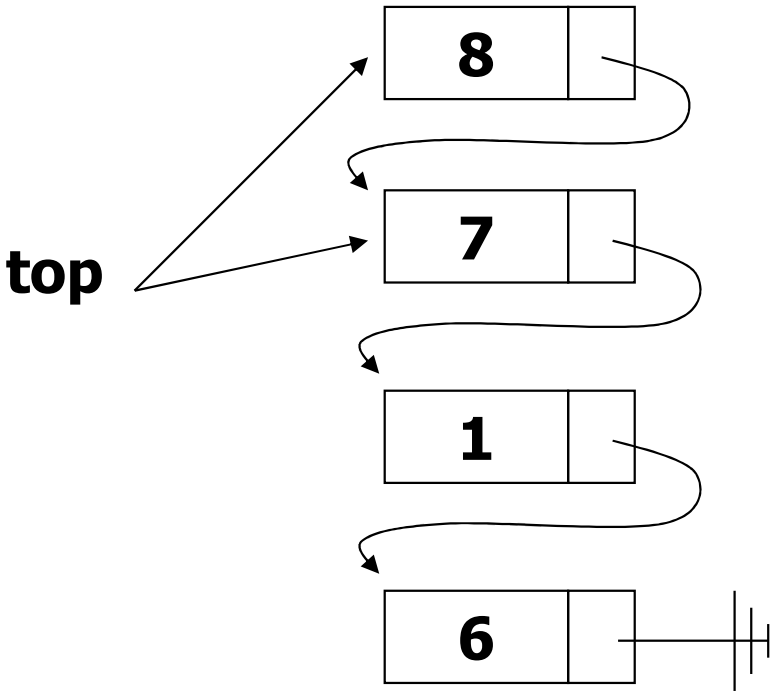
Stack: Example



C Code

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

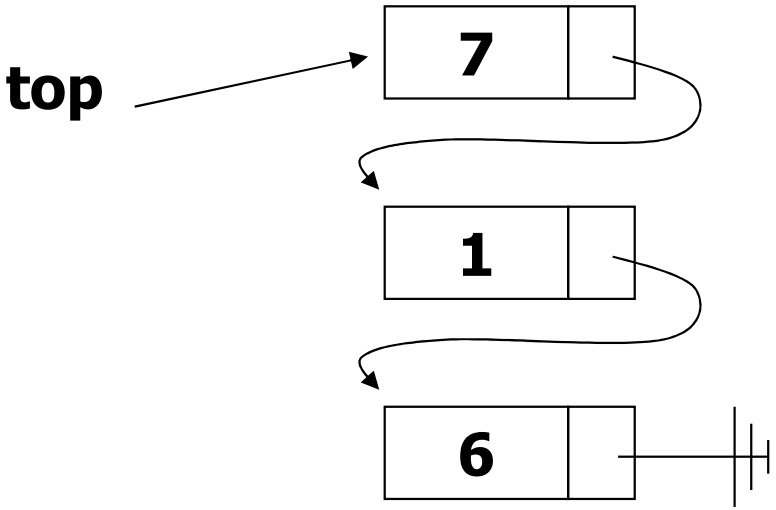
Stack: Example



C Code

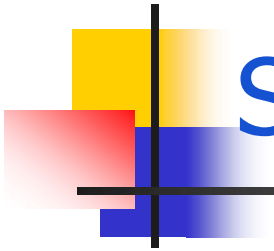
```
Stack s;
s.push(6);
s.push(1);
s.push(7);
s.push(8);
s.pop();
```

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;
```

Stack Implementation: createStack, isEmpty

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


int isEmpty(Stack top) {
   if (top == NULL)
      return 1;
   else
      return 0;
}
```

Stack Implementation: push

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

Stack Implementation: pop

```
int Pop() {
        int item;
        stack *temp;
        if( top == NULL) {
                printf(" Empty Stack …");
                return -1;
        }
        else {
                temp=top;
                item=top->data;
                top=top->next;
                temp->next=NULL;
                free(temp);
                return(item);
        }
}
```

Stack Implementation: display

```
void display(Stack *top) {
        Stack *temp;
        temp = top;
        if (top == NULL) {
        printf("Empty stack …");
                return;
        }
        else {
        while (temp != NULL) {
                        printf("%d\n", temp->data);
                        temp = temp->next;
        }
        }
}
```