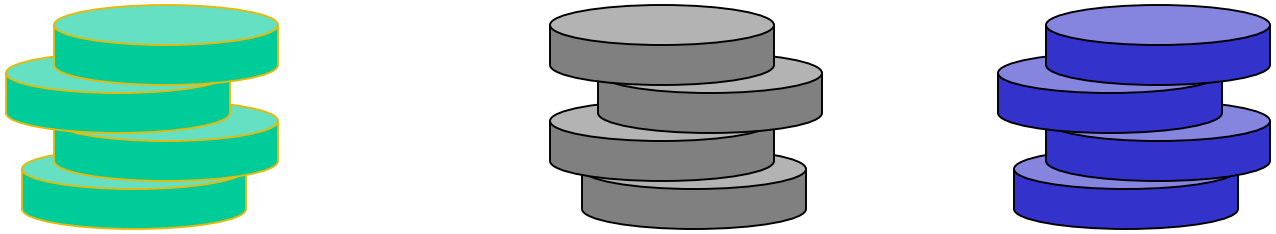
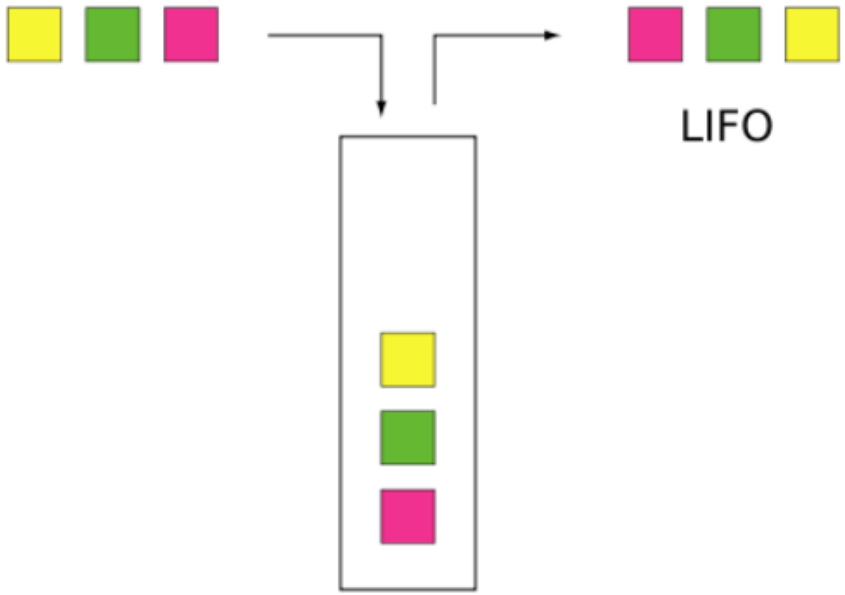
Stacks



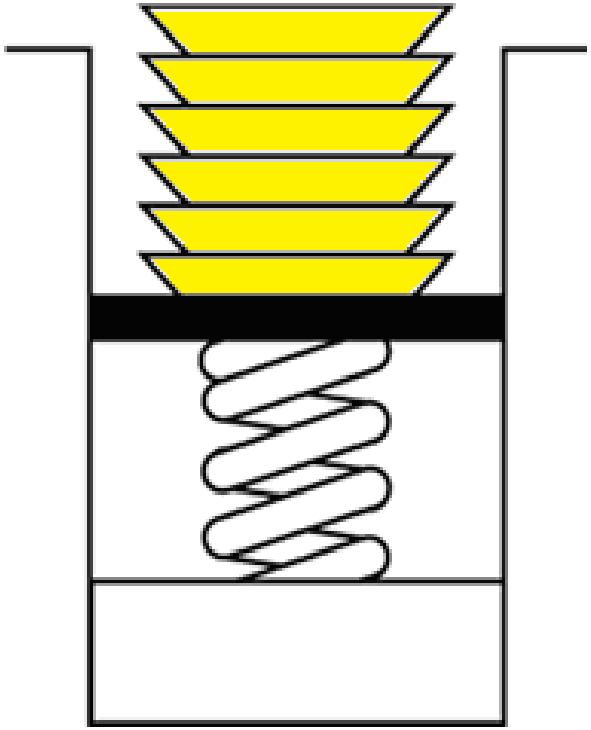
Gurpreet Singh Lehal
Punjabi University, Patiala

Stack

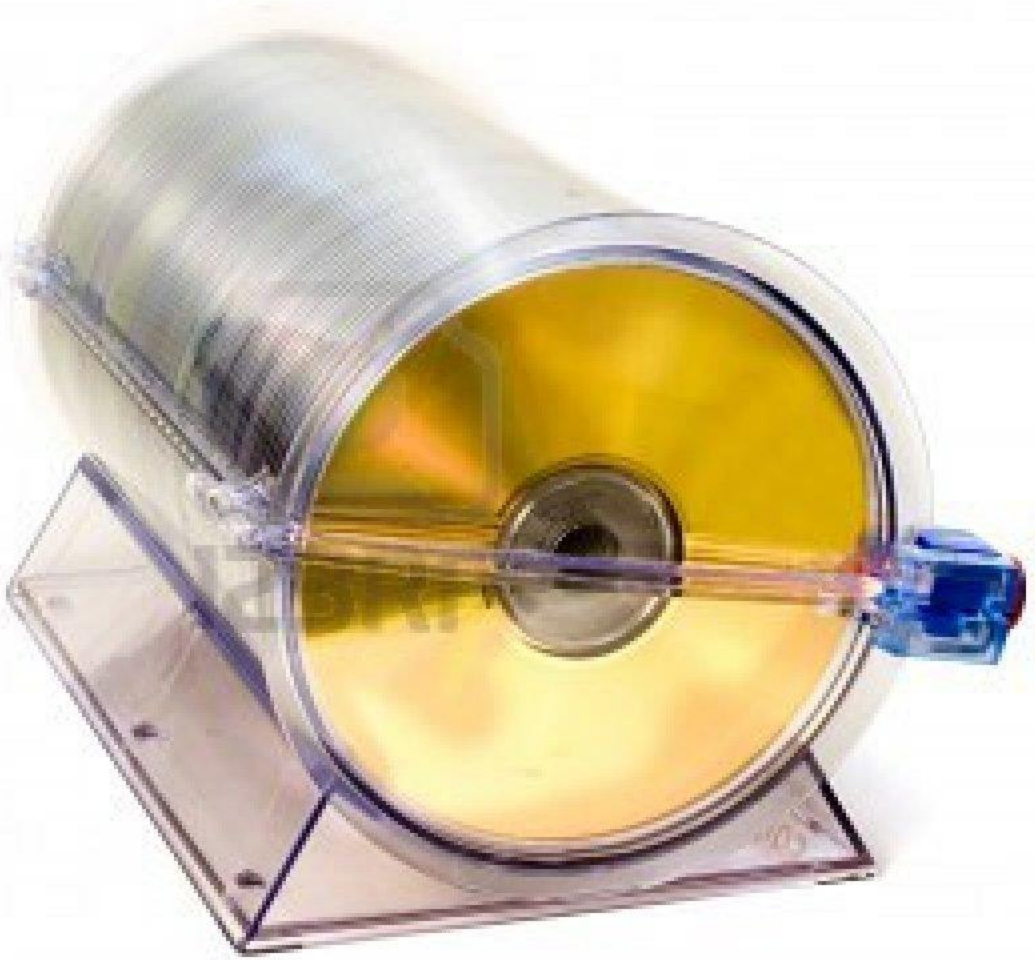
- A stack is a *limited access linear* list of homogeneous items in which access is allowed only at one point of the structure
- The last element to be inserted into the stack is the first to be removed. This strategy of removal and retrieval is referred as, LIFO(last



Stack of cafeteria plates



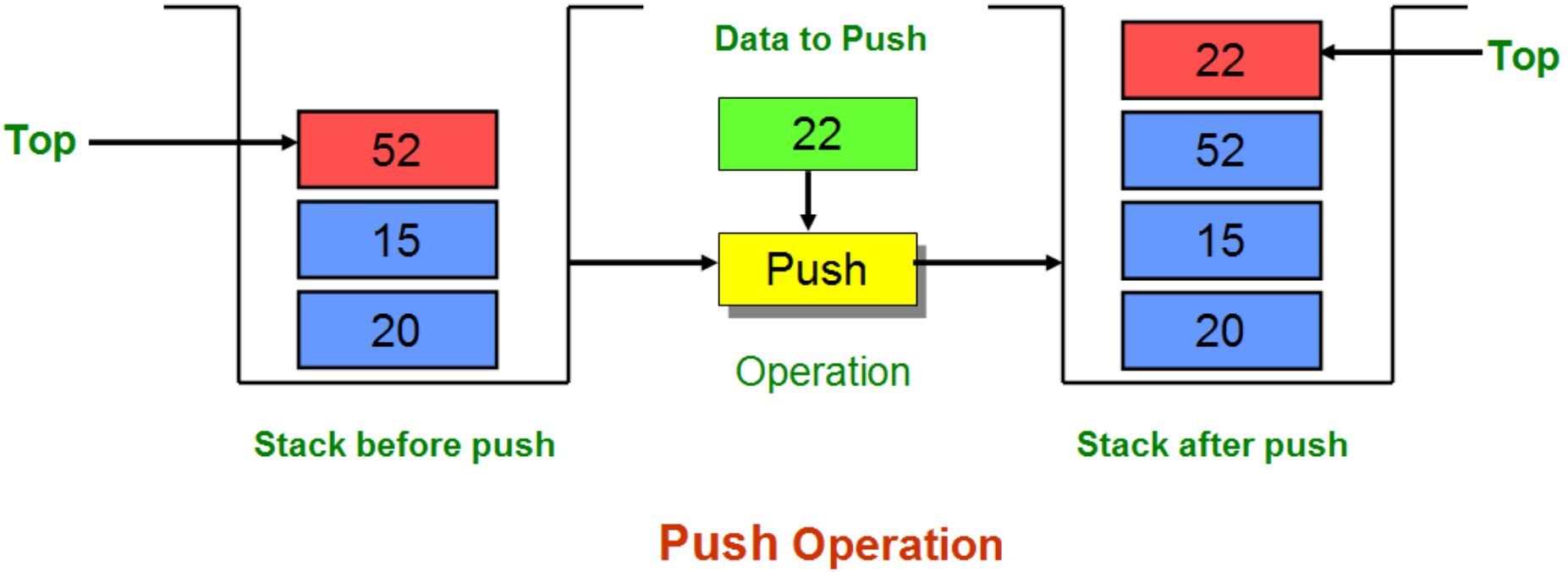
CDs in CD Container



Vegetable Stack

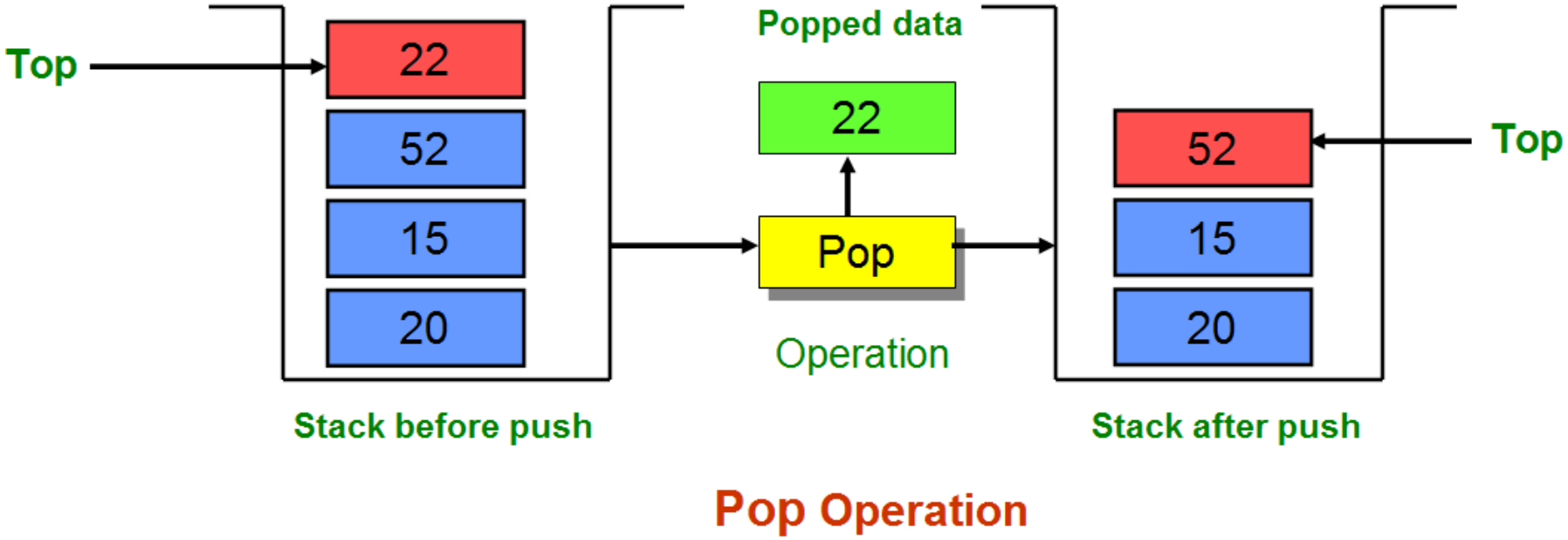
Shuttlecocks in their container

Push



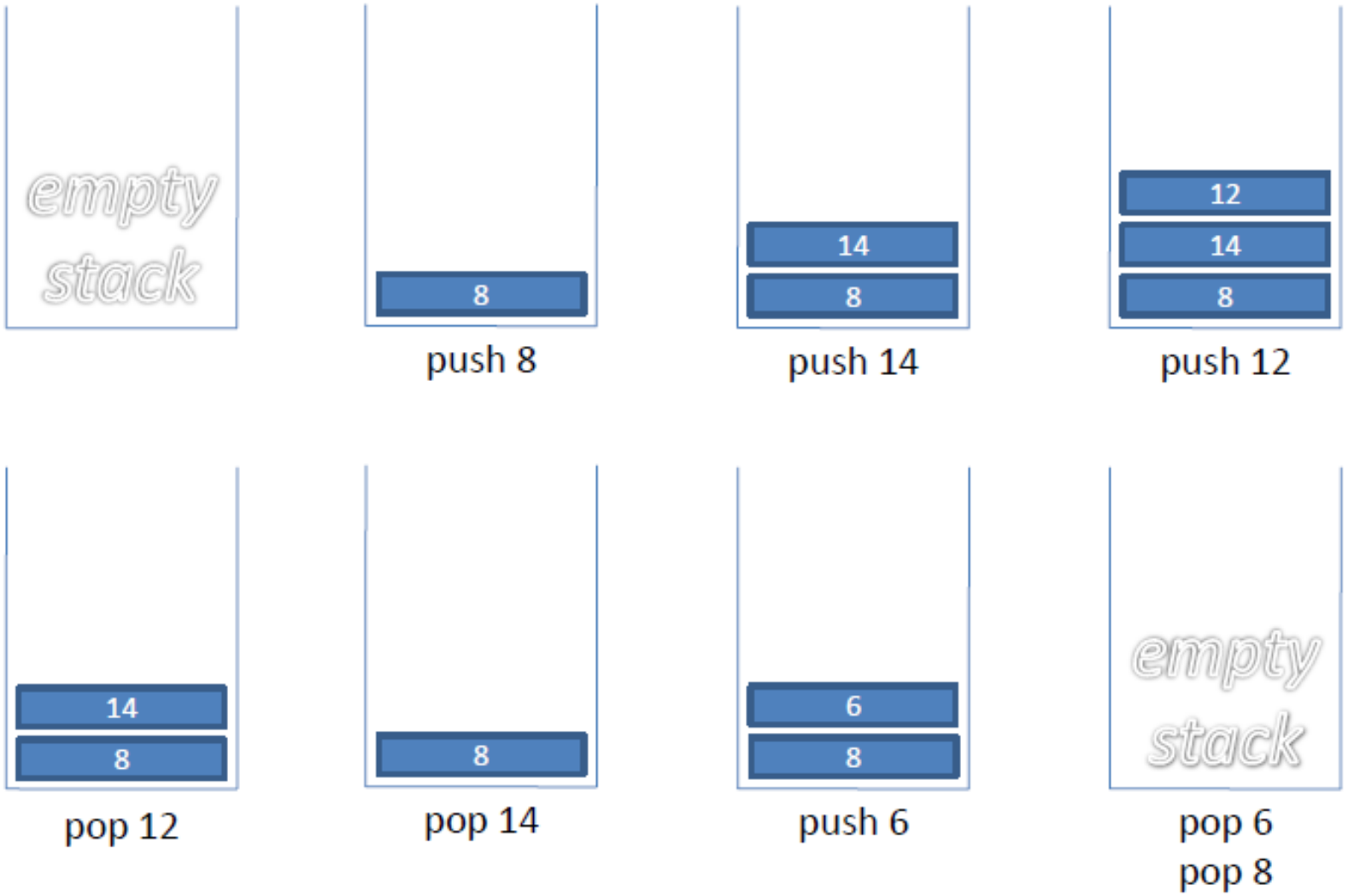
Push Operation

Pop



Pop Operation

A Sequence of Push and Pop operations



Array Implementation of Stack

```
typedef char StackItemType;
class Stack{
public:
    Stack(int size) {
      items = new StackItemType[size];
      MaxStack = size;
      top = -1;
   }
   ~Stack(){
      delete [] items;
   }
   bool isEmpty();
   bool isFull();
   bool push(StackItemType newItem);
   bool pop(StackItemType *stackTop);
private:
  StackItemType *items;
  int top, MaxStack;
 };
```

Array Implementation of Stack

```
bool Stack::isEmpty() {
   return top < 0;
}

bool Stack::isFull() {
   return top >= MaxStack-1;
}

bool Stack::push(StackItemType newItem){
   if (isFull())
      return false;
   else{
      items[++top] = newItem;
      return true;
   }
}
```

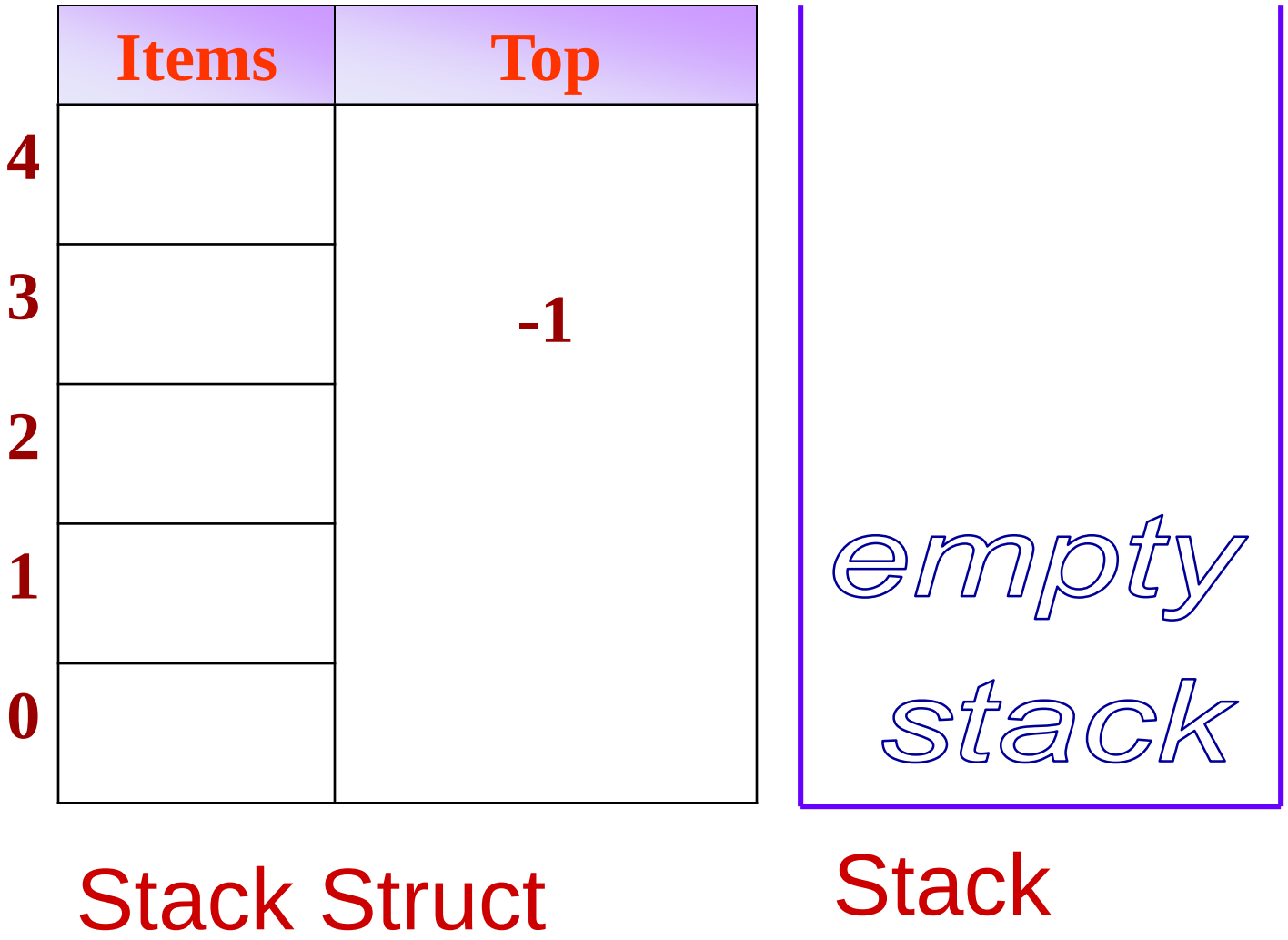
Array Implementation of Stack

```
bool Stack::pop(StackItemType *stackTop){
    if (isEmpty())
        return false;
    else {
         *stackTop = items[top--];
         return true;
    }
}
```

A Sample Program for Stacks

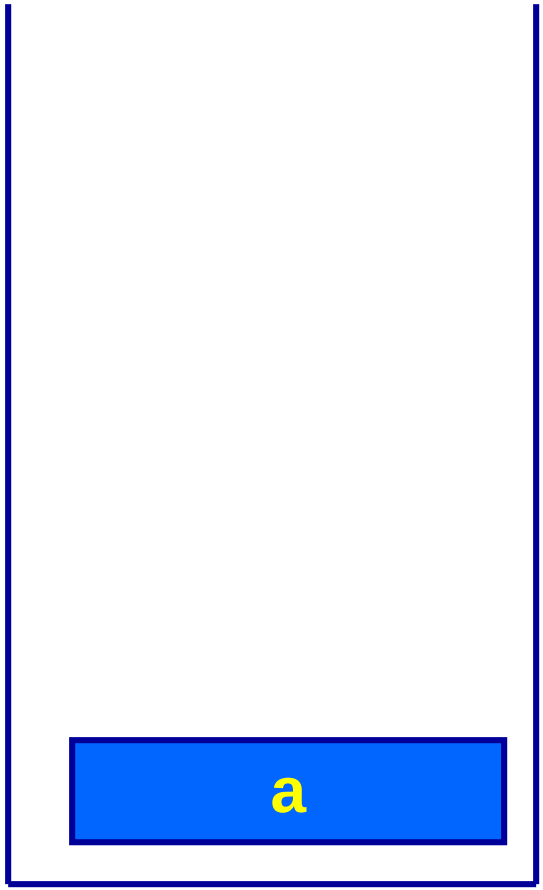
```
int main(){
        StackItemType c;
        Stack stack(5);
        stack.push('a');
        stack.push('b');
        stack.push('c');
        stack.pop(&c);
        printf("%c\n",c);
        stack.pop(&c);
        printf("%c\n",c);
        stack.push('d');
        stack.pop(&c);
        printf("%c\n",c);
        stack.pop(&c);
        printf("%c\n",c);
}
```

Stack stack(5)



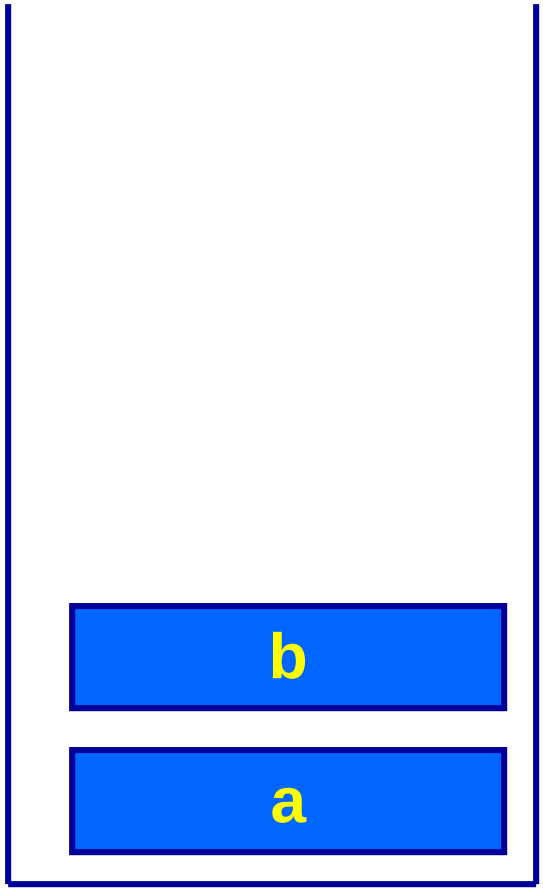
stack.push('a')

	Items	Top
4		0
3		
2		
1		
0	a	



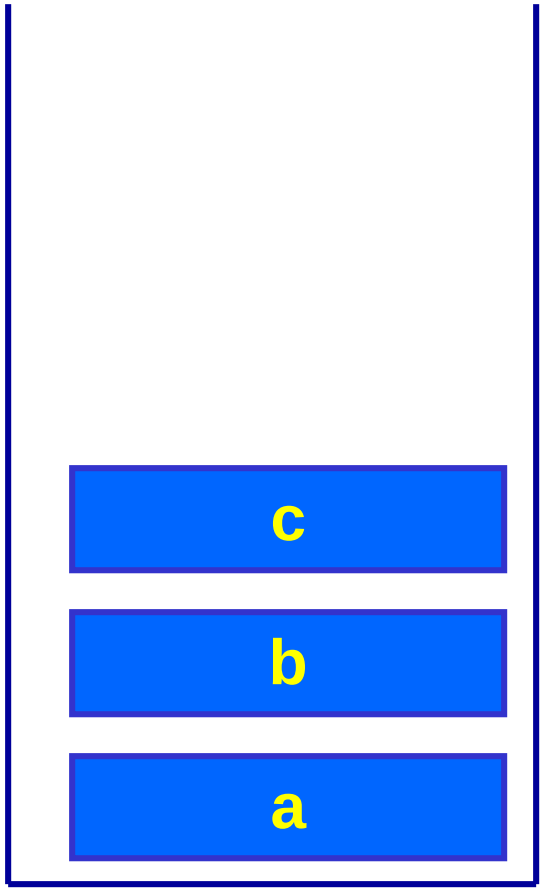
stack.push('b')

	Items	Top
4		1
3		
2		
1	b	
0	a	



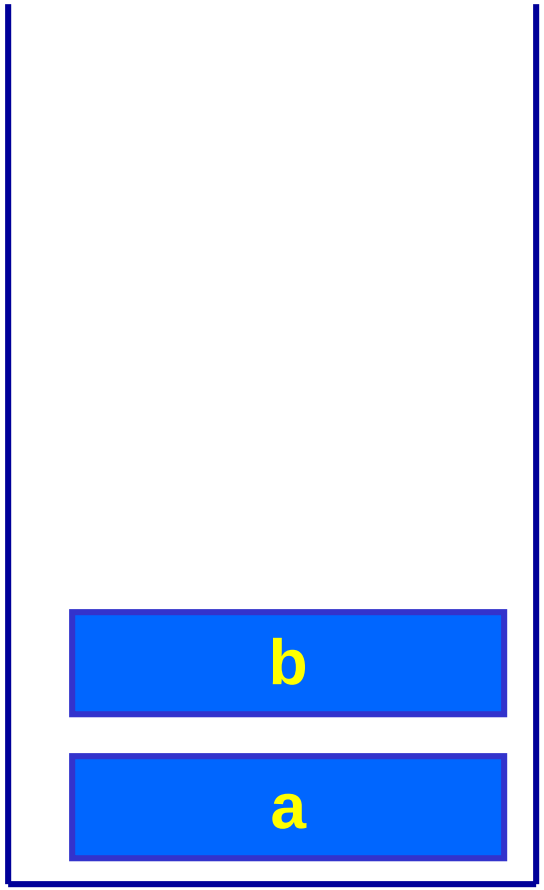
stack.push('c')

	Items	Top
4		2
3		
2	c	
1	b	
0	a	



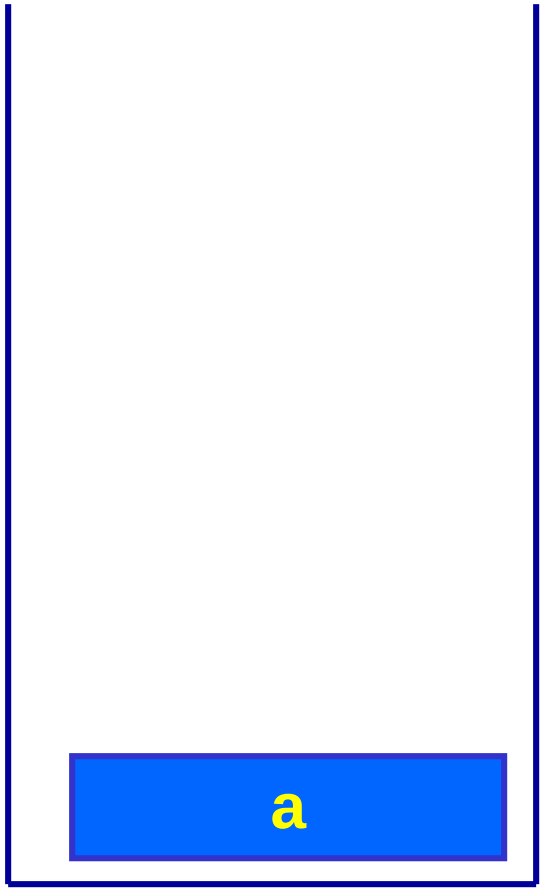
stack.pop(&c)->'c'

	Items	Top
4		1
3		
2	c	
1	b	
0	a	



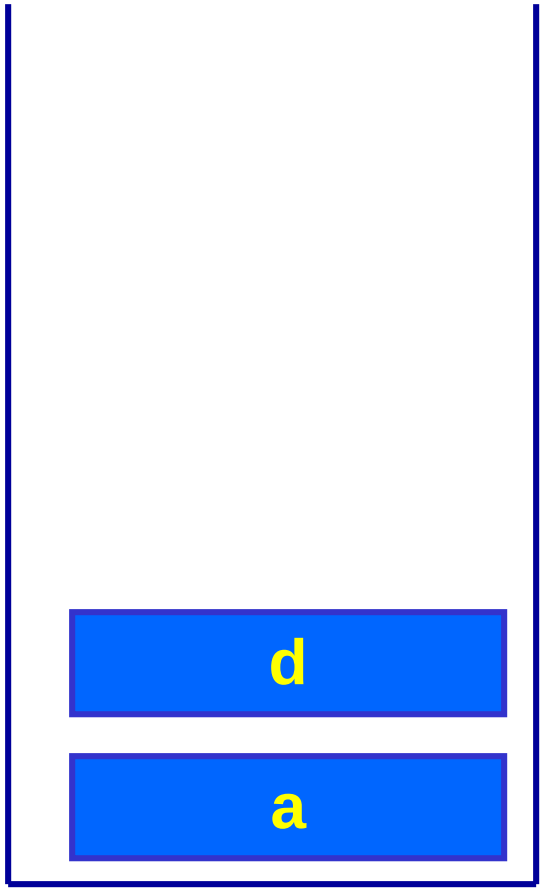
stack.pop(&c)->’b’

	Items	Top
4		0
3		
2	c	
1	b	
0	a	



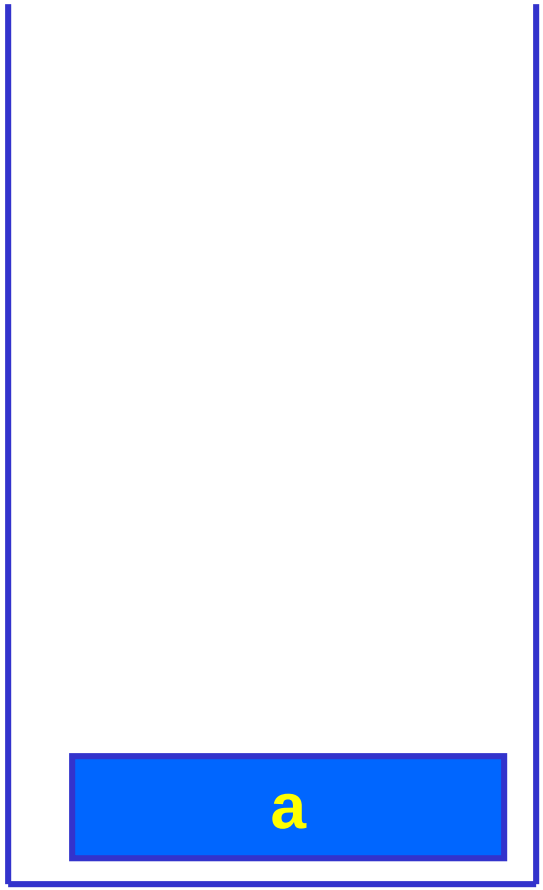
stack.push('d')

	Items	Top
4		1
3		
2	c	
1	d	
0	a	



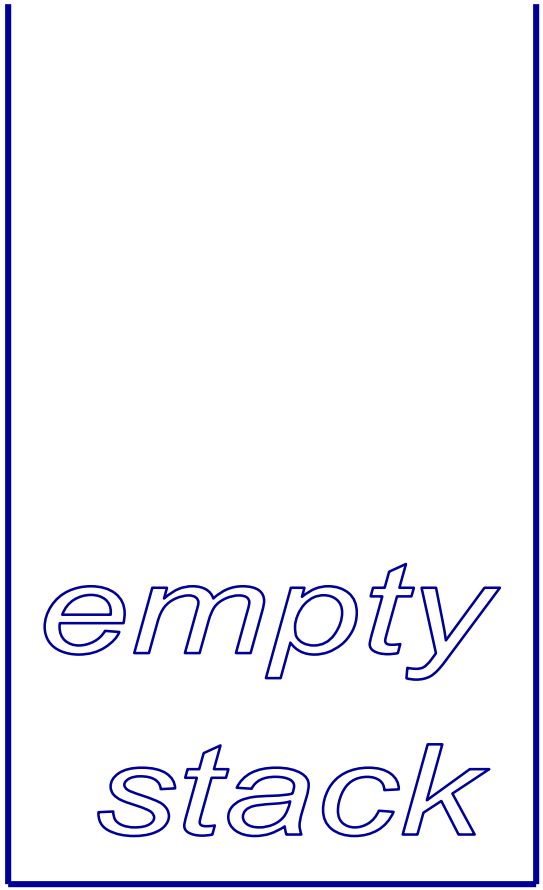
stack.pop(&c)->'d'

	Items	Top
4		0
3		
2	c	
1	d	
0	a	



stack.pop(&c)->'a'

	Items	Top
4		-1
3		
2	c	
1	d	
0	a	



Strengths and Limitations of Stacks

The main strengths of stacks are:

- very simple data type
- very fast
- direct access to last (first) element added

Limitation of stacks:

- the set of operations is very restricted (no access to elements other than last (first), no searching, no iterating).

Stacks in daily conversation

- "I'm afraid I've got real work to do, so this'll have to be pushed way down on my stack." Here stack refers to the set of things a person has to do in the future
- "I haven't done it yet because every time I pop my stack something new gets pushed." If a person says he has popped something from his stack, that means he has finally finished working on it and can now remove it from the list of things hanging overhead.
- If you are interrupted several times in the middle of a conversation one may say, "My stack overflowed" means "I forget what we were talking about." The implication is that more items were pushed onto the stack than could be remembered, so the least recent items were lost.

Applications of Stacks

- Stacks are a very common data structure
  - compilers
    - parsing data between delimiters (brackets)
  - operating systems
    - program stack
  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - artificial intelligence
    - finding a path

Applications of Stacks

- Checking matching brackets
- Page-visited history in a Web browser
- Base Conversion
- Undo sequence in a text editor
- Store the function arguments, the local variables/objects and the return address of the calling function
- Evaluating algebric expressions
- Converting Infix to Postfix

Balanced Symbol Checking

- In processing programs and working with computer languages there are many instances when symbols must be balanced

  `{},[],()`

Balancing Bracket Pairs

- A string is balanced if:
  - all the opening and closing brackets in the string are paired
    - opening brackets: ([{
    - closing symbols:)] }
  - each closing bracket in the string must correspond to the previous unmatched opening bracket *of the same kind*:
    - e.g. "([ ])" -- correct
    - e.g. "([ )]" -- wrong

Algorithm for Balanced Bracket Checking

Any algorithm for bracket matching has to keep in mind the following rules:

1. The number of opening and closing brackets must be same
2. The order of brackets is also important. The opening bracket must come before matching closing bracket. Thus () is valid, while)(is invalid.
3. Two pairs of matched brackets must either nested or be disjoint. We can have [()] or [](), but not ([)]

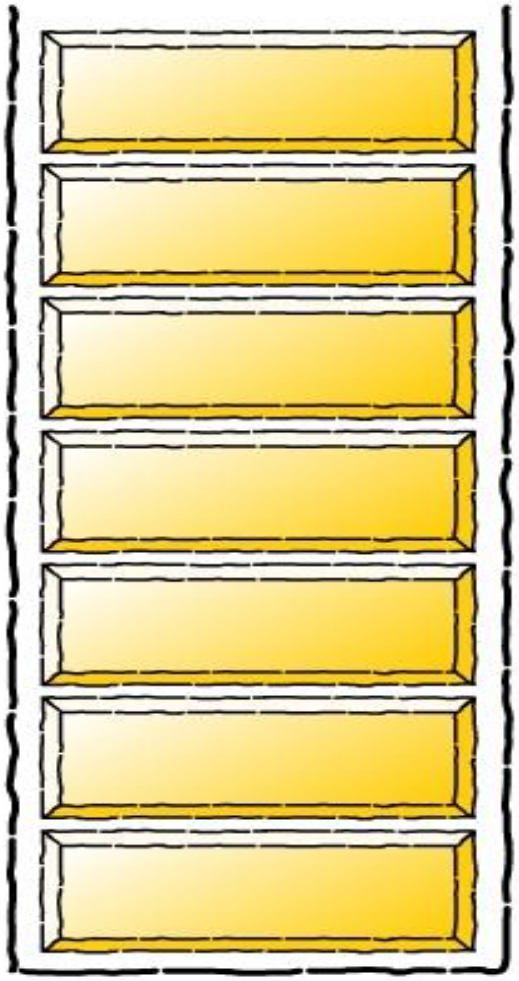
All the above rules can easily be checked by using a stack

Algorithm for Matching Bracket Checking

- Read symbols until end of expression
  - if the symbol is an opening bracket push it onto the stack
  - if it is a closing bracket do the following
    - if the stack is empty report an error
    - otherwise pop the stack. If the symbol popped does not match the closing bracket report an error
- At the end of the expression if the stack is not empty report an error

Matching Brackets Using Stacks

Stack



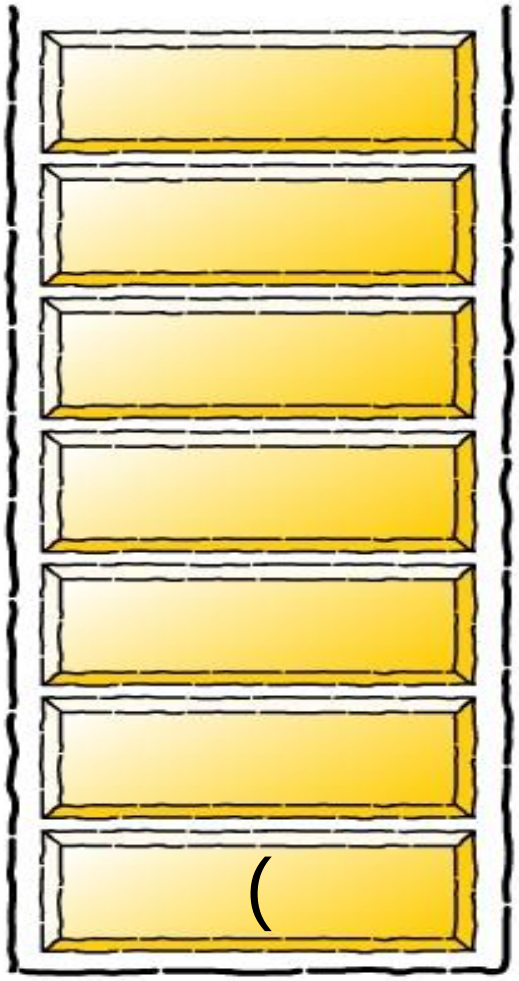
Expression

({([])})

Operation

Matching Brackets Using Stacks

Stack



Expression

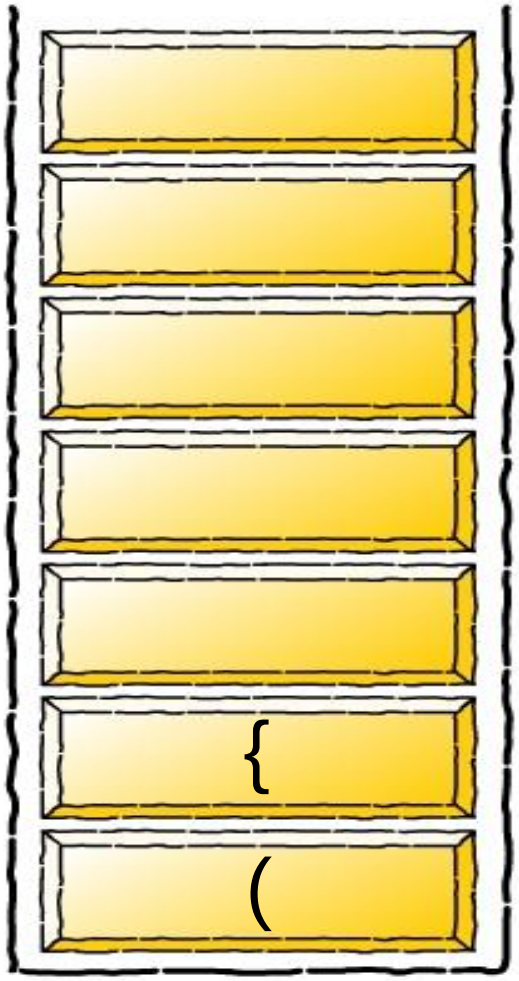
({ ([ ]) })

Operation

Push (

Matching Brackets Using Stacks

Stack



Expression

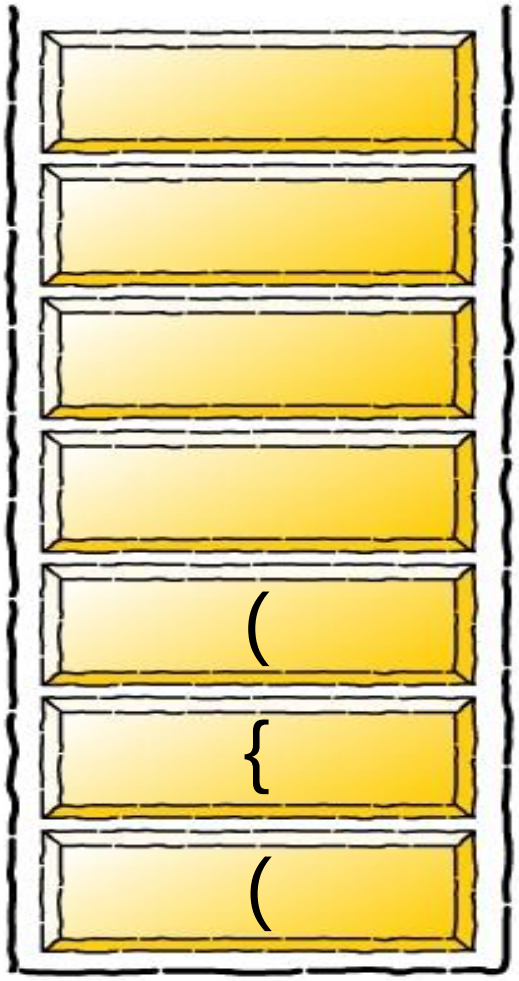
({ ([ ]) })

Operation

Push {

Matching Brackets Using Stacks

Stack



Expression

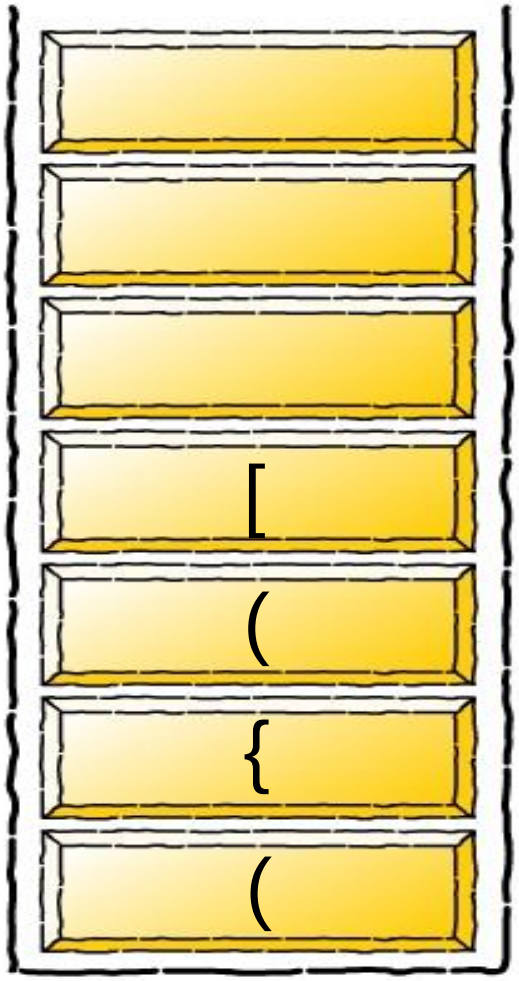
({ ([ ]) })

Operation

Push (

Matching Brackets Using Stacks

Stack



Expression

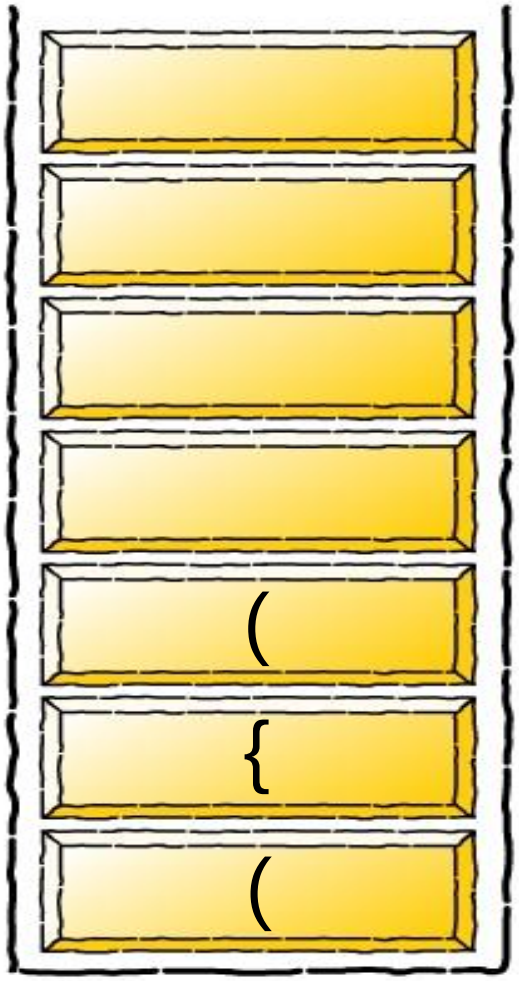
({ ([ ]) })

Operation

Push [

Matching Brackets Using Stacks

Stack



Expression

({ ([ ]) })

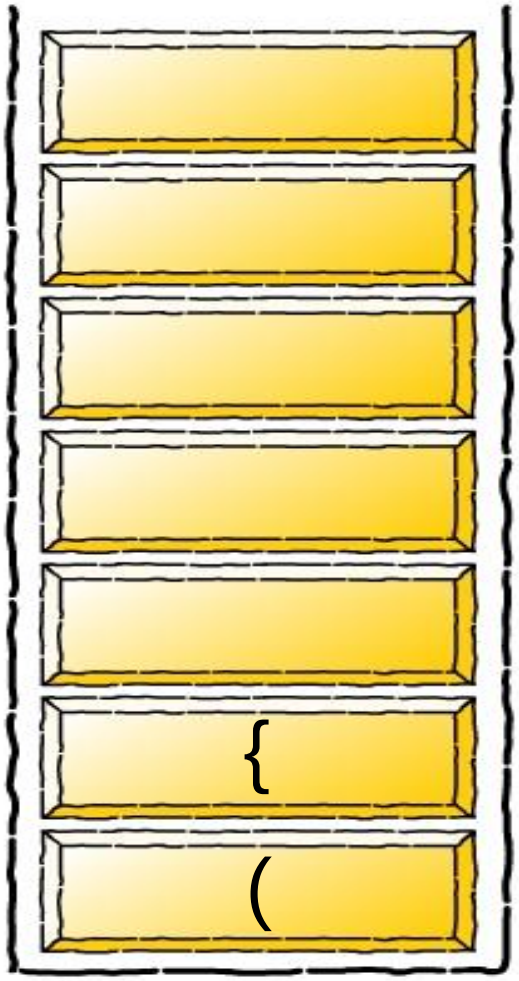
Operation

Pop [

[ ] match

Matching Brackets Using Stacks

Stack



Expression

({ ([ ]) })

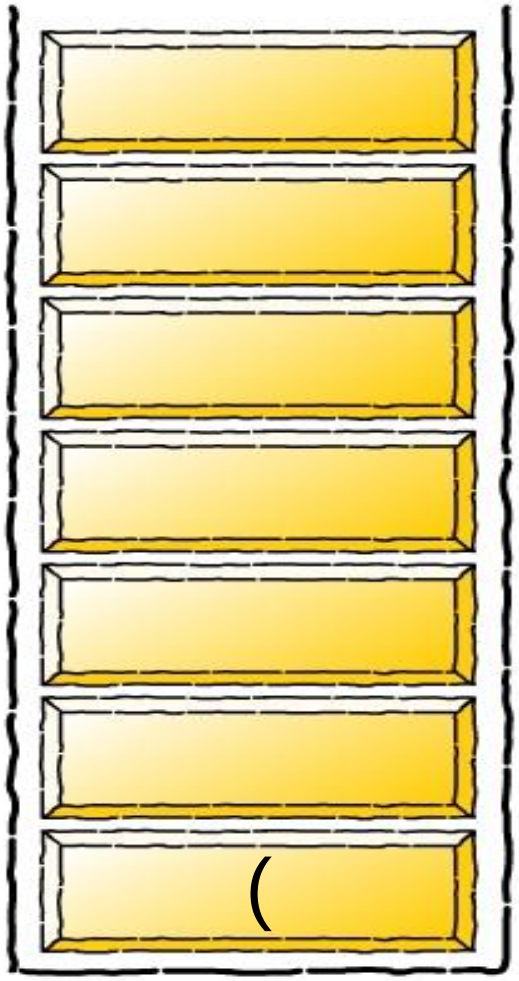
Operation

Pop (

( ) match

Matching Brackets Using Stacks

Stack



Expression

({ ([ ]) })

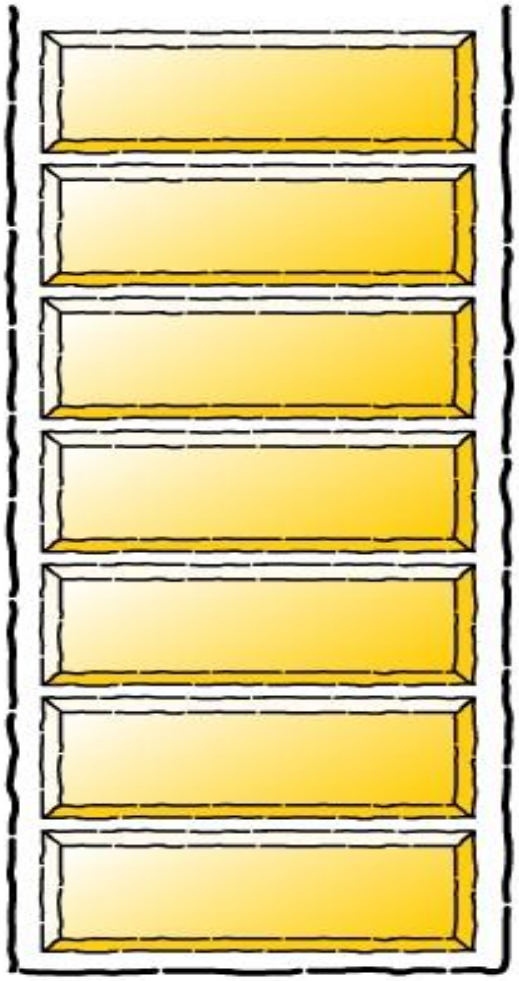
Operation

Pop {

{ } match

Matching Brackets Using Stacks

Stack



Expression

({ ([ ]) })

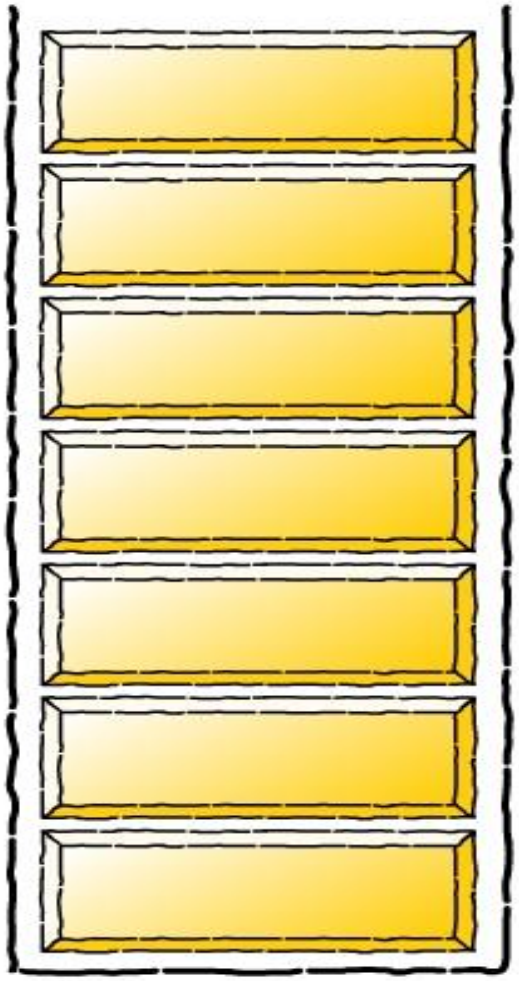
Operation

Pop (

( ) match

Matching Brackets Using Stacks

Stack



Expression

({([])})

Operation

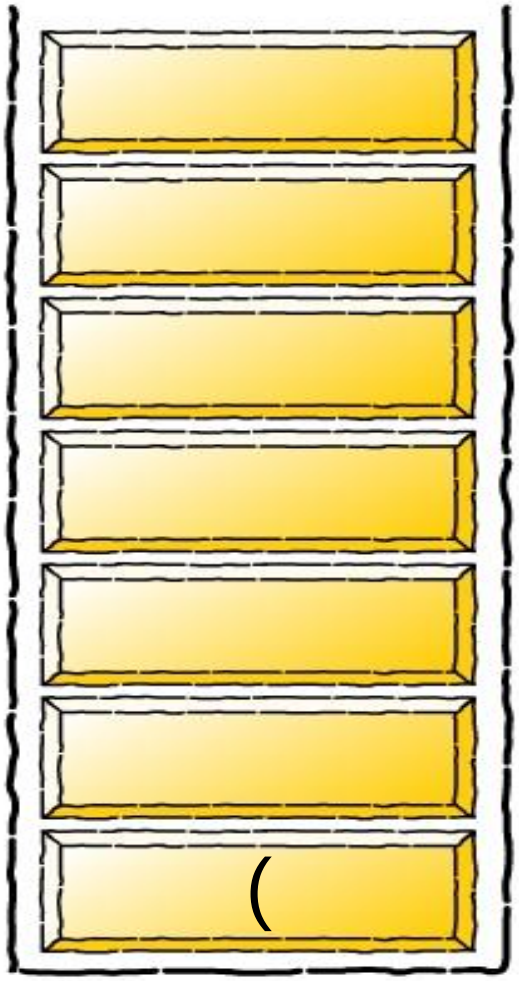
End of Expression

Stack empty

Braces match!!!

Matching Brackets Using Stacks

Stack



Expression

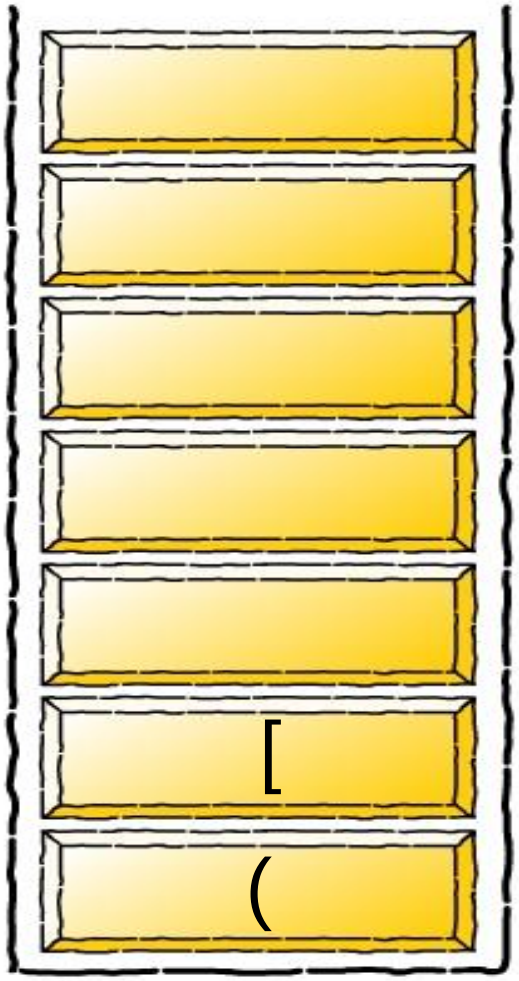
([ )])

Operation

Push (

Matching Brackets Using Stacks

Stack



Expression

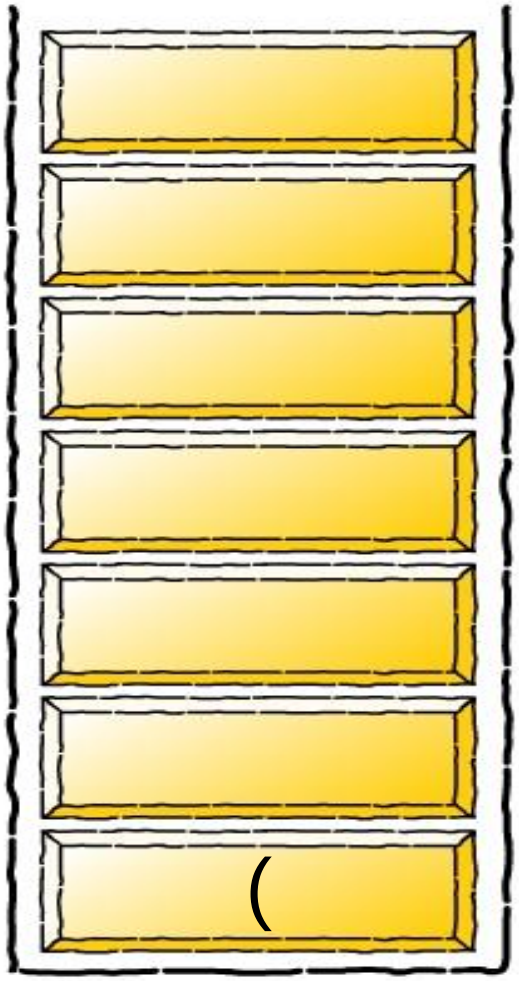
([ )])

Operation

Push [

Matching Brackets Using Stacks

Stack



Expression

([ )])

Operation

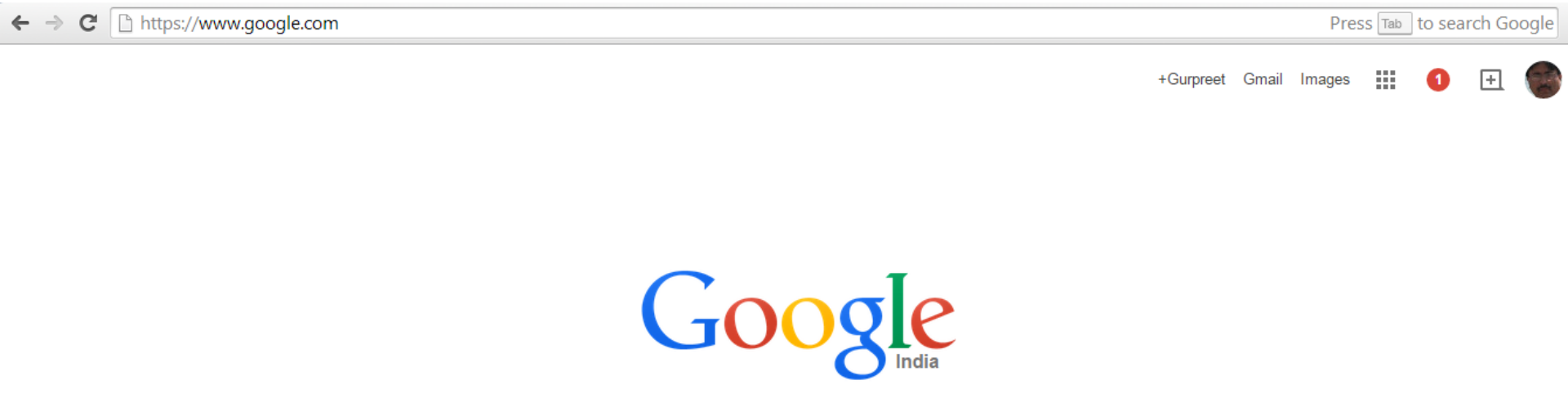
Pop [

[ ) do not match

Thus brackets do not match!!

```
bool CheckBraces(char *c){
   char c1;
   int i=-1;
   Stack stack(strlen(c));
   while(c[++i]){
      if(c[i]=='(' || c[i]=='{' || c[i] == '[')
           stack.push(c[i]);
      if(c[i]==')' || c[i]=='}' || c[i] == ']'){
           if(!stack.pop(&c1)) return false;
           if(c1=='(' && c[i] !=')')return false;
           if(c1=='[' && c[i] !=']')return false;
           if(c1=='{' && c[i] !='}')return false;
        }
   }
   if(!stack.isEmpty())return false;
   return true;
}
```

Page-visited history in a Web browser

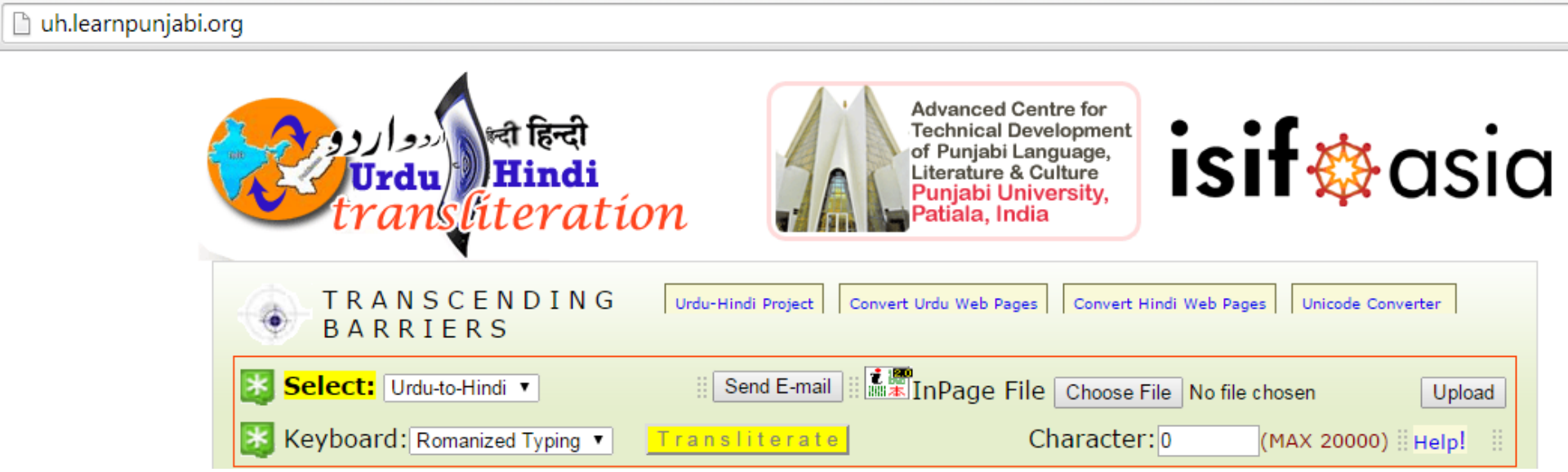


Page-visited history in a Web browser

www.learnpunjabi.org/

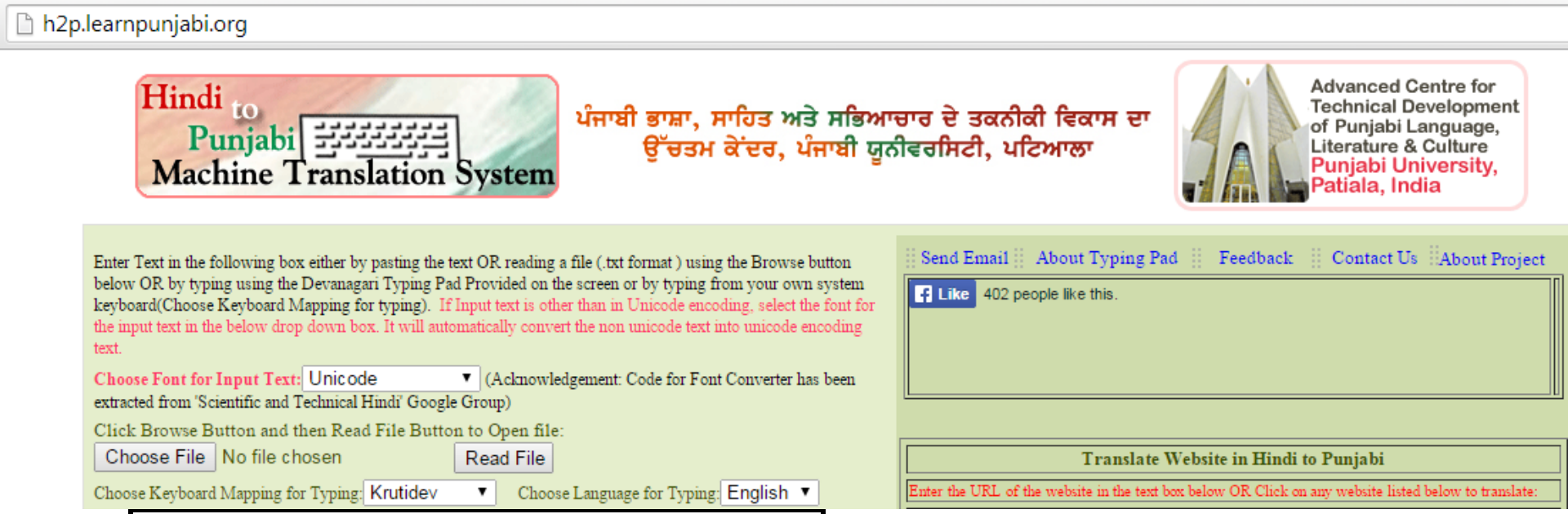
Google.com

Page-visited history in a Web browser



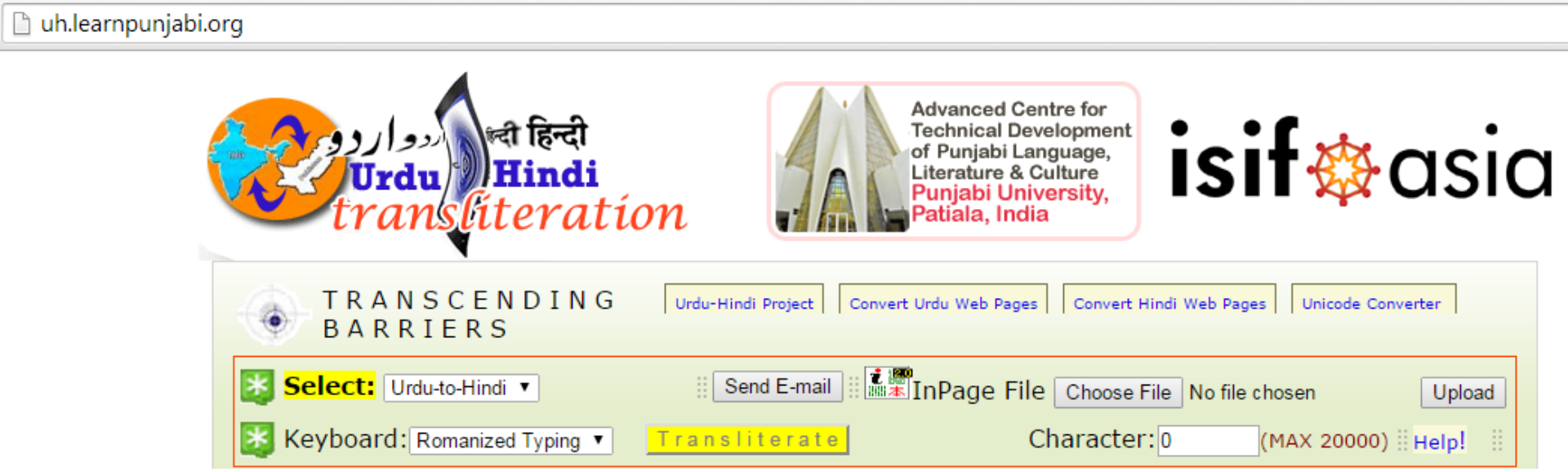
Learnpunjabi.org
Google.com

Page-visited history in a Web browser



Uh.learnpunjabi.org
Learnpunjabi.org
Google.com

Page-visited history in a Web browser



Learnpunjabi.org
Google.com

Page-visited history in a Web browser

www.learnpunjabi.org/

Google.com

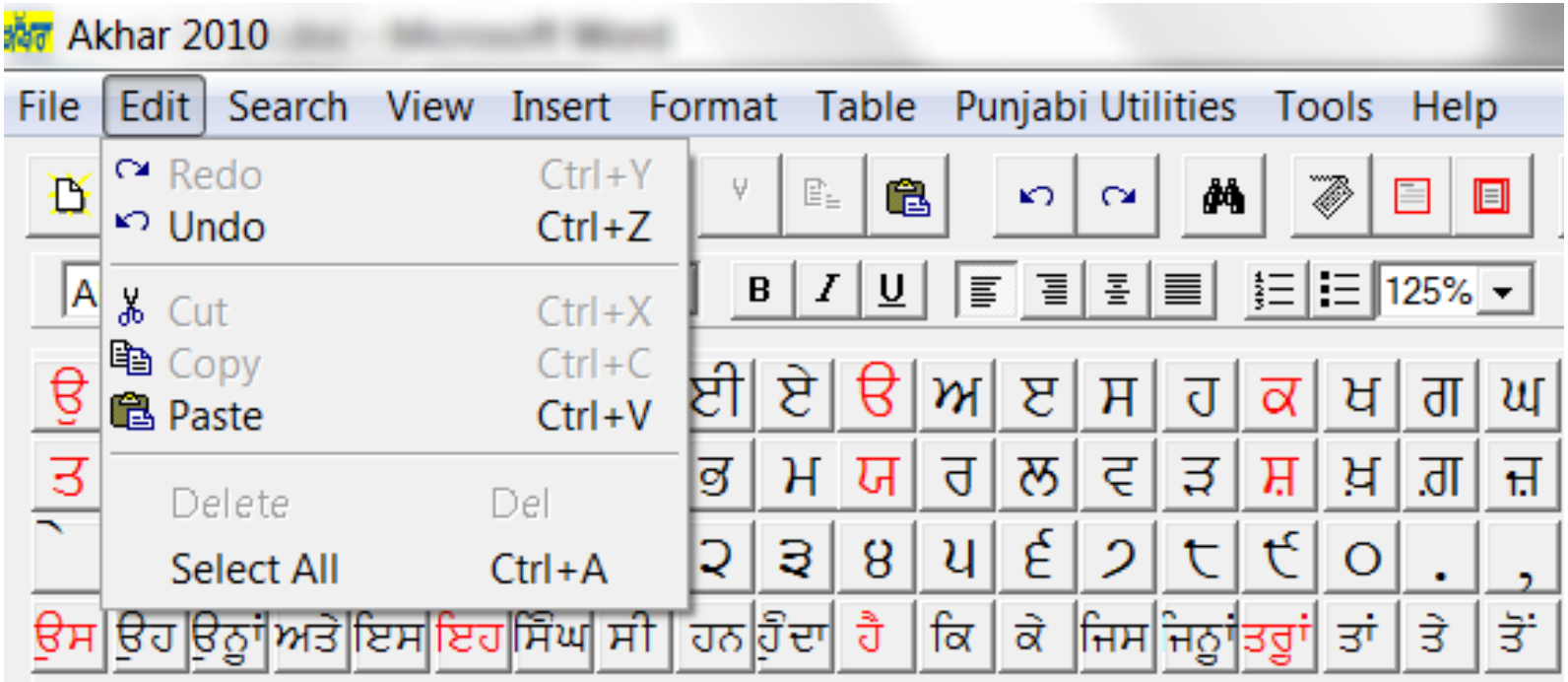
Page-visited history in a Web browser

Learnpunjabi.org
Google.com

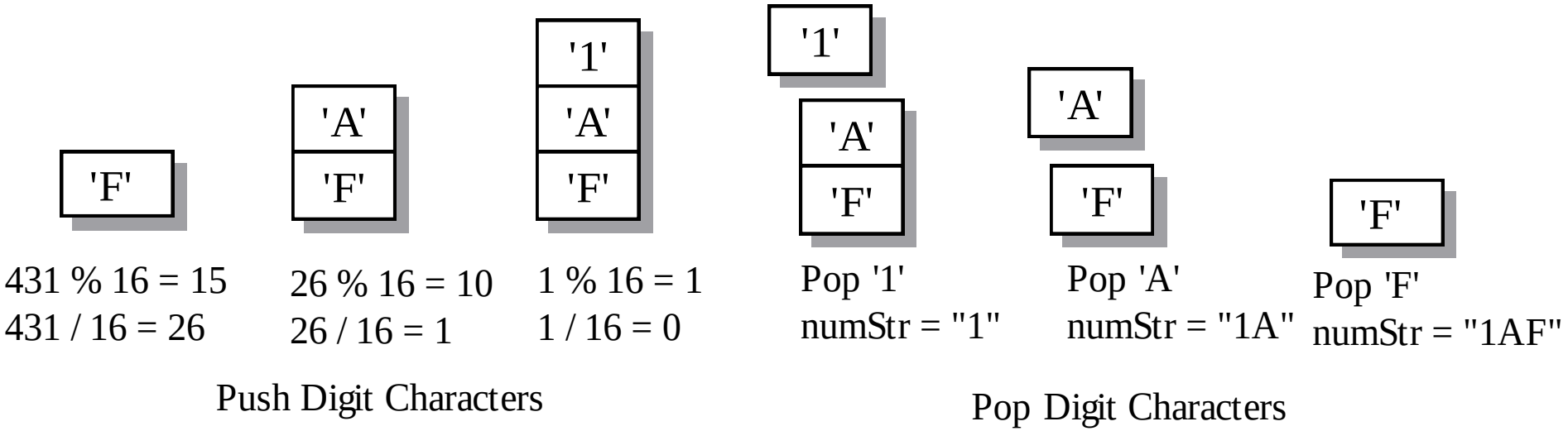
Back and Forward Buttons in a Web browser

- To allow the user to move both forward and backward two stacks are employed.
- When the user presses the back button, the link to the current web page is stored on a separate stack for the forward button.
- As the user moved backward through previous pages, the link to each page is moved in turn from the back to the forward stack.
- When the user pushes the forward button, the action is the reverse of the back button. Now the item from the forward stack is popped, and becomes the current web page. The previous web page is pushed on the back stack.

Undo sequence in a text editor



Using a Stack to Create a Hex Number



```
void DecToHex(int dec, char *hex){
    int rem;
    Stack stack(8);
    while(dec){
        rem = dec % 16;
        if(rem<10)
                stack.push(rem + '0');
        else
                stack.push(rem-10 + 'a');
        dec/=16;
    }
    while(!stack.isEmpty())
        stack.pop(hex++);
    *hex = 0;
}
```

Algebric Expression

- An Operand is the quantity (unit of data) on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4,0,9.7,-1 etc.
- An Operator is a symbol which signifies a mathematical or logical operation between the operands. Example of familiar operators include +,-,*, /, ^
- An Algebraic Expression is a legal combination of operands and the operators. An example of algebric expression is 3+5

Infix, Postfix and Prefix Notations

Depending on where we place this operator and the operands, we have three different notations

- **INFIX:** Expressions in which operands surround the operator are called Infix expressions, e.g. A+B, 8*4 etc.

- **PREFIX::** In Prefix notation operator comes before the operands, e.g. +AB, *+xyz etc.

- **POSTFIX:** In Postfix notation the operator comes after the operands, e.g. AB+, xyz+* etc.

Need for Prefix and Postfix

- For human beings, infix expressions are easy to read and understand as compared to Prefix and Postfix notations, but they are difficult to parse for the computers.
- To evaluate an infix expression we need to consider Operators' Priority, Associative property and brackets ( )
  - Thus for example, an expression such as A*(B+C) / D will mean: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."
- As the expression has to be scanned multiple times and intermediate results have to be stored, it makes computer evaluation of infix expressions more difficult and time consuming than is necessary.

Need for Prefix and Postfix

- In contrast postfix and prefix expression forms do not rely on operator priorities or brackets and so are much easier to evaluate.
- In case of postfix expression, one does not have to worry about operator precedence as the order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. As for example, for the infix expression A * (B + C) / D, the equivalent postfix expression is A B C + * D / and since the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication. .

Examples of infix to prefix and postfix

INFIX	POSTFIX	PREFIX	NOTES
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Note : Operand order remains the same in all three notations

Need for Prefix and Postfix

- Prefix and postfix expressions
  - Never need
    - Precedence rules
    - Association rules
    - Parentheses
  - Have
    - Simple grammar expressions
    - Straightforward recognition and evaluation algorithms
    - Are very easy to evaluate

Fully Parenthesized Expression

- A Fully Parenthesized Expression has exactly one set of Parentheses enclosing each operator and its operands.
  - ((A + B) * C)
  - ((A + B) * (C + (D ^ E)))

Infix to Prefix Conversion

- Convert the infix expression to Fully Parenthesized Expression (FPE)
- Move each operator to the left of its operands & remove the parentheses:
- Infix : A + B * C + D
- FPE : ((A + (B * C)) + D)
- Prefix conversion steps:

1. ((A + * B C) + D)
2. (+ A * B C) + D)
3. + + A * B C D

Some exercises to try

Convert to prefix and postfix

- 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 + * /

Evaluation of Infix Expression

- To evaluate an infix expression we need to consider Operators' Priority, Associative property and brackets ( )
  - Thus for example, an expression such as A*(B+C) / D will mean: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

- As the expression has to be scanned multiple times and intermediate results have to be stored, it makes computer evaluation of infix expressions more difficult and time consuming than is necessary.

Evaluation of Infix Expression

- The problem of evaluating infix expression can be broken in to 2 stages:

1. Infix to Postfix Conversion

   5+6*7 -> 567*+

2. Evaluating a Postfix expression

   567*+ = 47

Precedence Table

Symbol	Off Stack Precedence	On Stack Precedence
+	1	1
-	1	1
*	2	2
/	2	2
^	3	3
(	4	0
)	0	0

Algorithm for Infix to Postfix

Set the Postfix String to Empty String

Scan the Infix expression left to right

- If the character x is an operand
  - Append the character to the Postfix String
- If the character x is a left or right bracket
  - If the character is (
    - Push it into the stack
  - if the character is)
    - Repeatedly pop and append to the Postfix String all the operators/characters until (is popped from the stack.
    - Do not append the brackets to the Postfix String

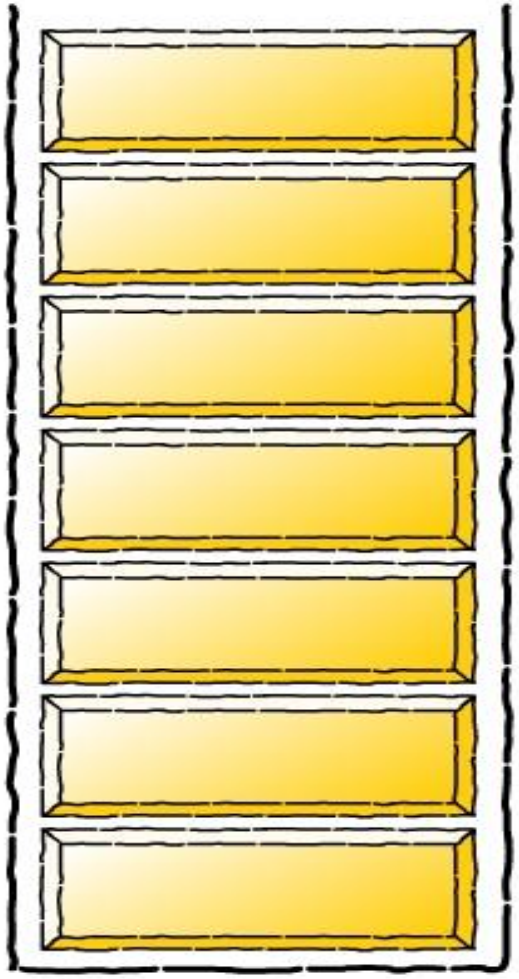
Algorithm for Infix to Postfix contd..

- If the character x is an operator
  - Check the character y currently at the top of the stack.
  - If Stack is empty or y=(or y is an operator of **lower precedence** than x, then push x into stack.
  - If y is an operator of **higher or equal** precedence than x, then pop and output y and push x into the stack.

When all characters in infix expression are processed repeatedly pop the character(s) from the stack and append them to the Postfix String until the stack is empty.

Infix to postfix conversion

Stack



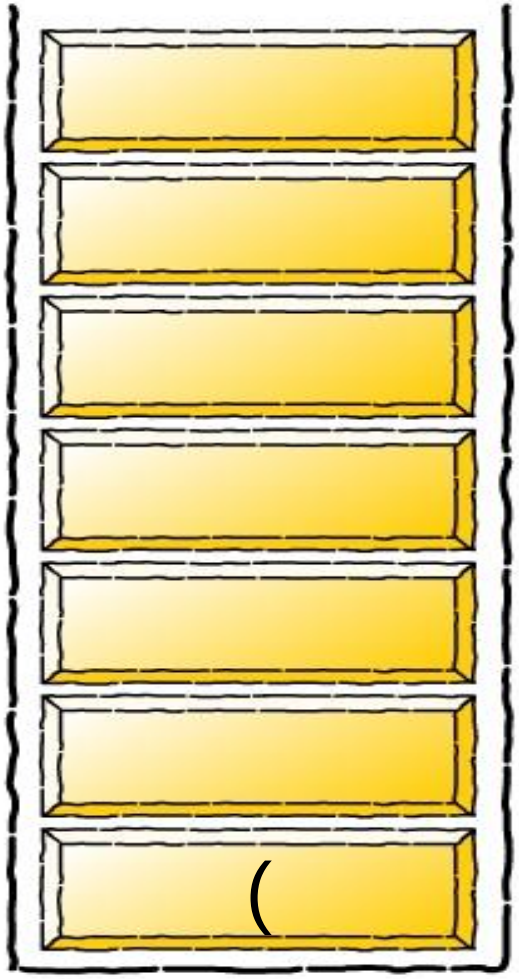
Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

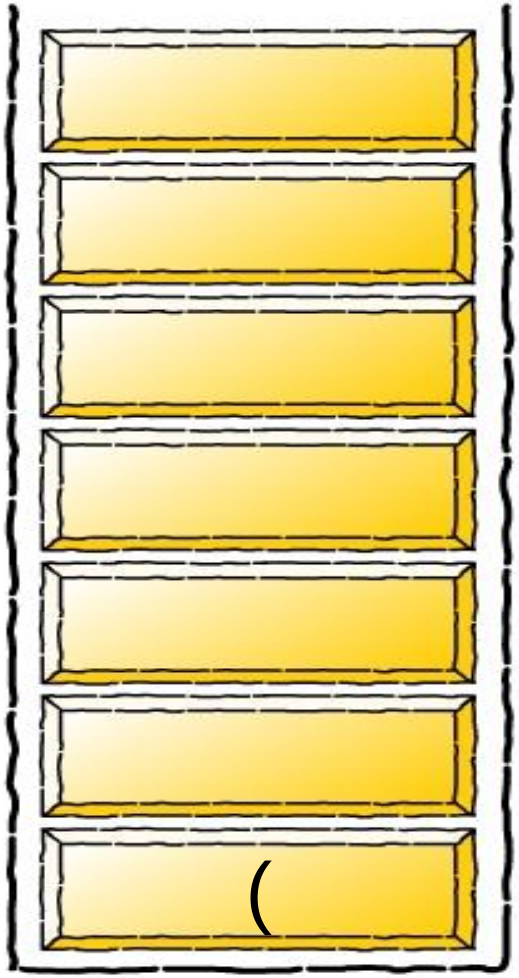
Postfix Expression

Comments

(is pushed onto the operator stack

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

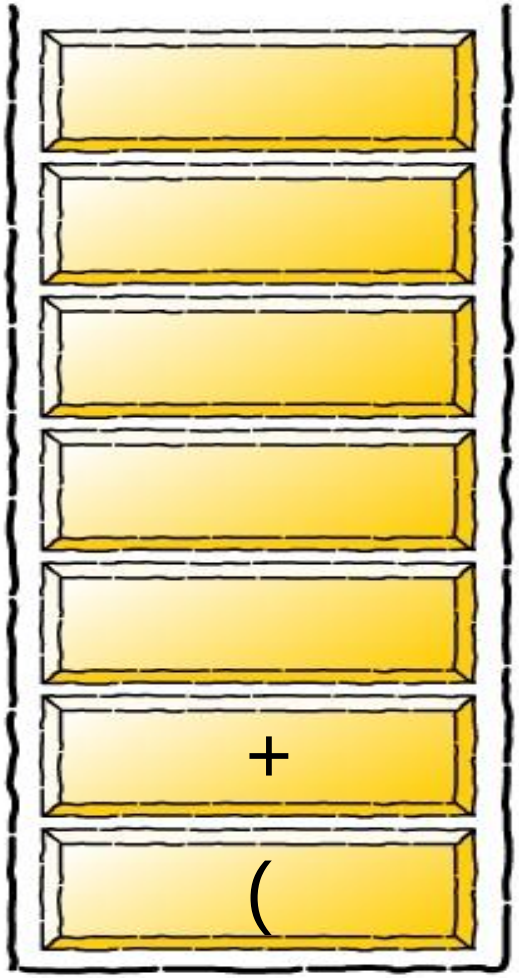
a

Comments

a is an operand and it is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

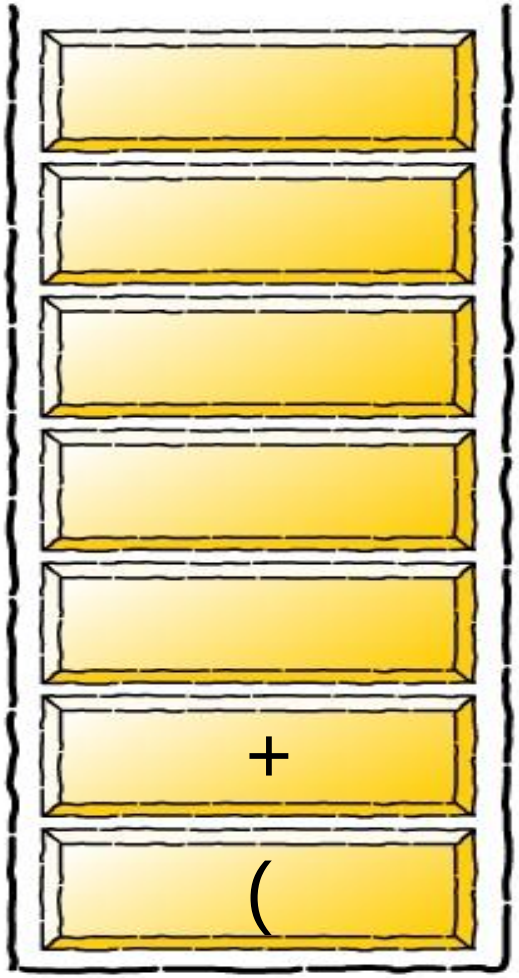
a

Comments

The precedence of + is greater than (and so it pushed on the stack

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

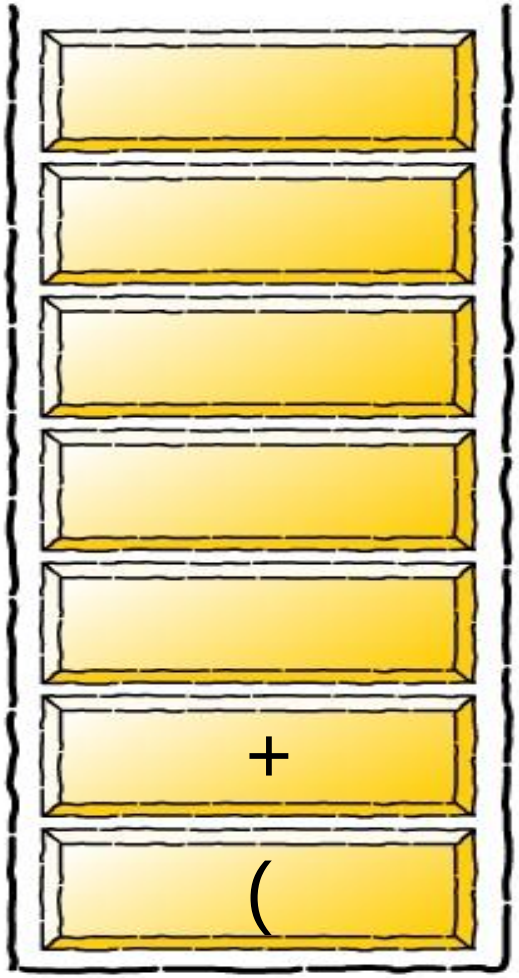
a b

Comments

b is an operand and it is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

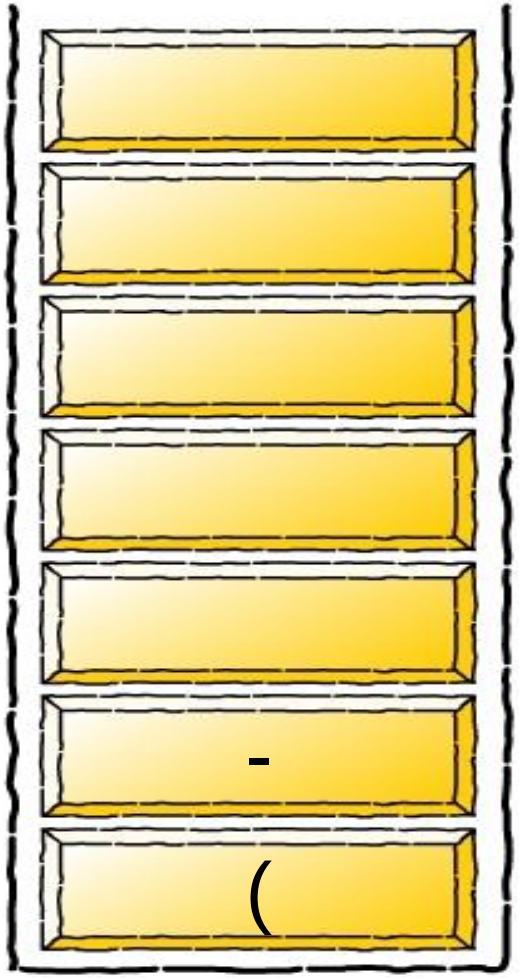
a b

Comments

The precedence of - is same as + and so we pop + and append it to postfix string.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

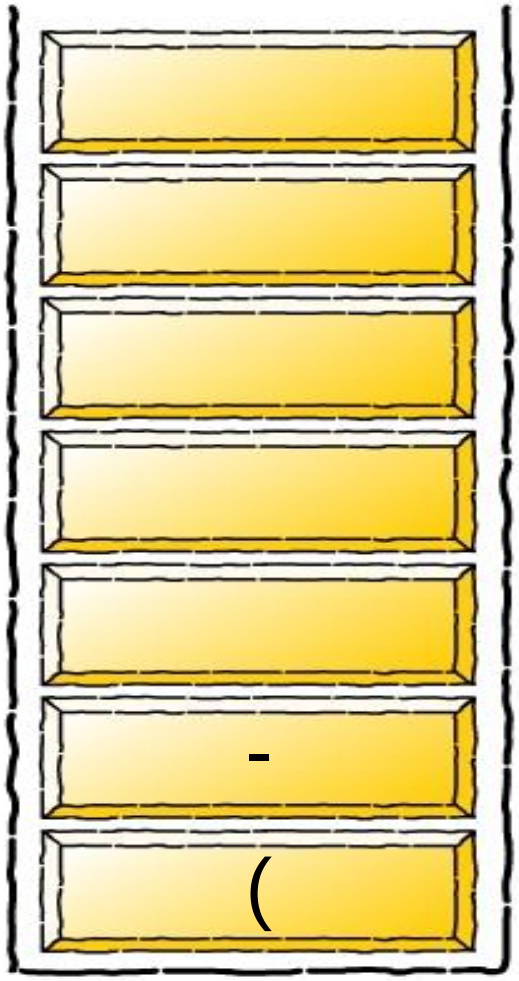
a b +

Comments

Then operator – is next pushed onto the stack

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

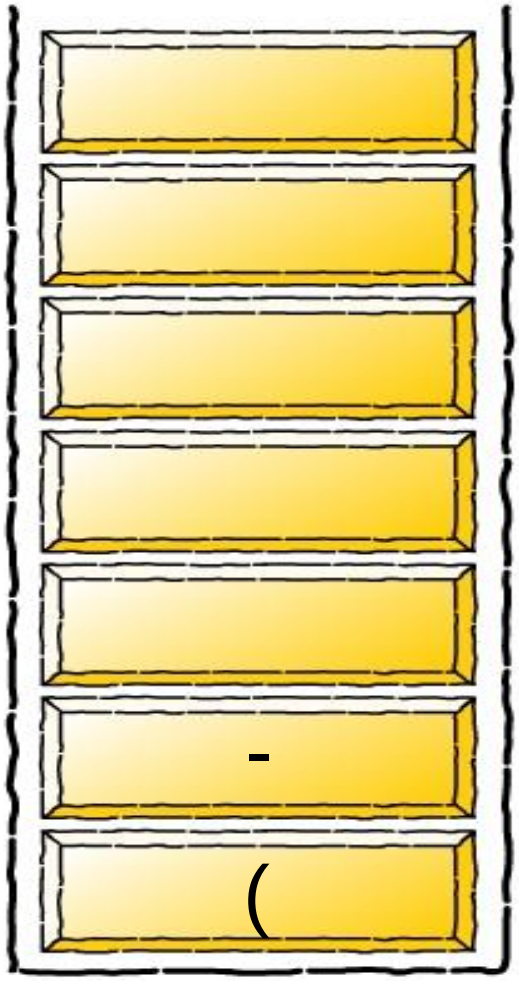
a b + c

Comments

c is an operand and it is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

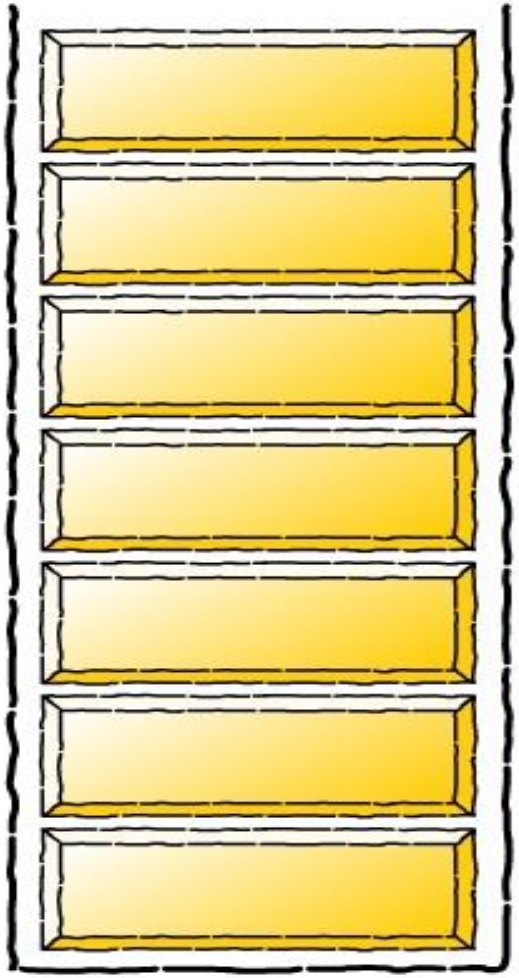
a b + c

Comments

) causes all the stack elements to be popped till (is encountered.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

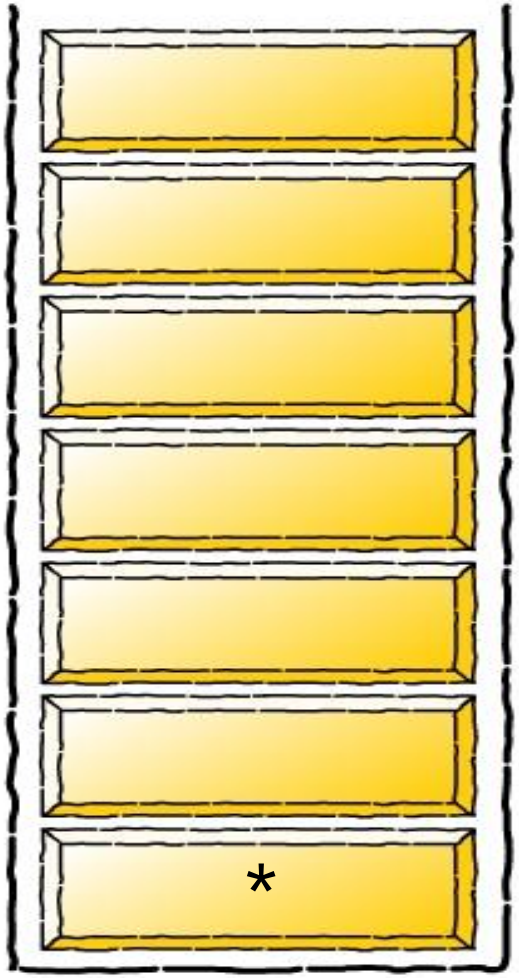
a b + c -

Comments

The popped elements, except (are appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

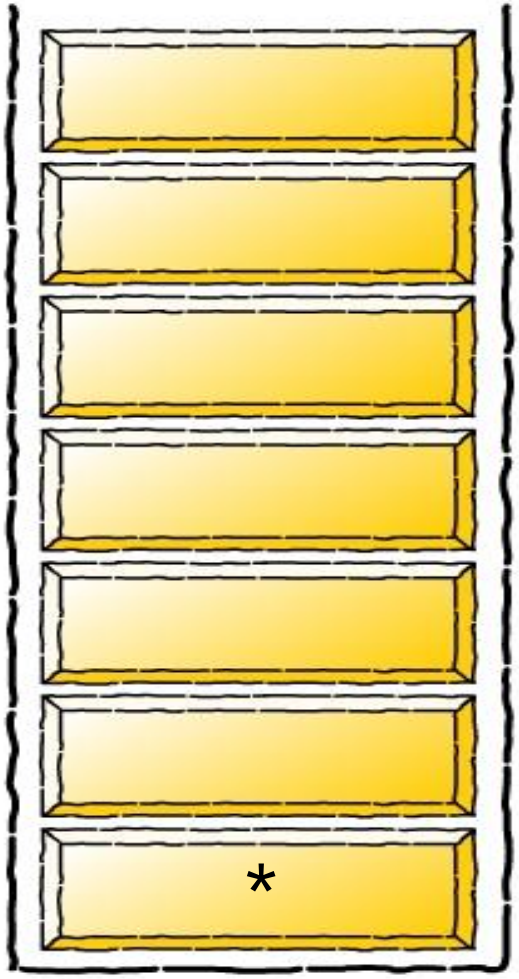
a b + c -

Comments

The operator * is pushed onto the stack as the stack is empty.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

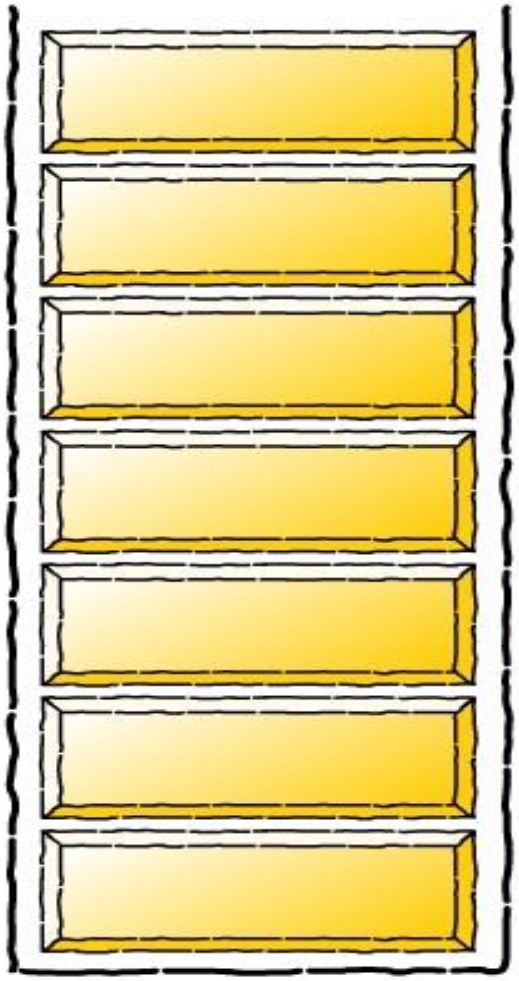
a b + c - d

Comments

The operand d is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

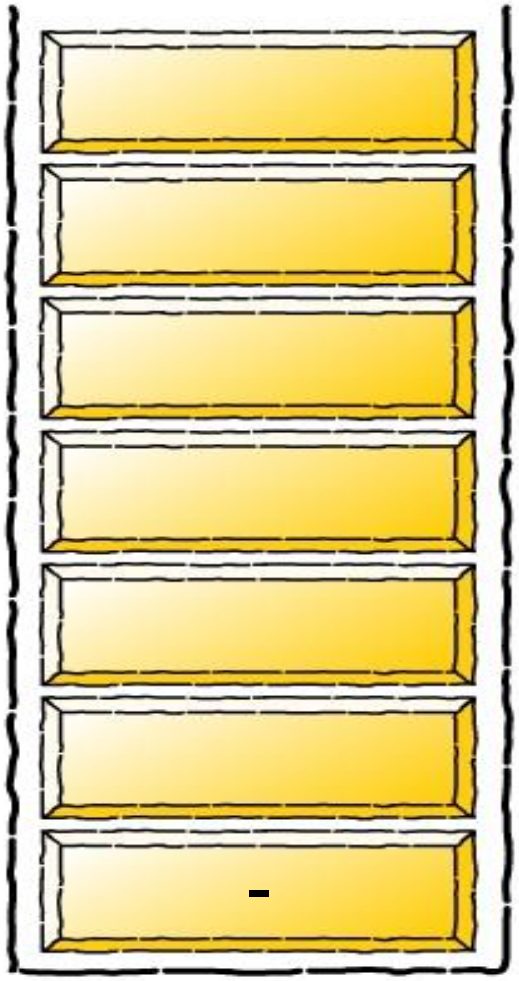
a b + c – d *

Comments

The precedence of - is lesser than * and so we pop * and append it to postfix string.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

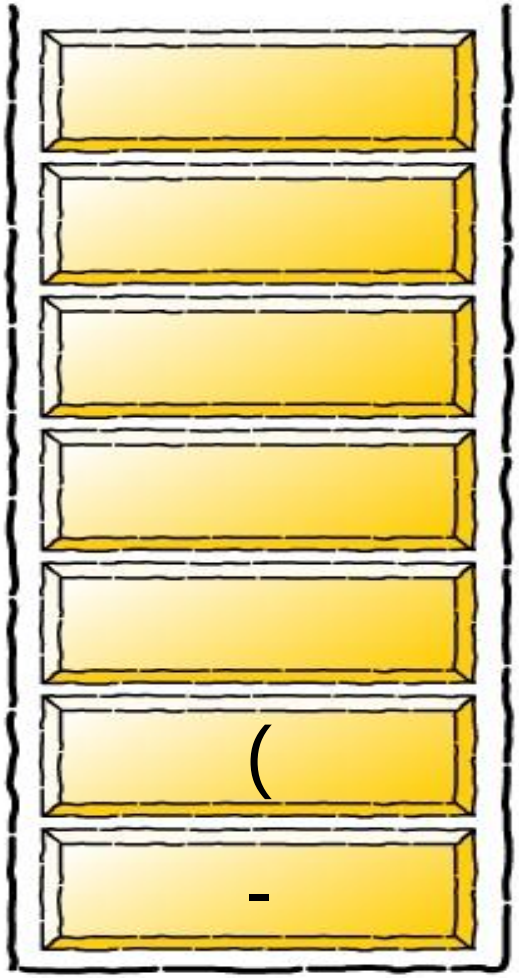
a b + c – d *

Comments

Then operator – is next pushed onto the stack

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

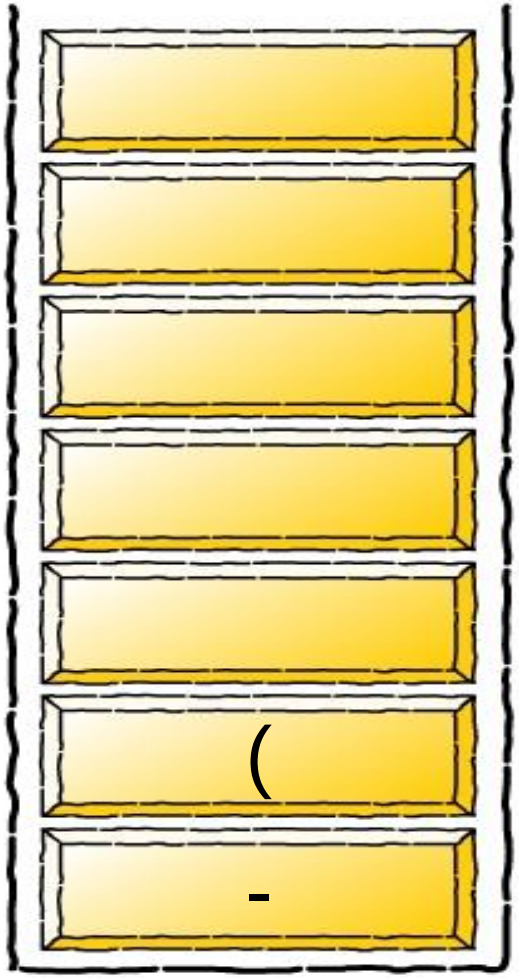
a b + c – d *

Comments

(is pushed onto the operator stack, as it has highest offstack precedence

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

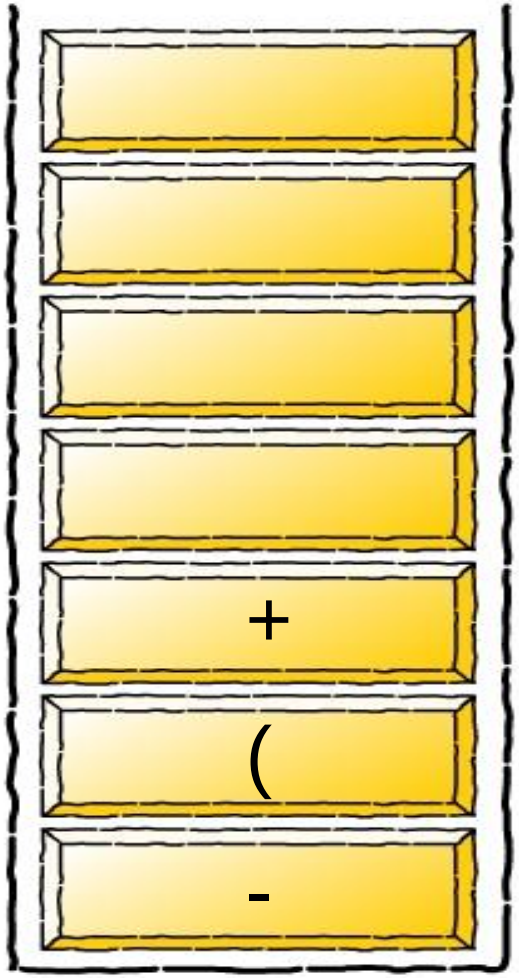
a b + c – d * e

Comments

The operand e is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

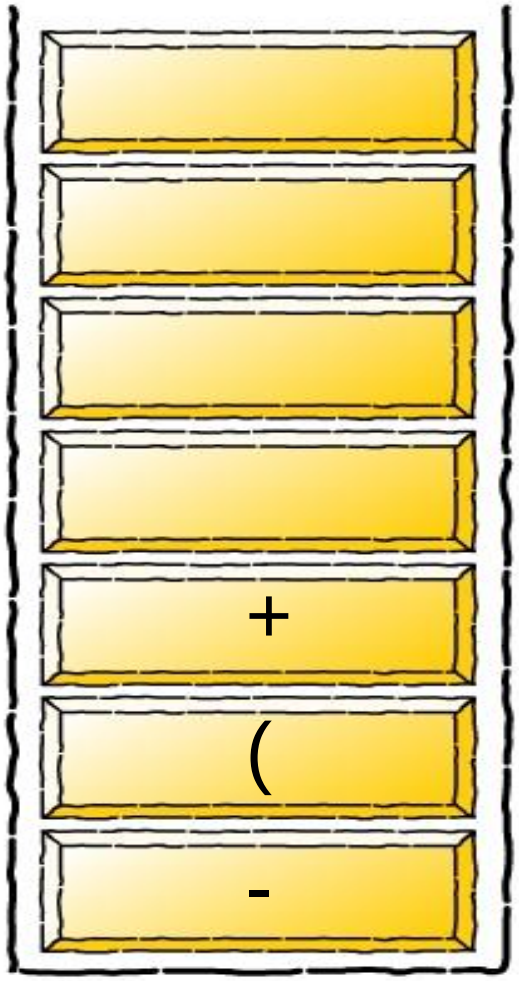
a b + c – d * e

Comments

The precedence of + is greater than (and so it pushed on the stack

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

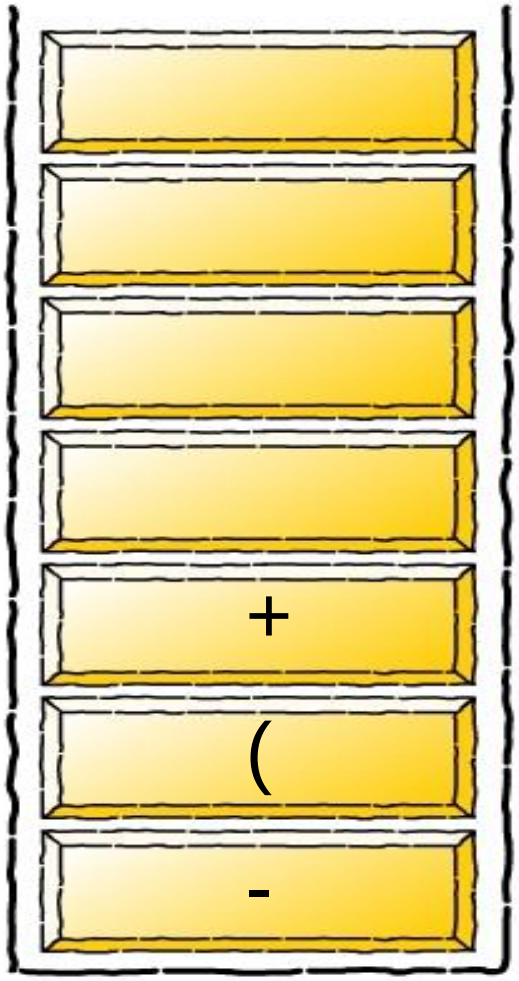
a b + c – d * e f

Comments

The operand f is appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

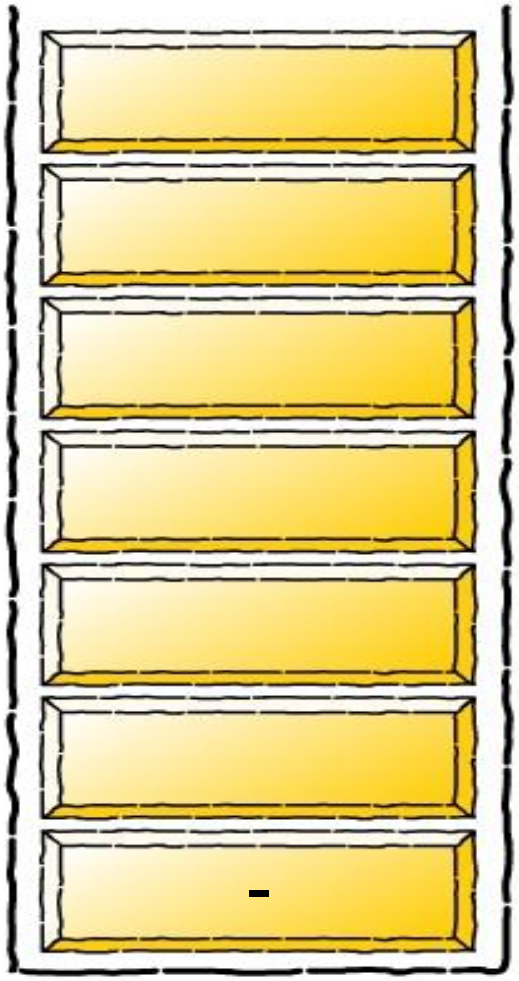
a b + c – d * e f +

Comments

) causes all the stack elements to be popped till (is encountered.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

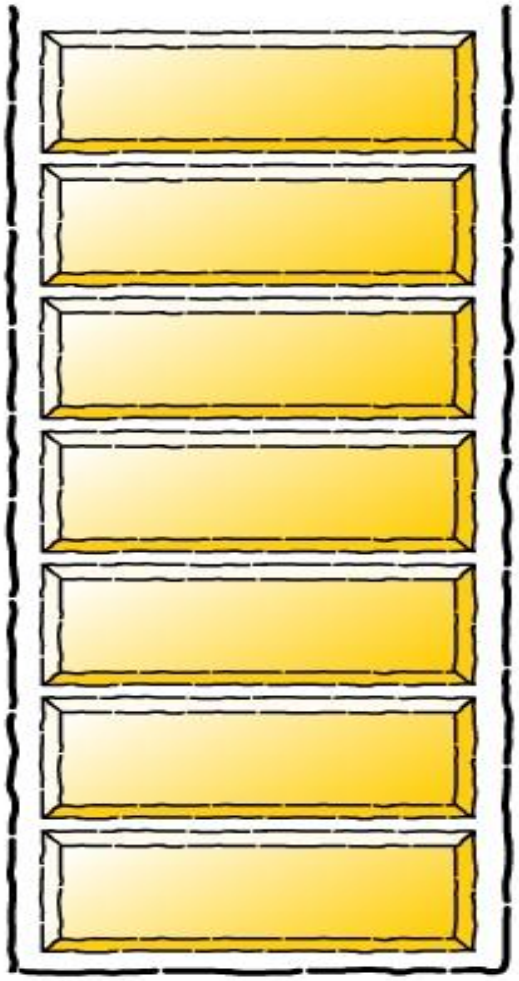
a b + c – d * e f +

Comments

The popped elements, except (are appended to postfix expression.

Infix to postfix conversion

Stack



Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

a b + c – d * e f + -

Comments

On reaching end of string. All elements are popped and appended to expression.

Example

1 - 2 * 3 ^ 3 - (4 + 5 * 6) * 7

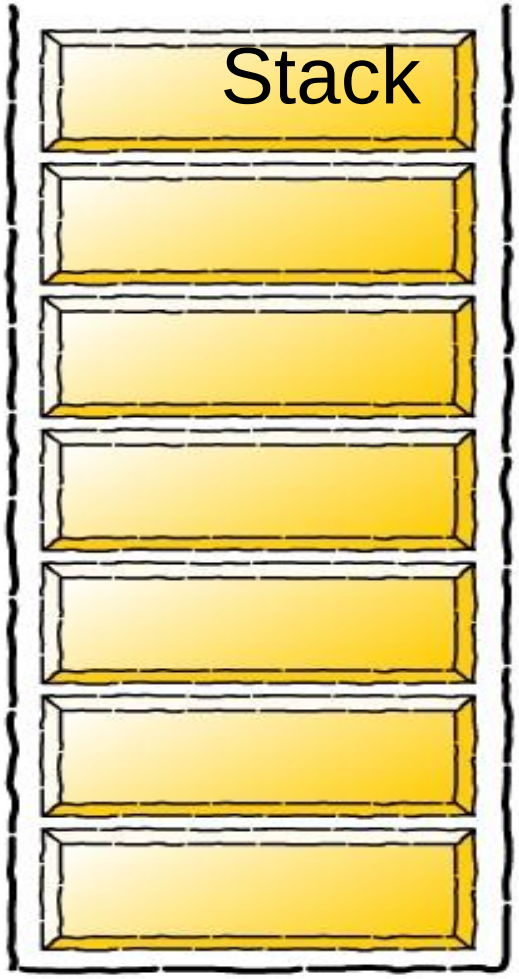
Show algorithm in action on above equation

Some exercises to try

Convert to prefix and postfix

- 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 + * /

Evaluation of Postfix expression

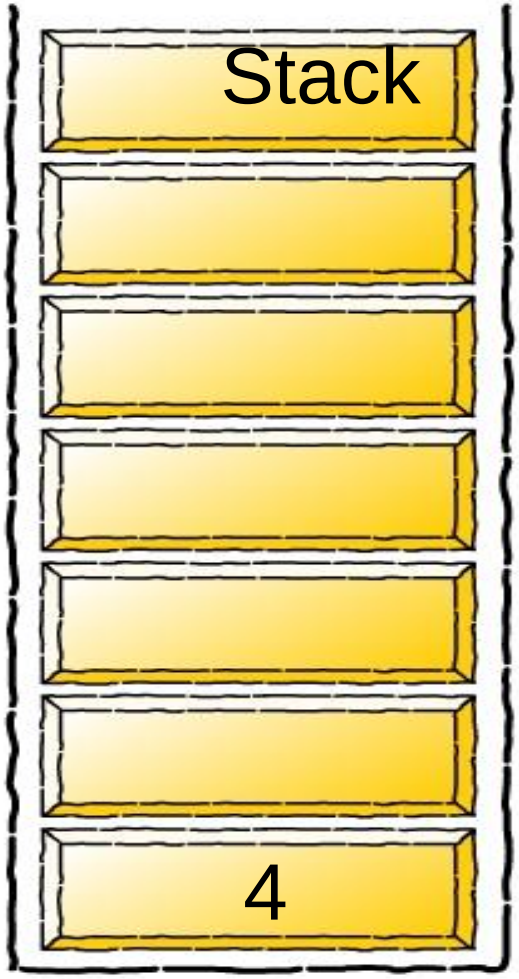


Expression

4 5 + 7 2 - *

Operation

Evaluation of Postfix expression



Expression

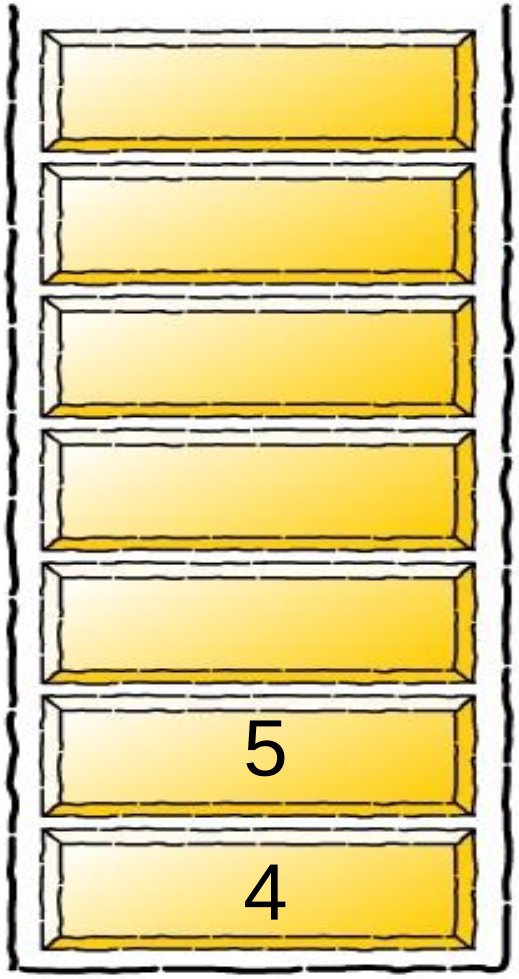
4 5 + 7 2 - *

Operation

Push 4

Evaluation of Postfix expression

Stack



Expression

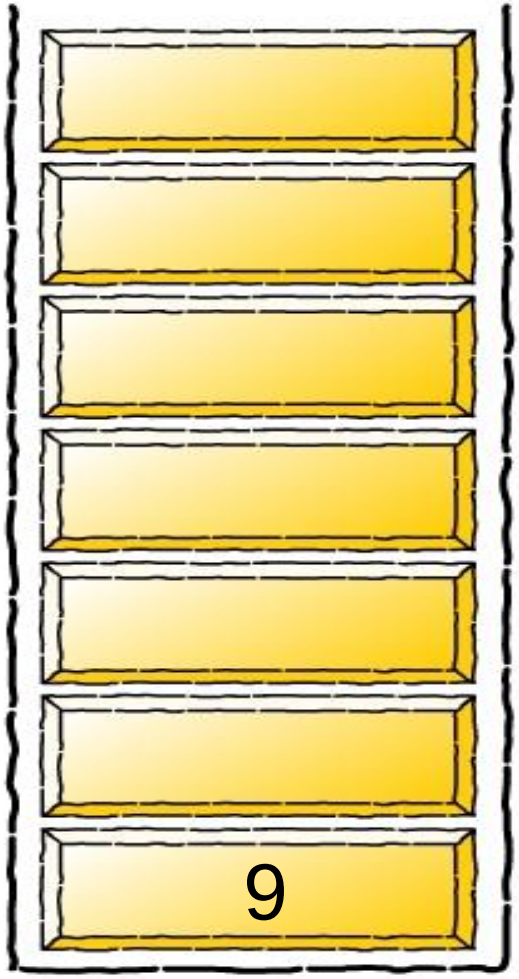
4 5 + 7 2 - *

Operation

Push 5

Evaluation of Postfix expression

Stack



Expression

4 5 + 7 2 - *

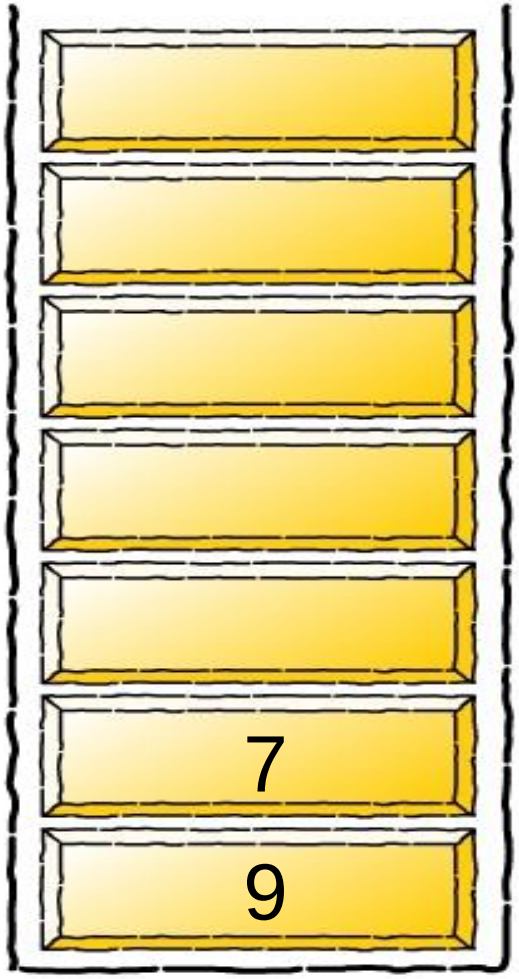
Operation

Pop 5 and 4. Add

Push the result (9)

Evaluation of Postfix expression

Stack



Expression

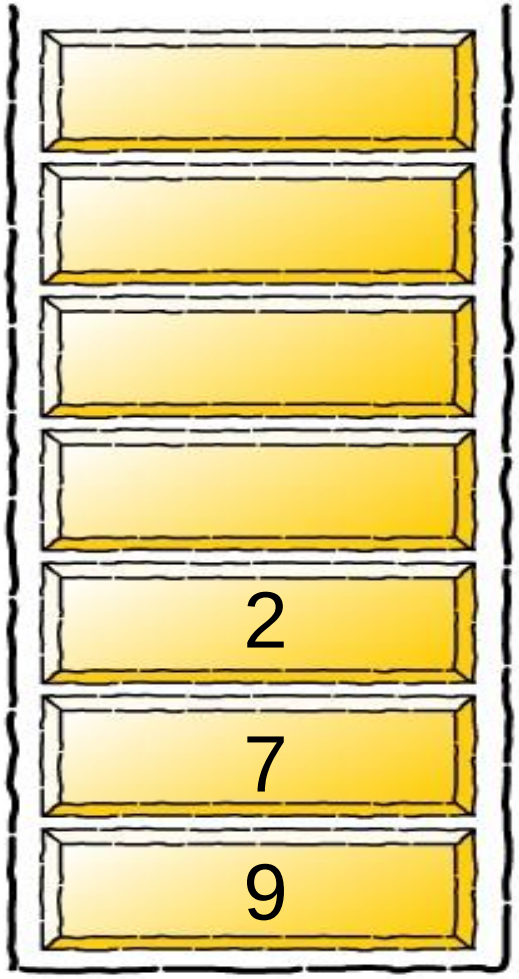
4 5 + 7 2 - *

Operation

Push 7

Evaluation of Postfix expression

Stack



Expression

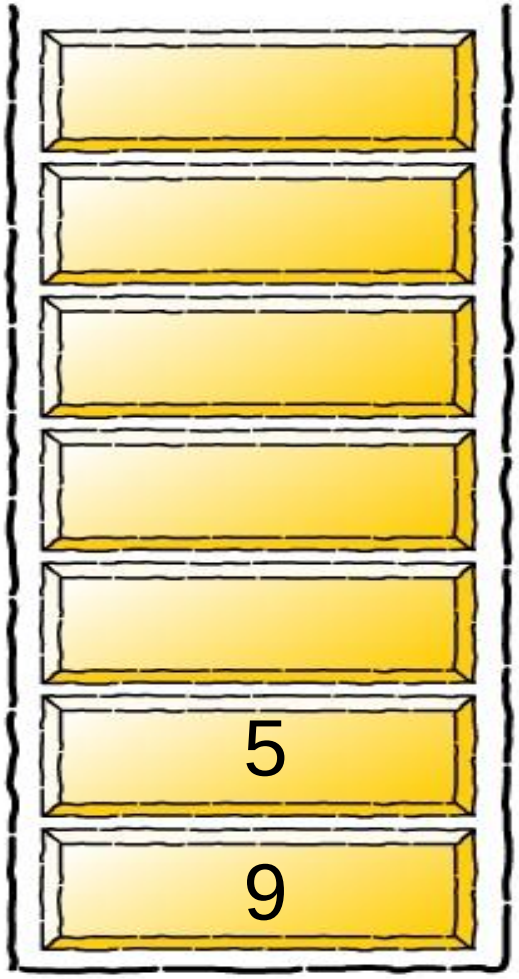
4 5 + 7 2 - *

Operation

Push 2

Evaluation of Postfix expression

Stack



Expression

4 5 + 7 2 - *

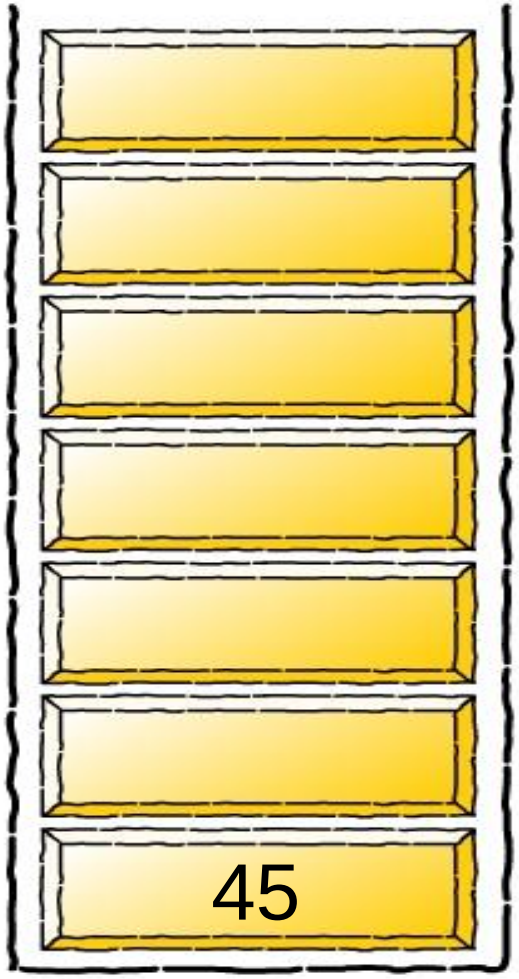
Operation

Pop 2 and 7. Subtract

Push the result (5)

Evaluation of Postfix expression

Stack



Expression

4 5 + 7 2 - *

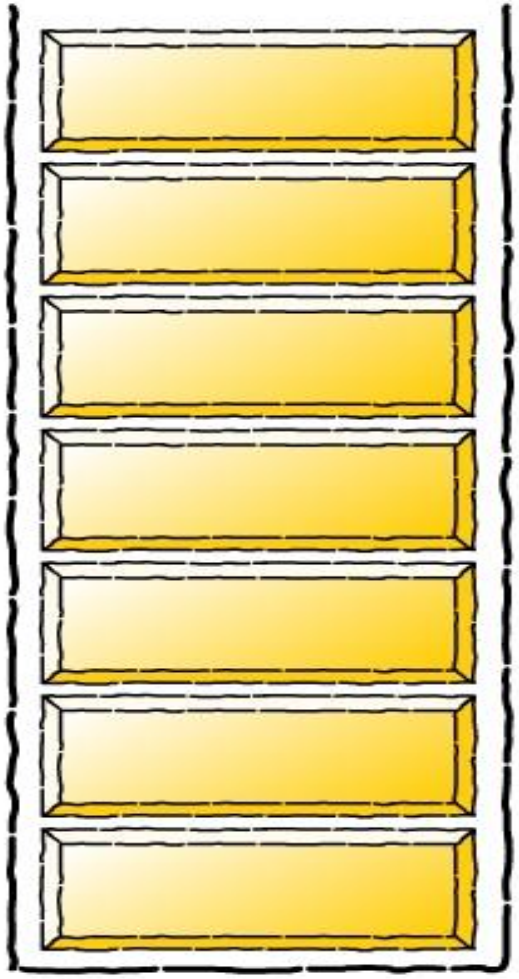
Operation

Pop 5 and 9. Multiply

Push the result (45)

Evaluation of Postfix expression

Stack



Expression

4 5 + 7 2 - *

Operation

Pop 45

Result : 45

An Exercise

Convert the following to postfix and evaluate

- 2*3+16/2-(8-4+3*4^2/2)+12-3*32/2^4-5*3
- 2 3 * 16 2 / + 8 4 - 3 4 2 ^ * 2 / + - 12 + 3 32 * 2 4 ^ / - 5 3 * -

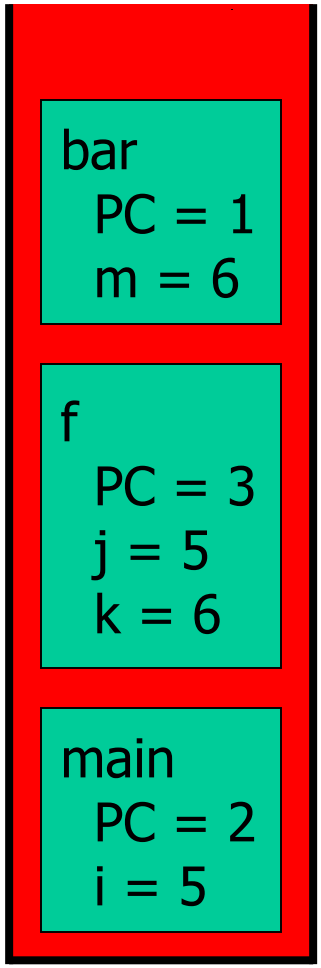
Stacks in Function Calls

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

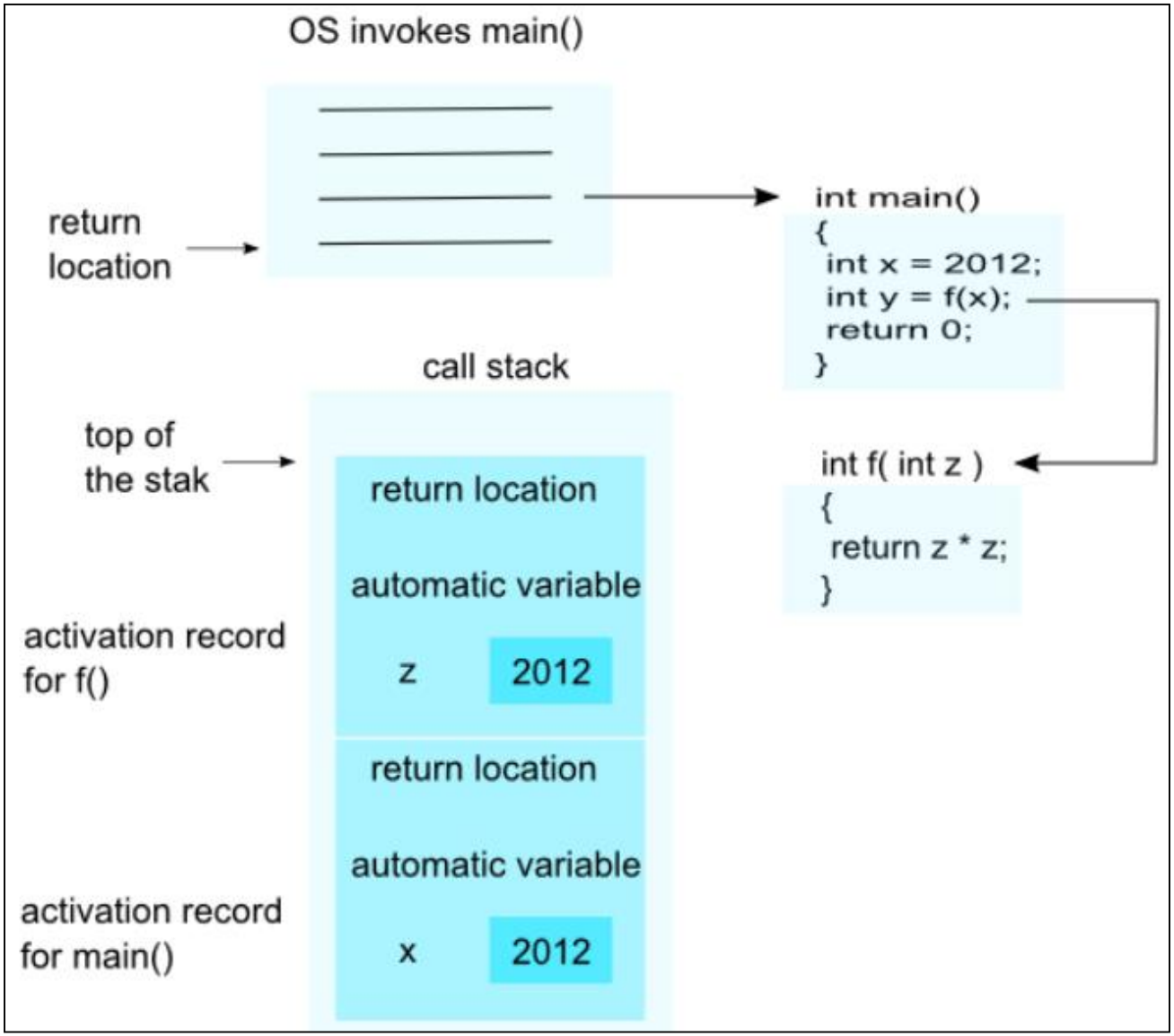
```
main() {
   int  i = 5;
   f(i);
   }

f(int j) {
   int k;
   k = j+1;
   bar(k);
   }

bar(int m) {
   …
   }
```



Stacks in Function Calls



To eliminate the need for direct implementation of recursion

- As recursive function calls require a lot of overhead, it is often the case that recursive algorithms are “unrolled” into non-recursive ones. Since recursive calls are entered/exited in LIFO order the use of stacks to mimic recursion is a natural choice.