

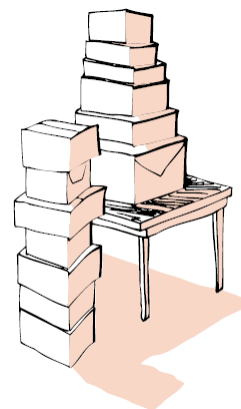
# Data structure and algorithms lab

## STACK & QUEUE

**Lecturers : Cao Tuan Dung**  
**[dungct@soict.hust.edu.vn](mailto:dungct@soict.hust.edu.vn)**  
**Dept of Software Engineering**  
**Hanoi University of Science and Technology**

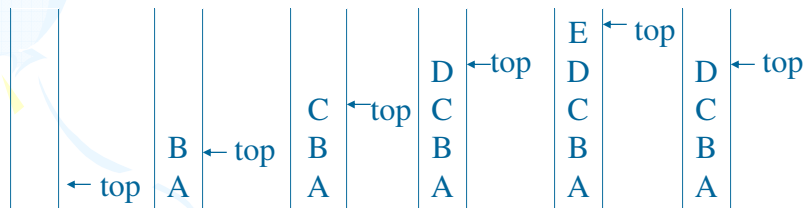
## Topics of this week

- Data structure: Stack
  - Implementation of stack using array
  - Implementation of stack using linked list
- Data structure Queue
  - Implementation of circular queue using array
  - Implementation of queue using linked list
- Exercises on Stack and Queue



## Stack

- A stack is a **linear data structure** which can be **accessed only at one of its ends** for storing and retrieving data.
- A LIFO (Last In First Out) structure



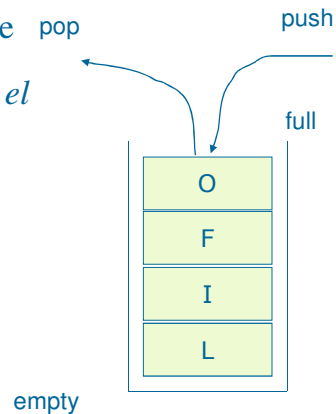
Inserting and deleting elements in a stack

## Applications of Stacks

- Page-visited history in a web browser
- Undo sequence in a text editor
- Chain of function calls in the C++ run-time system
- Infix to postfix conversion
- Postfix expression evaluation
- Parenthesis matching
- HTML tag matching
- Maze solution finding

## Operations on a stack

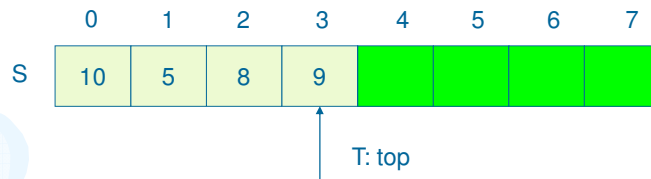
- *initialize(stack)* --- clear the stack
- *isEmpty(stack)* --- check to see if the stack is empty
- *isFull(stack)* --- check to see if the stack is full
- *push(el, stack)* --- put the element *el* on the top of the stack
- *pop(stack)* --- take the topmost element from the stack
- *top(stack)* --- see the value of the topmost element
- How to implement a stack?



## Separate implementation from specification

- **INTERFACE:** specify the allowed operations
- **IMPLEMENTATION:** provide code for operations
- **CLIENT:** code that uses them.
- Could use either array or linked list to implement stack
- Client can work at higher level of abstraction

## Implementation using array



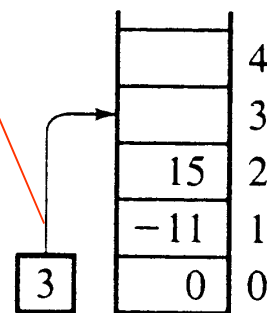
- Each element is stored as an array's element.
- stack is empty: top = 0
- stack is full: top = Max\_Element

## Stack specification (stackarr.h)

```
#include <stdio.h>
#define MAX 50
typedef int Eltype;
typedef Eltype StackType[MAX];
int top;

void initialize(StackType stack);
int isEmpty(StackType stack);
int isFull(StackType stack);
void push(Eltype el, StackType stack);
Eltype pop(StackType stack);
Eltype peek(StackType stack);
```

Top of stack



(a)

## array implementation of stack (stackarr.c)

```

initialize(StackType stack)    push(Etype el, StackType stack)
{
    top = 0;
    {
        if (isFull(stack))
            printf("stack overflow");
        else stack[top++] = el;
    }
}
isEmpty(StackType stack)      Etype pop(StackType stack)
{
    return top == 0;
    {
        if (isEmpty(stack))
            printf("stack underflow");
        else return stack[--top];
    }
}
isFull(StackType stack)
{
    return top == MAX;
}

```

## continue

```

Etype peek(StackType stack){
    if (isEmpty(stack)) {
        printf("stack underflow");
        return -999999;
    }
    else return stack[top-1];
}

```

## Test the stack lib (testStackArr.c)

```
#include "stackarr.h"

int main() {
    int a[6] = {4,7, 1, -9, 26, 13}; int i;
    StackType s; initialize(s);
    for (i=0; i<6; i++) push(a[i],s);
    printf("Pop all elements in stack!\n");
    while (! empty(s)) {
        printf("%4d\n", pop(s));
    }
    return 0;
}
```

## stack implementation using structure

- Implementation (c): stack is declared as a *structure* with two fields: one for storage, one for keeping track of the topmost position

```
#define Max 50
typedef int Eltype;
typedef struct StackRec {
    Eltype storage[Max];
    int top;
};
typedef struct StackRec StackType;
```

## stack implementation using structure

```

initialize(StackType *stack)    push(Etype el, StackType *stack)
{
    (*stack).top=0;
}
isEmpty(StackType stack)
{
    return stack.top == 0;
}
isFull(StackType stack)
{
    return stack.top == Max;
}

if (full(*stack))
    printf("stack overflow");
else (*stack).storage[
    (*stack).top++]=el;

Etype pop(StackType *stack)
{
    if (empty(*stack))
        printf("stack underflow");
    else return
        (*stack).storage[--(*stack).top];
}
  
```

## Compile file with library

You've got stack.h, stack.c and test.c

You need to insert this line:

```
#include "stack.h"
```

into stack.c and test.c

```
gcc -c stack.c
```

```
gcc -c test.c
```

```
gcc -o test.out test.o stack.o
```

## Simple program using stack

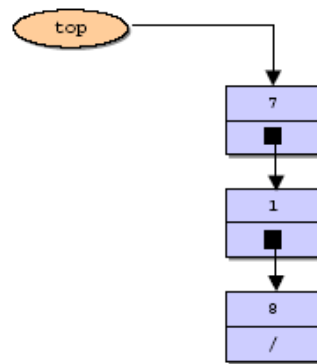
- Write a program that convert a positive integer in decimal form to binary form using library stack you have developed.
- Extend this program to transform from decimal to hexadecimal base.

## Implementation using linked list

- Implementation of stacks using linked lists are very simple
- The difference between a normal linked list and a stack using a linked list is that some of the linked list operations are not available for stacks
- Being a stack we have only one insert operation called push().
  - In many ways push is the same as insert in the front
- We have also one delete operation called pop().
  - This operation is the same as the operation delete from the front



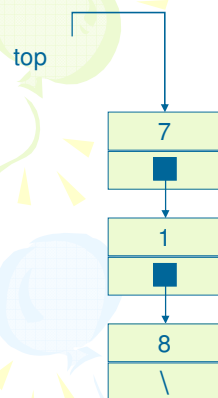
## Pictorial view of stack



```

typedef struct node_t {
    int element;
    struct node_t *next;
} node;
  
```

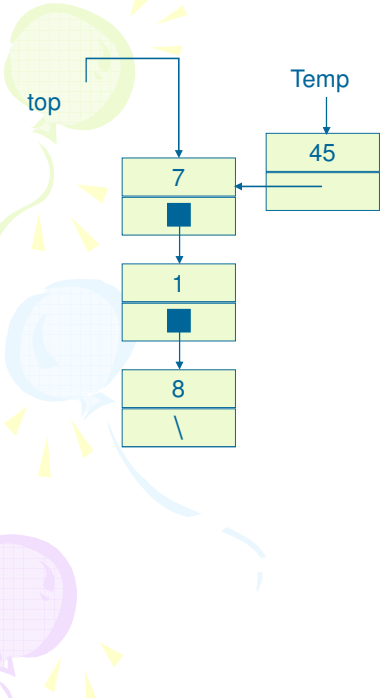
## Push



```

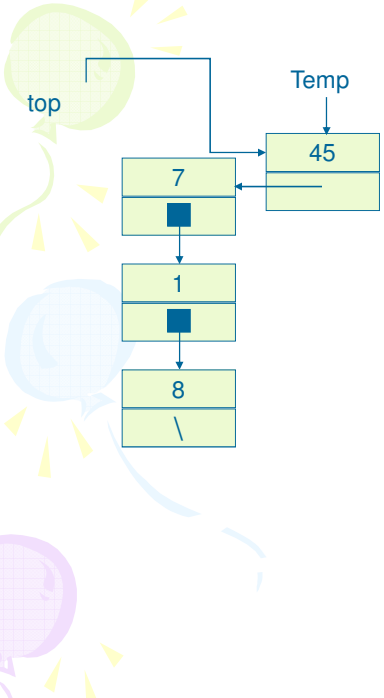
node *push(node *p, int value)
{
    node *temp;
    temp=(node *)malloc(sizeof(node));
    if(temp==NULL) {
        printf("No Memory available\n");
        Error\n";
        exit(0);
    }
    temp->element = value;
    temp->next = p;
    p = temp;
    return(p);
}
  
```

## Push



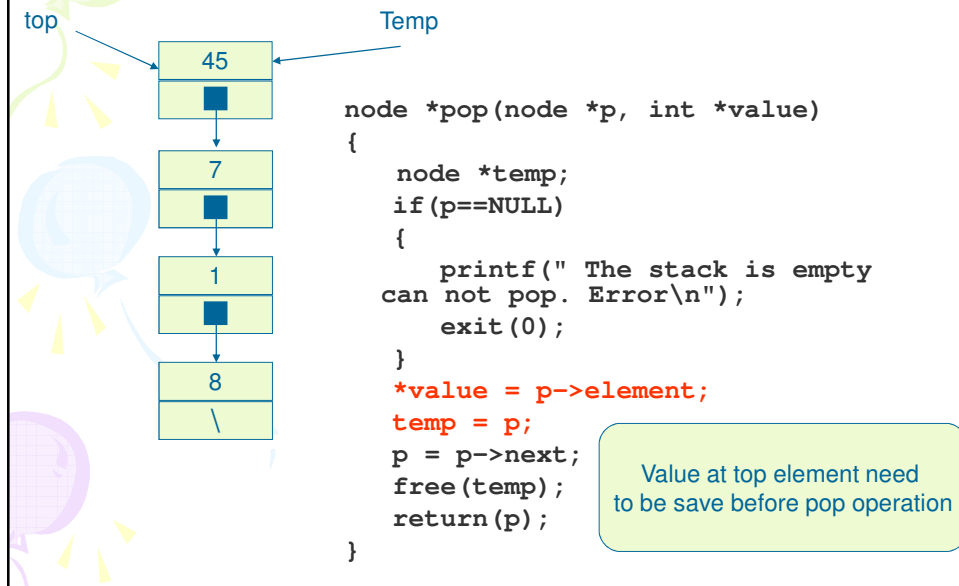
```
node *push(node *p, int value)
{
    node *temp;
    temp=(node *)malloc(sizeof(node));
    if(temp==NULL) {
        printf("No Memory available Error\n");
        exit(0);
    }
    temp->element = value;
    temp->next = p;
    p = temp;
    return(p);
}
```

## Push

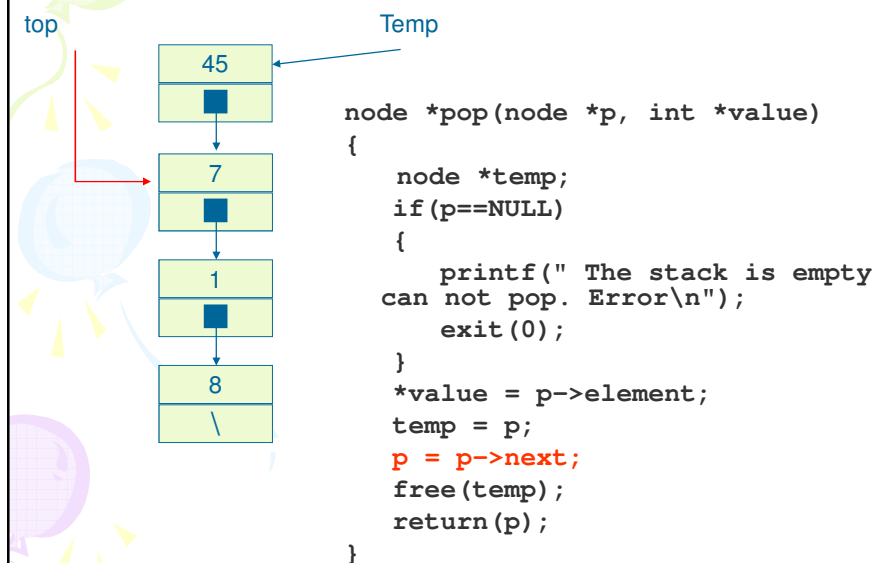


```
node *push(node *p, int value)
{
    node *temp;
    temp=(node *)malloc(sizeof(node));
    if(temp==NULL) {
        printf("No Memory available Error\n");
        exit(0);
    }
    temp->element = value;
    temp->next = p;
    p = temp;
    return(p);
}
```

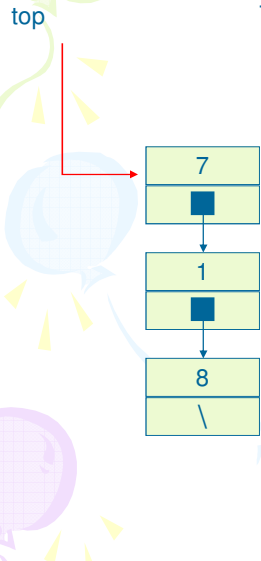
## Pop (linked list)



## Pop (linked list)



## Pop (linked list)



```

node *pop(struct node *p, int
*value)
{
    node *temp;
    if (p==NULL)
    {
        printf(" The stack is empty
can      not pop Error\n");
        exit(0);
    }
    *value = p->element;
    temp = p;
    p = p->next;
    free(temp);
    return (p);
}
  
```

## Using stack in program

```

#include <stdio.h>
#include <stdlib.h>
#include "stacklist.h"
void main()
{
    node *top = NULL;
    int n,value;
    do
    {
        do
        {
            printf("Enter the element
            to be pushed\n");
            scanf("%d",&value);
            top = push(top,value);
            printf("Enter 1 to
            continue\n");
            scanf("%d",&n);
        } while(n == 1);

        printf("Enter 1 to pop an element\n");
        scanf("%d",&n);
        while( n == 1)
        {
            top = pop(top,&value);
            printf("The value popped is
            %d\n",value);
            printf("Enter 1 to pop an
            element\n");
            scanf("%d",&n);
        }
        printf("Enter 1 to continue\n");
        scanf("%d",&n);
    } while(n == 1);
}
  
```

## stacklist.h implementation

- Implement a library supports working with multiple stacks at the same time, using linked structure.

```
typedef struct StackType_t{
    elementType element;
    struct StackType_t *next;
} StackType;

int isEmpty(StackType *top){
    return (top == NULL);
}
```

## function prototypes

```
void push(elementType element, StackType
**top);
```

```
elementType pop(StackType **top);
```

```
elementType top(StackType **top);
```

- Remember to implement freeStack function in your lib.

```
void freeStack(StackType **top)
```



## Exercises

- Test the "stack" type that you've defined in a program that read from user a string, then reverse it.



### Exercise 4.1 Stack using array

- We assume that you make a mobile phone's address book.
- Declare a structure "Address" that can hold at least "name", "telephone number" and "e-mail address".
- Write a program that copies data of an address book from a file to another file using a stack. First, read data of the address book from the file and push them on a stack. Then pop data from the stack and write them to the file in the order of popped. In other words, data read first should be read out last and data read last should be read out first.

## Adding very large numbers

- Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks

2		9		1
3		2		4
7		6		3
8		5		6
	+		=	1

8732    +    5629    =    14361

## Adding very large numbers: detail algorithm

*Read the numerals of the first number and store the numbers corresponding to them on one stack;*

*Read the numerals of the second number and store the numbers corresponding to them on another stack;*

**carry**=0;

*while at least one stack is not empty*

*pop a number from each non-empty stack and add them;*

*push the sum (minus 10 if necessary) on the result stack;*

*store carry in **carry**;*

*push carry on the result stack if it is not zero;*

*pop numbers from the result stack and display them;*



## Homework

- Complete three libraries for stack using:
  - array (stackarr.h)
  - structure (stackstruct.h)
  - linkedlist (stacklist.h)
- Write a program using one among these lib to:
  - a) add very large integer numbers
  - b) subtract very large integer numbers



## Additional Homework

- Study and implement algorithms for multiplying very large integers.
- Hint:
  - You can use naïve algorithm based on basic multiplication that we were taught during school days.



## Homework

- Write a program that simulates the Undo function of editors (Office, Text Editor, ..) as follows. The program uses two stacks: A stack contains integers and a stack contains elements that are strings and have the following menu interface :
  - Add 1 integer to the stack
  - View top
  - Remove from the stack contains integers 1 element
  - Undo
- Hint: To set the undo function, we save the name of the operations just done on the integer stack in string stack (except the View Top operation because it does not change the data). The program based on the last action that was taken to do the opposite.
- For example :
  - PUSH 20 → UNDO = POP
  - POP 15 → UNDO PUSH 15

## Homework – Check for balanced parentheses

- In programming languages (for example, C) the parentheses must be in pairs that are symmetrical and in the correct order. A properly syntactic expression will take the form: {... ([ ] ..) ( ) { } }
- Use stacks to check for balanced parenthesis in the source code of a C program (a file .c). We not only check the opening and closing brackets but also check the ordering of brackets. For an example we can say that the expression "[{ } ( ) { ( ) } ]" it is correct, but "{ [ ] }" it is not correct.
- If the parentheses are not in correct syntax, print the line of code causes the imbalance.

## Check for balanced parentheses

- Hint: Algorithm

- Declare a character stack  $S$ .
- Traverse the expression string  $exp$ .
  - If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
  - If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
- After complete traversal, if there is some starting bracket left in stack then "not balanced"

## Parentheses Matching Algorithm

**Algorithm** BalancedParenCheck ( $C, n$ ):

**Input:** An array  $C$  of  $n$  characters, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

**Output:** **true** if and only if all the grouping symbols in  $C$  match

Let  $S$  be an empty stack

**for**  $i=0$  to  $n-1$  **do**

**if**  $C[i]$  is an opening grouping symbol **then**

    push( $C[i]$ ,  $S$ )

**else if**  $C[i]$  is a closing grouping symbol **then**

**if** isEmpty( $S$ ) **then**

**return false** {nothing to match with}

**if** pop( $S$ ) does not match the type of  $C[i]$  **then**

**return false** {wrong type}

**if** isEmpty( $S$ ) **then**

**return true** {every symbol matched}

**else return false** {some symbols were never matched}

36

## Postfix Notation

- Advantages of postfix notation
  - No need to use parentheses
  - No need to consider precedence of operators
- Two questions
  - How to change infix notation into postfix notation?
  - How to evaluate an expression in postfix notation?

37

## Infix to postfix conversion

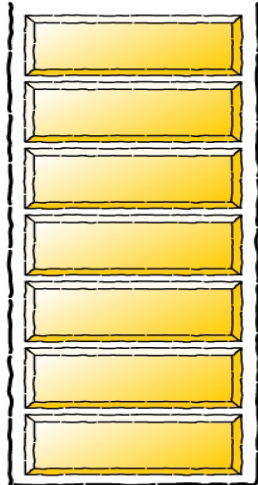
Scan the Infix expression left to right

- If the character  $x$  is an operand
  - Output the character into the Postfix Expression
- If the character  $x$  is a left or right parenthesis
  - If the character is "("
    - Push it into the stack
  - if the character is ")"
    - Repeatedly pop and output all the operators/characters until "(" is popped from the stack.
- If the character  $x$  is a regular operator
  - **Step 1:** Check the character  $y$  currently at the top of the stack.
  - **Step 2:** If Stack is empty or  $y = '('$  or  $y$  is an operator of **lower precedence** than  $x$ , then push  $x$  into stack.
  - **Step 3:** 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 output them until the stack is empty.

## Infix to postfix conversion

Stack



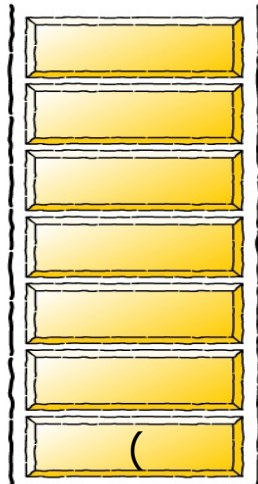
Infix Expression

$(a + b - c) * d - (e + f)$

Postfix Expression

## Infix to postfix conversion

Stack



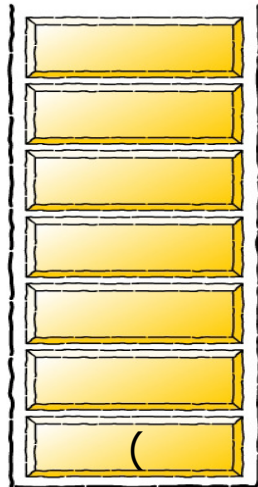
Infix Expression

$a + b - c) * d - (e + f)$

Postfix Expression

## Infix to postfix conversion

Stack



Infix Expression

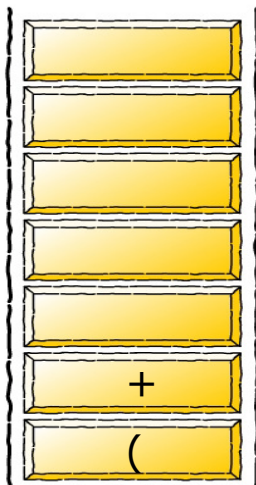
$+ b - c ) * d - ( e + f )$

Postfix Expression

a

## Infix to postfix conversion

Stack



Infix Expression

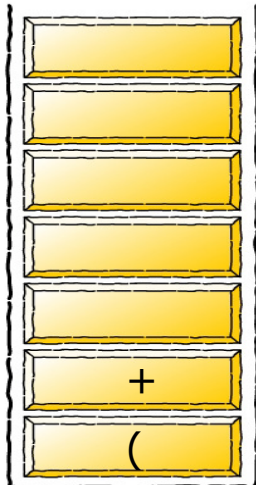
$b - c ) * d - ( e + f )$

Postfix Expression

a

## Infix to postfix conversion

Stack



Infix Expression

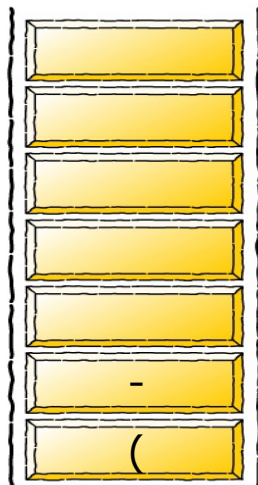
- c ) \* d - ( e + f )

Postfix Expression

a b

## Infix to postfix conversion

Stack



Infix Expression

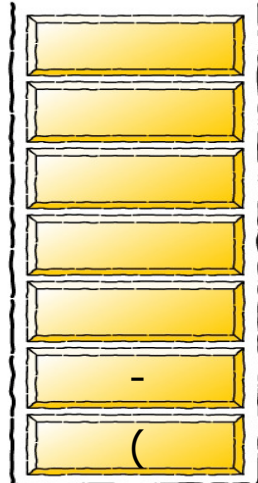
c ) \* d - ( e + f )

Postfix Expression

a b +

## Infix to postfix conversion

Stack



Infix Expression

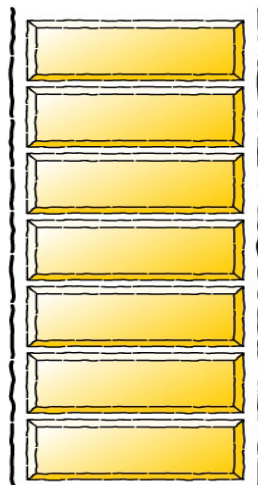
) \* d - ( e + f )

Postfix Expression

a b + c

## Infix to postfix conversion

Stack



Infix Expression

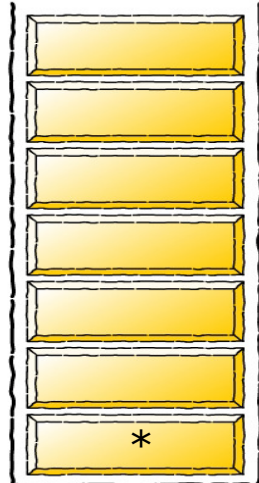
\* d - ( e + f )

Postfix Expression

a b + c -

## Infix to postfix conversion

Stack



Infix Expression

$d - (e + f)$

Postfix Expression

$a b + c -$

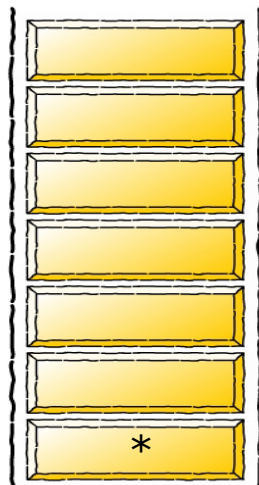
---



---

## Infix to postfix conversion

Stack



Infix Expression

$-(e + f)$

Postfix Expression

$a b + c - d$

---

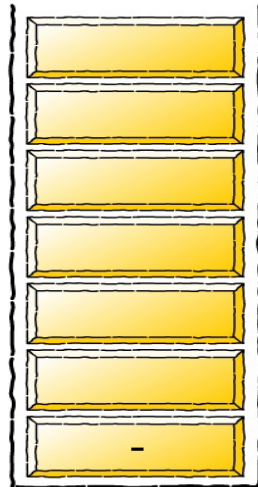


---



## Infix to postfix conversion

Stack



Infix Expression

( e + f )

Postfix Expression

a b + c - d \*

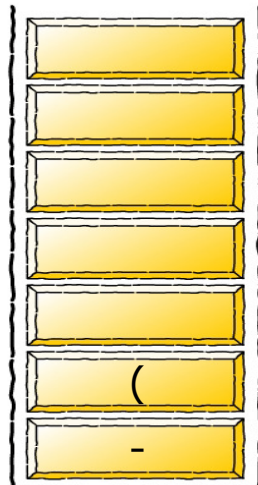
---



---

## Infix to postfix conversion

Stack



Infix Expression

e + f )

Postfix Expression

a b + c - d \*

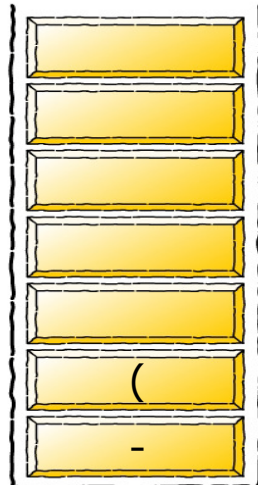
---



---

## Infix to postfix conversion

Stack



Infix Expression

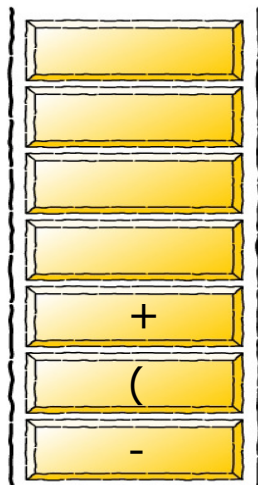
+ f )

Postfix Expression

a b + c - d \* e

## Infix to postfix conversion

Stack



Infix Expression

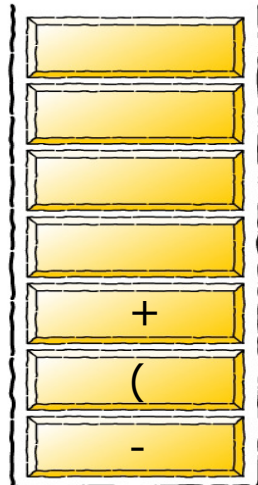
f )

Postfix Expression

a b + c - d \* e

## Infix to postfix conversion

Stack



Infix Expression

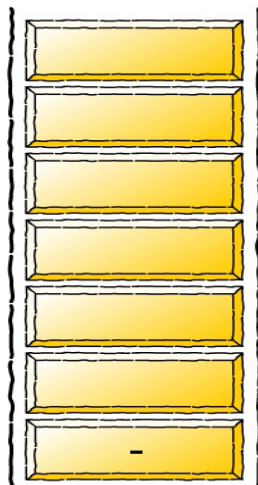
)

Postfix Expression

a b + c - d \* e f

## Infix to postfix conversion

Stack



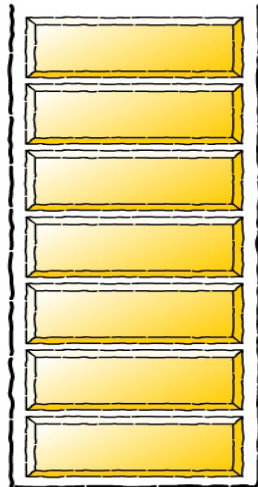
Infix Expression

Postfix Expression

a b + c - d \* e f +

## Infix to postfix conversion

Stack



Infix Expression

Postfix Expression

a b + c - d \* e f +

-

### Exercise 4-2: Conversion to Reverse Polish Notation Using Stacks

- Write a program that converts an expression in the infix notation to an expression in the reverse polish notation. An expression consists of single-digit positive numbers (from 1 to 9) and four operators (+, -, \*, /). Read an expression in the infix notation from the standard input, convert it to the reverse polish notation, and output an expression to the standard output. Refer to the textbook for more details about the Reverse Polish Notation.
- For example,

3+5\*4

is input, the following will be output.

3 5 4 \* +

## Solution

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef char elementType;
#include"stacklist.h"
#define MAX 30
int prior(char operator){
    if (operator == '*' || operator == '/')
        return 2;
    else if (operator == '+' || operator == '-') return 1;
    else if (operator == '(' || operator == ')') return 0;
}
int main()
{
    int i=0, read = 0;
    char v, ele;
    char input[MAX], output[MAX];
    StackType *st;
    printf("Input an infix expression: ");
    scanf("%s", input);
```

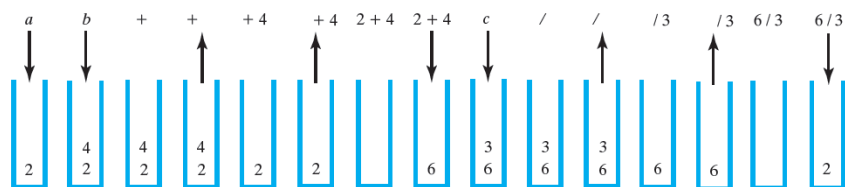
## Solution

```
// Complete the program
/*
.....
*/
```

## Postfix evaluation using stack

- Postfix evaluation can easily be accomplished using a stack to keep track of operands
- As operands are encountered or created (through evaluation) they are pushed on stack
- When operator is encountered, pop two operands, evaluate, push result

59



The stack during the evaluation of the postfix expression  
 $a b + c /$  when  $a$  is 2,  $b$  is 4, and  $c$  is 3

## Postfix expression evaluation

- Write a program that reads any postfix expression involving multiplication and addition of interger.
- For example
- `./posteval 5 4 + 6 * => 54`

## Solution (First version)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef int elementType;
#include "stacklist.h"

int main(){
    StackType *top = NULL;
    char a[100];
    int i, n;
    int b, c;
    printf("Input a postfix
expression of which the
operands are one-digit
intergers: ");
    gets(a);
    n = strlen(a);

    /* Fill in the code here */

    return 0;
}
```

## Solution using union type: polish.h

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#define EMPTY 0
#define FULL 10000
struct elm_t {
    enum {operator, value} kind;
    union {
        char op;
        int val;
    } u;
} elementType;
typedef enum {false, true} boolean;
struct node_t { /* an element on the stack */
    elementType d;
    struct node_t *next;
} node;
```

## Solution: polish.h

```
typedef struct stack_t {
    int cnt; /* a count of the elements */
    elem *top; /* ptr to the top element */
} stack;
boolean isEmpty(stack *stk);
int evaluate(stack *polish);
void fill(stack *stk, char *str);
boolean isFull(const stack *stk);
void initialize(stack *stk);
elementType pop(stack *stk);
void prn_data(data *dp);
void prn_stack(stack *stk);
void push(elementType d, stack *stk);
data top(stack *stk);
```



## Solution: eval.c

```
#include "polish.h"
int evaluate(stack *polish)
{
    elementType d, d1, d2;
    stack eval;
    initialize(&eval);
    while (!isEmpty(polish)) {
        d = pop(polish);
        switch (d.kind) {
            case value:
                push(d, &eval);
                break;
            case operator:
                d2 = pop(&eval);
                d1 = pop(&eval);
                d.kind = value;
                switch (d.u.op) {
                    case '+':
                        d.u.val = d1.u.val + d2.u.val;
                        break;
                    case '-':
                        d.u.val = d1.u.val - d2.u.val;
                        break;
                    case '*':
                        d.u.val = d1.u.val * d2.u.val;
                }
                push(d, &eval);
            }
        }
        d = pop(&eval);
        return d.u.val;
    }
}
```

## Homework

- Use the stack to write an arithmetic calculator program that has a menu interface:
- 1. Enter arithmetic expressions (operands can have many digits, separated by a space between the operators, operands)
- 2. Change to suffix form
- 3. Calculating value (If expression is wrong – inform users)
- 4. Exit