Chapter 8 stack (堆疊)

Speaker: Lung-Sheng Chien

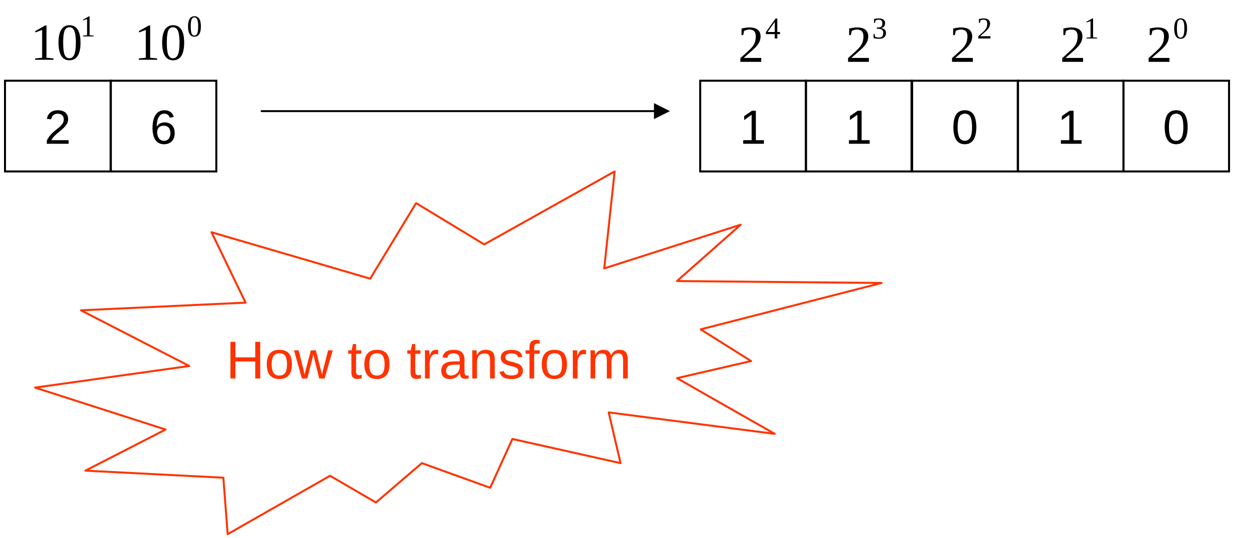
Reference book: Larry Nyhoff, C++ an introduction to data structures

OutLine

- LIFO: from base-10 to base-2
- Array-based stack implementation
- Application 1: railroad switching yard
- Application 2: expression evaluation
  - infix to postfix
  - Reverse Polish Notation

Problem: display the base-2 representation of a base-10 number

$$26 = 2\times 10^1 + 6\times 10^0$$
$$= 16 + 8 + 2$$
$$= 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0$$



Mathematical deduction [1]

$$26 = a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 + \cdots + a_n \times 2^n = \sum_{k=0}^{n} a_k 2^k$$

$$26 \equiv a_0 \pmod 2$$

$$a_0 = 0$$

$$(26 - a_0) = a_1 \times 2^1 + a_2 \times 2^2 + \cdots + a_n \times 2^n$$

$$13 = \frac{1}{2}(26 - a_0) = a_1 + a_2 \times 2^1 + \cdots + a_n \times 2^{n-1}$$

$$13 \equiv a_1 \pmod 2$$

$$a_1 = 1$$

$$6 = \frac{1}{2}(13 - a_1) = a_2 + a_3 \times 2^1 + a_4 \times 2^2 + \cdots + a_n \times 2^{n-2}$$

$$6 \equiv a_2 \pmod 2$$

Mathematical deduction [2]

$$\downarrow a_2 = 0$$

$$3 = \frac{1}{2}(6 - a_2) = a_3 + a_4 \times 2^1 + a_5 \times 2^2 + \cdots + a_n \times 2^{n-3}$$

$$3 \equiv a_3 \pmod 2$$

$$\downarrow a_3 = 1$$

$$1 = \frac{1}{2}(3 - a_3) = a_4 + a_5 \times 2^1 + a_6 \times 2^2 + \cdots + a_n \times 2^{n-4}$$

$$1 \equiv a_4 \pmod 2$$

$$\downarrow a_4 = 1$$

$$0 = \frac{1}{2}(1 - a_4) = a_5 + a_6 \times 2^1 + \cdots + a_n \times 2^{n-5}$$

$$0 = a_5 = a_6 = \cdots = a_n$$

stack: last-in-first-out (LIFO)

computation order

$$(a_0 = 0) \to (a_1 = 1) \to (a_2 = 0) \to (a_3 = 1) \to (a_4 = 1) \to (a_5 = 0)$$

display order

$$(a_4 = 1) \to (a_3 = 1) \to (a_2 = 0) \to (a_1 = 1) \to (a_0 = 0)$$

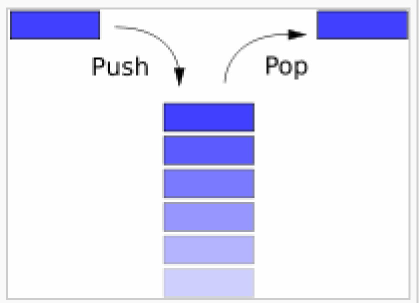
2^4	2^3	2^2	2^1	2^0
1	1	0	1	0
a_4	a_3	a_2	a_1	a_0

Last In in the computation order is *First Out* in the display order

We call "*stack*" as a kind of data structure (資料結構)

http://en.wikipedia.org/wiki/Stack_(data_structure)
http://en.wikipedia.org/wiki/Stack

- a **stack** is an abstract data type and data structure based on the principle of *Last In First Out* (*LIFO*)
- Stack machine: Java Virtual Machine
- Call stack of a program, also known as a function stack, execution stack, control stack, or simply the stack
- Stack allocation in MSDN library
- Application: Reverse Polish Notation, Depth-First-Search



Simple representation of a stack

OutLine

- LIFO: from base-10 to base-2
- Array-based stack implementation
- Application 1: railroad switching yard
- Application 2: expression evaluation
  - infix to postfix
  - Reverse Polish Notation

Stack container

- Collection of data elements (data storage)
  an *ordered* collection of data items that can be accessed at only one end, called the *top* of the stack

- Basic operations (methods)
  - construct a stack (empty stack)
  - *empty*: check if stack is empty
  - *top*: retrieve the top element of the stack
  - *push*: add an element at the top of the stack
  - *pop*: remove the top element of the stack

Requirement of stack

integrate into structure ***stack***

- ***stackEle***: data type
- type of physical storage: array, linked-list
- ordered mechanism: depends on physical storage
- index to top element in the stack

Methods of structure ***stack***

- *stack* stack_init*(*void*)
- *int empty*(*stack**)
- ***stackEle*** *top*(*stack**)
- *void push*(*stack**, ***stackEle***)
- *void pop*(*stack**)

Array-based stack: header file

stack.h

```c
#ifndef STACK_H
#define STACK_H

#define STACK_CAPACITY  5  // maximum size of stack
typedef int stackEle ; // integer stack

typedef struct{
    stackEle myArray[ STACK_CAPACITY ] ;
    int myTop ;
} stack ;

// construct (initialize) an empty stack
stack* stack_init( void ) ;

// return 1  if stack is empty and
//        0  otherwise
int  empty( stack* ) ;

// retrieve top element of the stack
stackEle top( stack* ) ;

// add an element at the top of the stack
void  push( stack*, stackEle ) ;

// remove the top element of the stack
void  pop( stack* ) ;

#endif // STACK_H
```

Type of physical storage (myArray)

index to top element in the stack (myTop)

Methods of structure *stack* (stack_init, empty, top, push, pop)

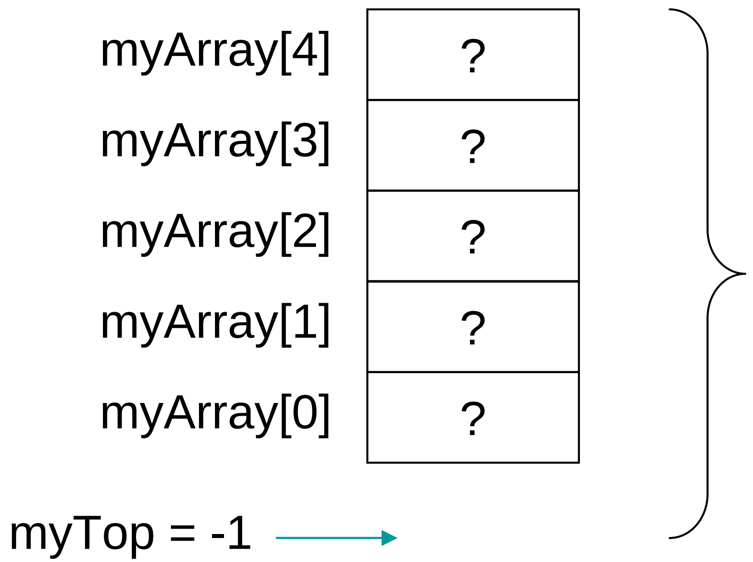
Question: what is "ordered mechanism" ?

Array-based stack: method [1]

stack.cpp

```
#include "stack.h"

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

// construct (initialize) an empty stack
stack*  stack_init( void )
{
    stack* s = (stack*) malloc( sizeof(stack) ) ;
    assert( s ) ;
    s->myTop = -1 ; // stack is empty
    return s ;
}
```

Set stack to empty is essential, or error occurs when do *push(), pop()* or *top()*



data encapsulation (資料隱藏)

You can change name of array or index “myTop” without notifying user.

Array-based stack: method [2]

stack.cpp

```
// return 1  if stack is empty and
//          0  otherwise
int  empty( stack* s )
{
    assert( s ) ;
    if ( 0 > s->myTop ) {
        return 1 ;
    }else{
        return 0 ;
    }
}

// retrieve top element of the stack
stackEle top( stack* s )
{
    assert( s ) ;
    if ( 0 > s->myTop ){
        printf("Error: stack is empty -- no value can be reported \n");
        exit(1) ;
    }
    return s->myArray[s->myTop] ;
}
```

Question 1 : what is purpose of "assert(s)" ?

Question 2: why is evaluation order of "*s->myArray[s->myTop]* " ?

Precedence and Associativity of C Operators

Symbol1	Type of Operation	Associativity
[ ] ( ) . -> postfix ++ and postfix	Expression	Left to right

Array-based stack: method [3]

stack.cpp

```
// add an element at the top of the stack
void  push( stack* s, stackEle val)
{
    assert( s ) ;
    s->myTop++ ;
    if ( STACK_CAPACITY < s->myTop ){ // chekc if overflow
        printf("Error: stack is full -- can't add new value \n");
        exit(1) ;
    }
    s->myArray[s->myTop] = val ;
}

// remove the top element of the stack
void  pop( stack* s)
{
    assert( s ) ;
    if ( 0 > s->myTop ){ // check if underflow
        printf("Error: stack is empty -- can't remove a value \n");
        exit(1) ;
    }
    s->myTop-- ; // remove top element
}
```

Upper limit of stack: size is fixed by *STACK_CAPACITY*

Lower limit of stack: empty or not

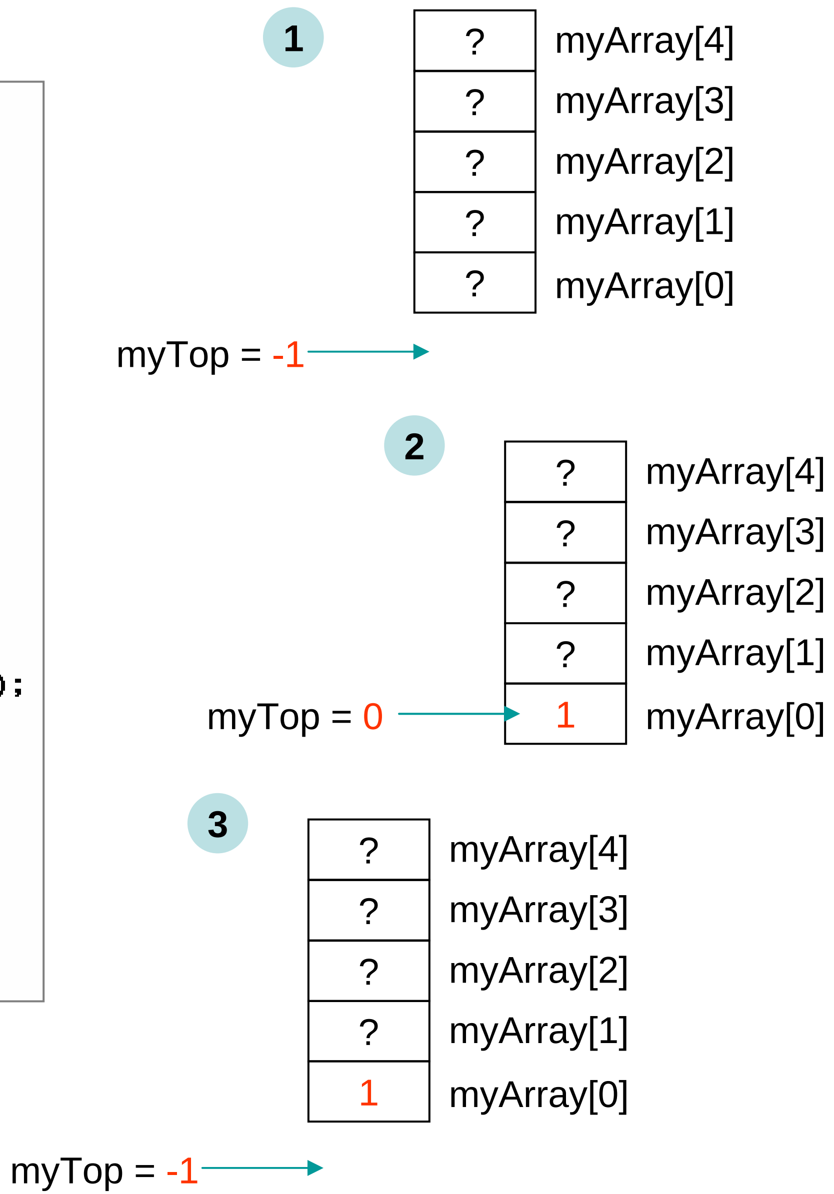
ordered mechanism

Question: maximum size of stack is limited by symbolic constant *STACK_CAPACITY,* can you solve this constraint?

Array-based stack: driver

main.cpp

```
#include "stack.h"

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

int main( int argc, char* argv[] )
{
    stack *mystack = stack_init() ;   1

    if ( empty(mystack) ){
        printf("mystack is empty\n");
    }else{
        printf("Error: mystack is NOT empty\n");
        exit(1) ;
    }

2   push( mystack, 1 ) ;
    printf("top element is %d \n", top(mystack) );

3   pop( mystack ) ;
    if ( !empty(mystack) ){
        printf("Error: mystack is NOT empty\n");
        exit(1) ;
    }

    return 0 ;
}
```



Pro and cons: array-based tack

- pro (in favor of)
  - easy to implement
  - ordered mechanism is natural
- con (contra)
  - maximum size is limited by *STACK_CAPACITY*
  - type of stack element is fixed to only one type
  - type of stack element must be primitive
  - user must call *stack_init()* explicitly, or fetal error occurs

Exercise

- *write a driver to test all methods and constraints in array-based stack*
- *do base-10 to base-2 transformation by array-based stack*
- *modify array-based stack such that maximum size is not limited by STACK_CAPACITY*
- *implement stack by linked-list*
- *how to solve "type of stack element must be primitive"*
- how *to allow more than two stacks of different type in a program*

OutLine

- LIFO: from base-10 to base-2
- Array-based stack implementation
- Application 1: railroad switching yard
- Application 2: expression evaluation
  - infix to postfix
  - Reverse Polish Notation

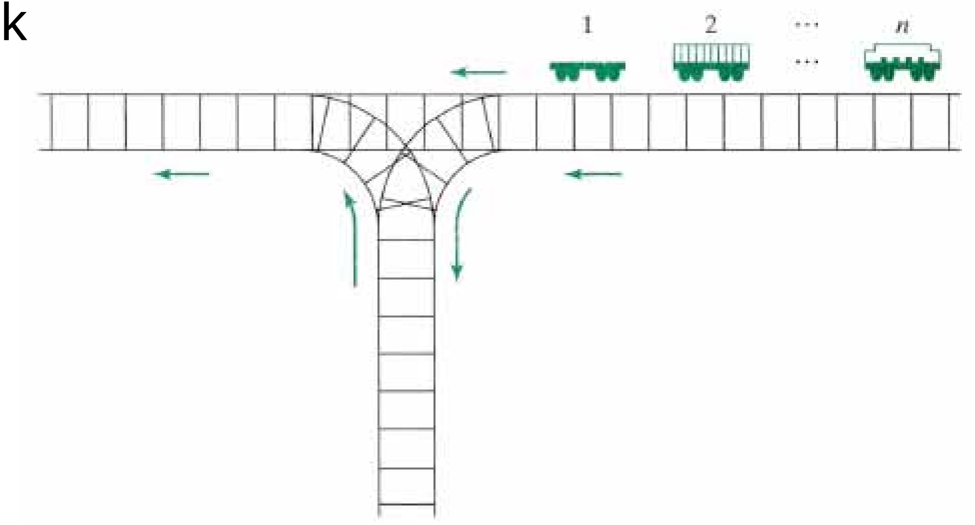
Application 1: railroad switching yard

- Railroad cars numbered 1,2,…,n on the right track are to be permuted and moved along on the left track.

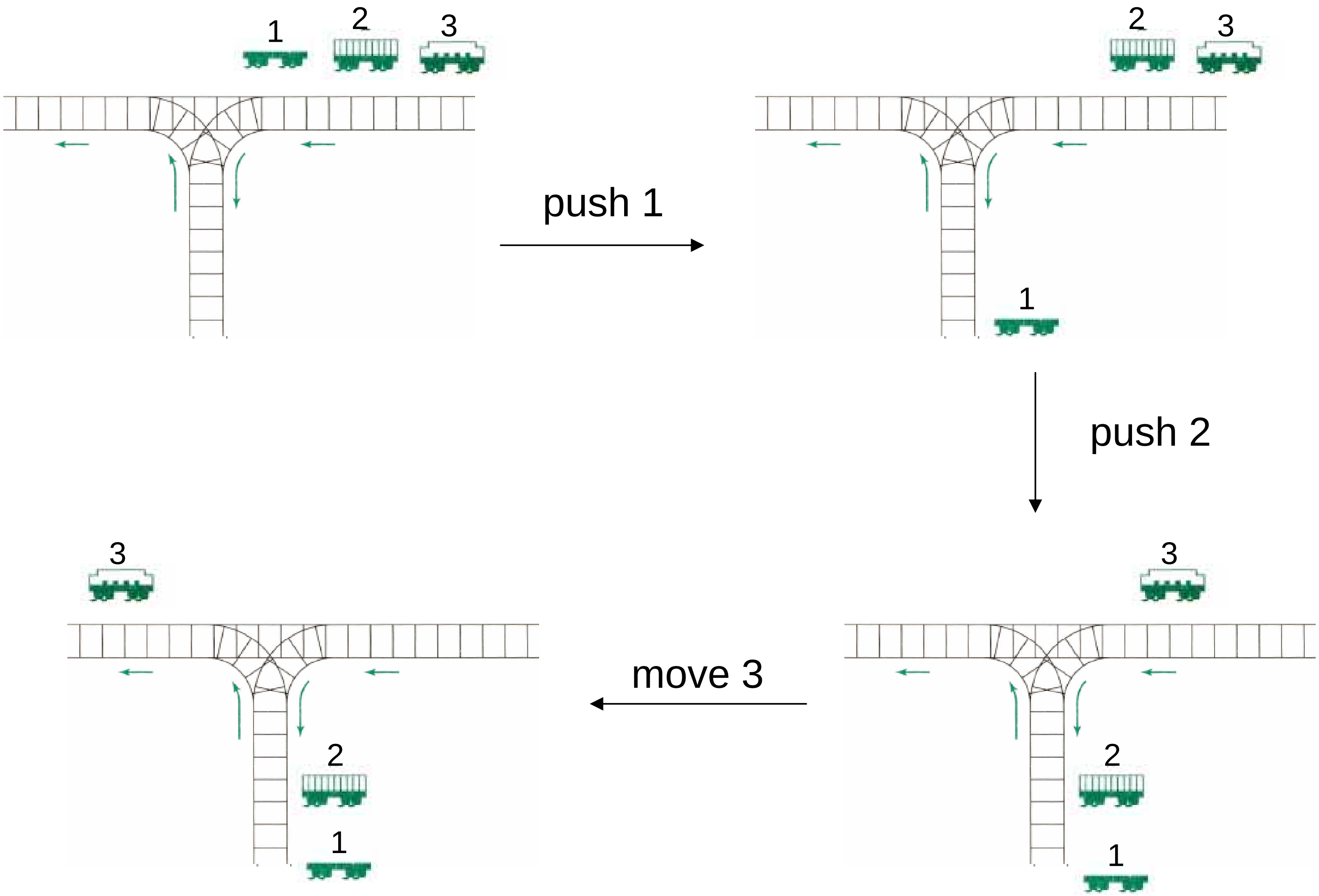
- A car may be moved directly onto the left track, or it may be shunted onto the siding to be removed at a later time and placed on the left track.

- The siding operates like a stack
  - push: move a car from the right track onto the siding

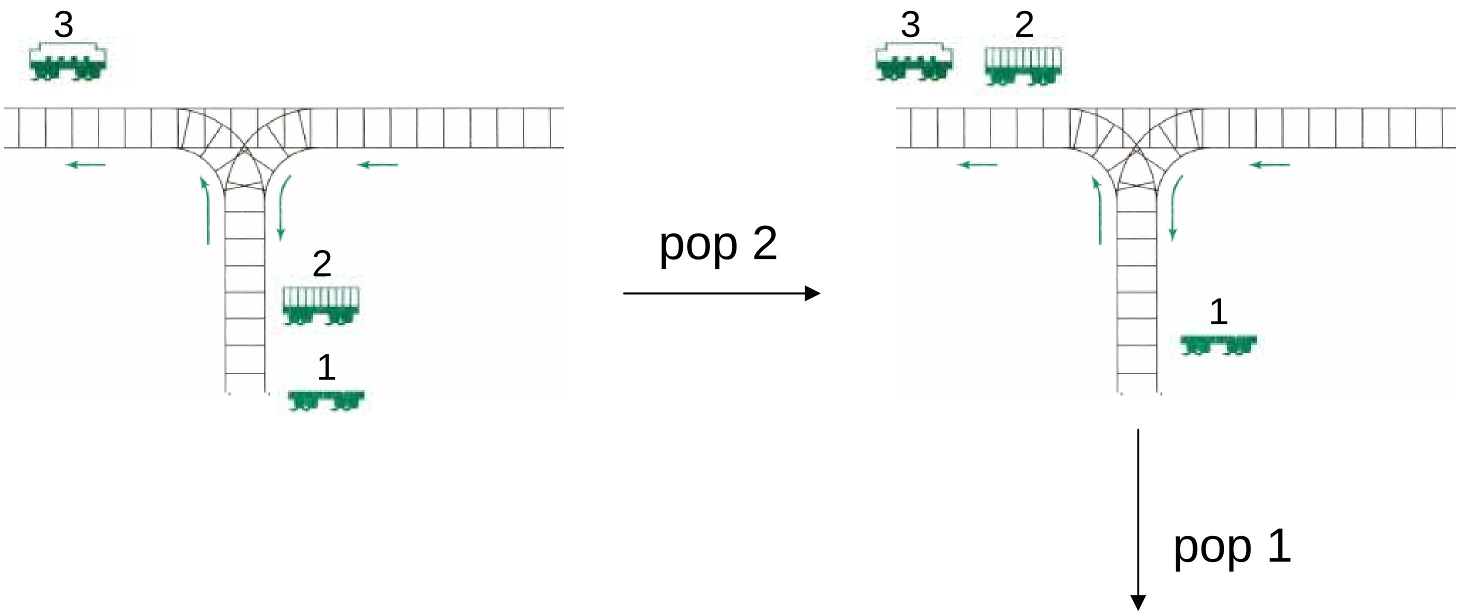
  -pop: move the “top” car from the siding onto the left track

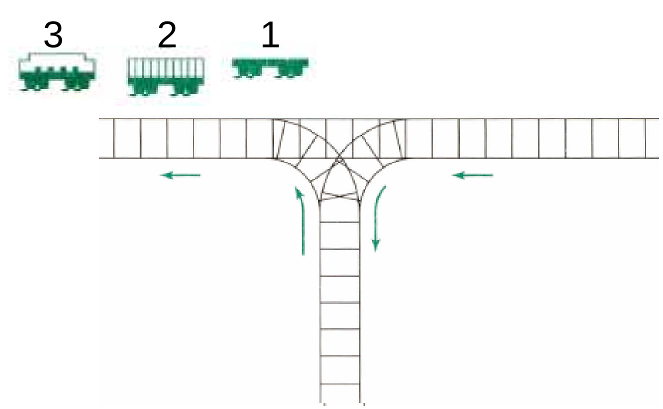


n=3, find all possible permutatoin of cars that can be obtained by a sequence of these operation



n=3, find all possible permutation of cars that can be obtained by a sequence of these operation





Hence *321* is a possible permutation

n=3, find all possible permutatoin of cars that can be obtained by a sequence of these operation

permutation	Operation sequence
123	
132	
213	
231	
312	
321	push 1, push 2, move 3, pop 2, pop 1

n=4, find all possible permutatoin of cars

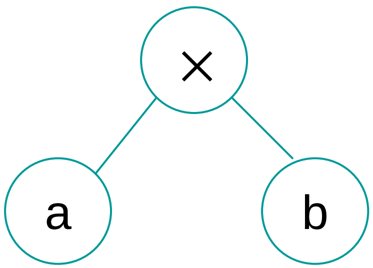
permutation	Operation sequence	permutation	Operation sequence
1234		3124	
1243		3142	
1324		3214	
1342		3241	
1423		3412	
1432		3421	
2134		4123	
2143		4132	
2314		4213	
2341		4231	
2413		4312	
2431		4321	

OutLine

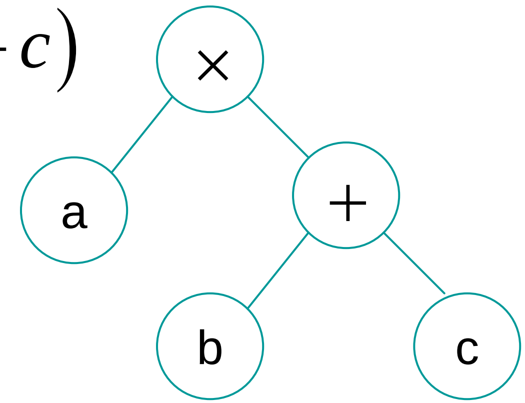
- LIFO: from base-10 to base-2
- Array-based stack implementation
- Application 1: railroad switching yard
- Application 2: expression evaluation
  - infix to postfix
  - Reverse Polish Notation

expression tree

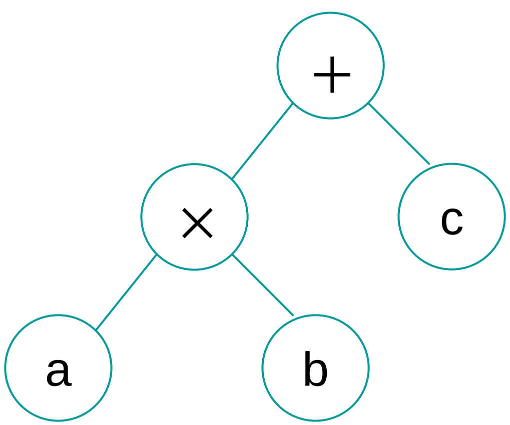
$$a \times b$$



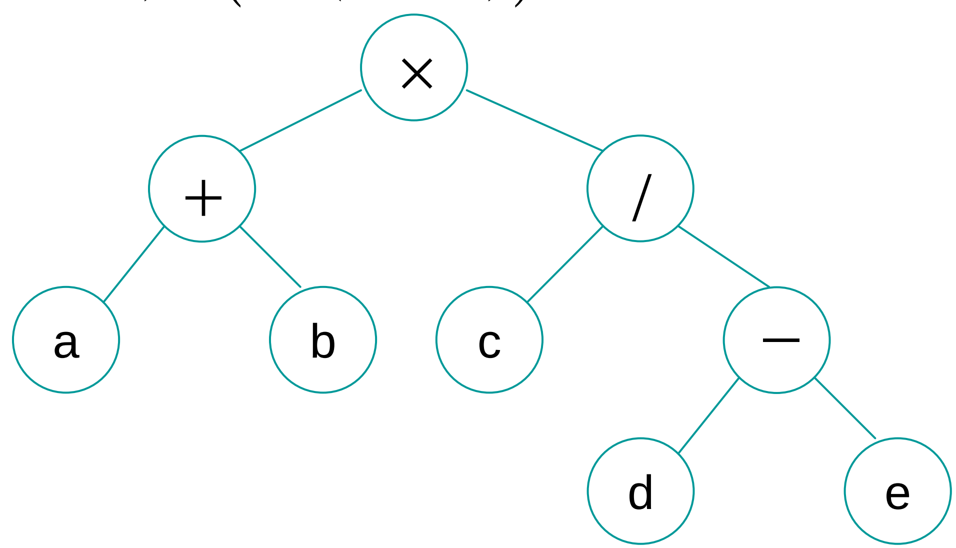
$$a \times (b + c)$$



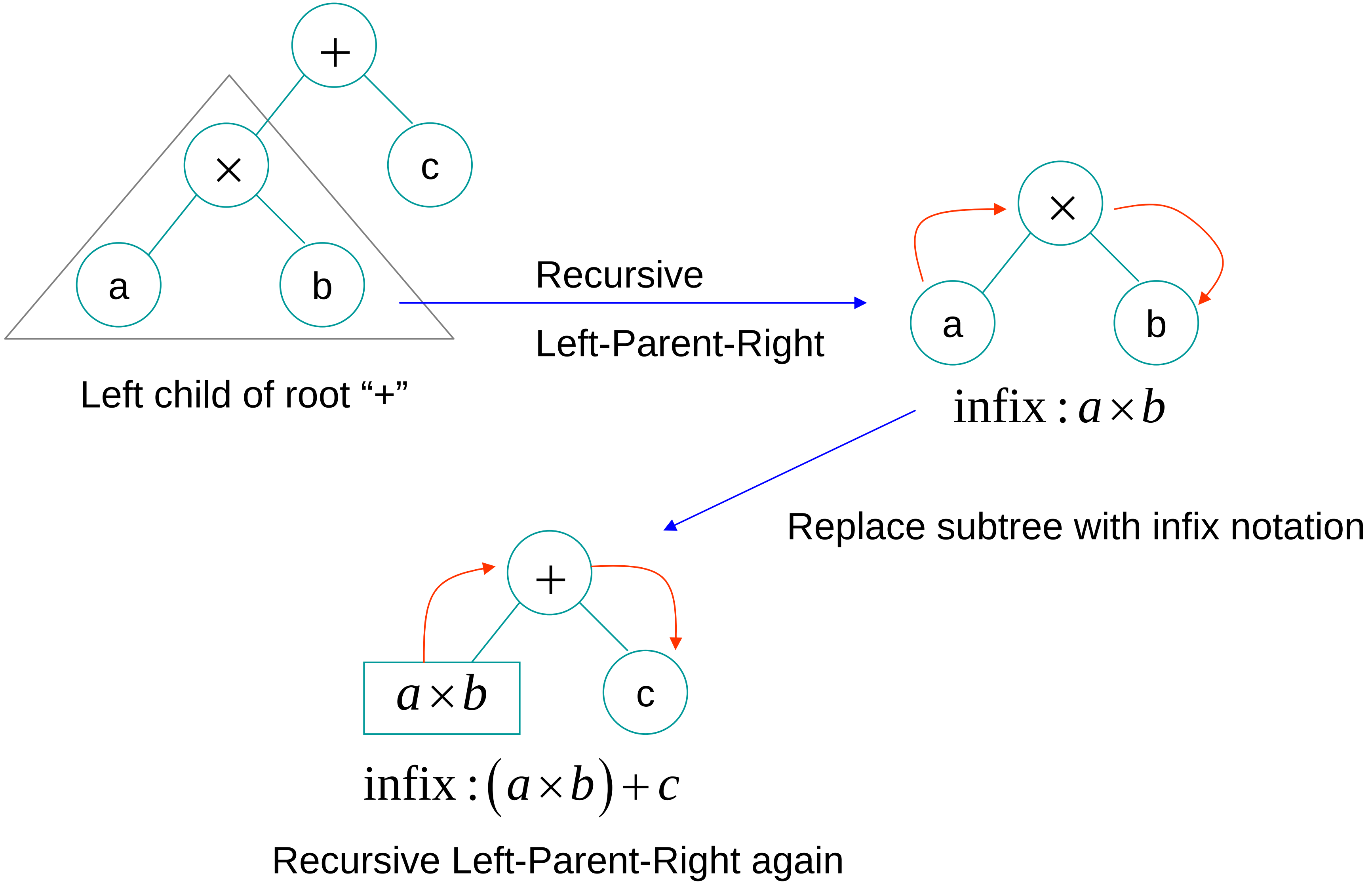
$$a \times b + c$$



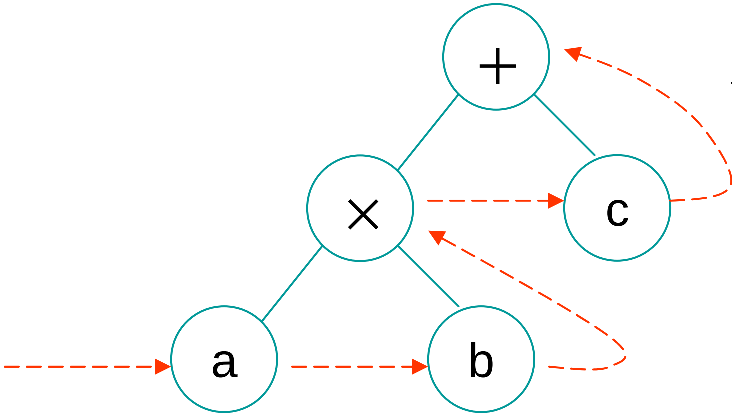
$$(a + b) \times (c / (d - e))$$



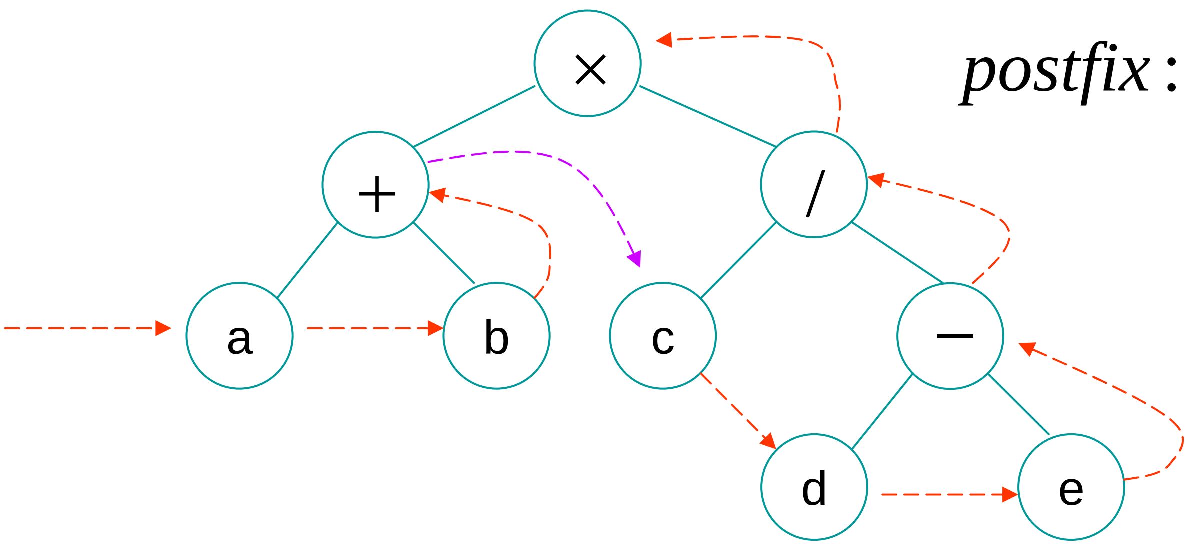
Infix notation: Left-Parent-Right order



postfix notation: Left-Right-Parent order



$$postfix : ab \times c +$$

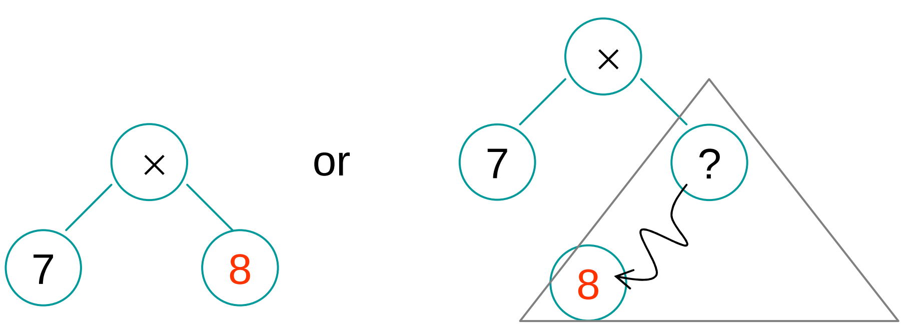


$$postfix : ab + cde - / \times$$

convert infix to postfix [1]

expression	stack	output	comments
$7\times8-(2+3)$	← top	7	Display **7**
$\times8-(2+3)$	$\times$ ← top	7	Push * since stack is empty
$8-(2+3)$	$\times$ ← top	78	Display **8**

so far, we cannot say that **8** is right child of operator * or left child of other operator



convert infix to postfix [2]

expression	stack	output	comments
$8-(2+3)$	$\times$ ← top	78	display **8**
$-(2+3)$	(empty) ← top	$78\times$	pop * and display it since precedence of * is higher than –
	$-$ ← top	$78\times$	push –
$(2+3)$	$($ ← top, $-$	$78\times$	push (since (is delimiter of sub-expression
$2+3)$	$($ ← top, $-$	78×2	display **2**

convert infix to postfix [3]

expression	stack	output	comments
+3)	+ ← top (−	78×2	push + since (is a delimiter of sub-expression, not arithmetic operator, or we can say precedence of (is lowest.
3)	+ ← top (−	78×23	display 3 so far, we cannot say that 3 is right child of operator + or not
)	(← top −	78×23+	pop + and display + since right delimiter of sub-expression is reached
	− ← top	78×23+	pop (, sub-expression is exhausted

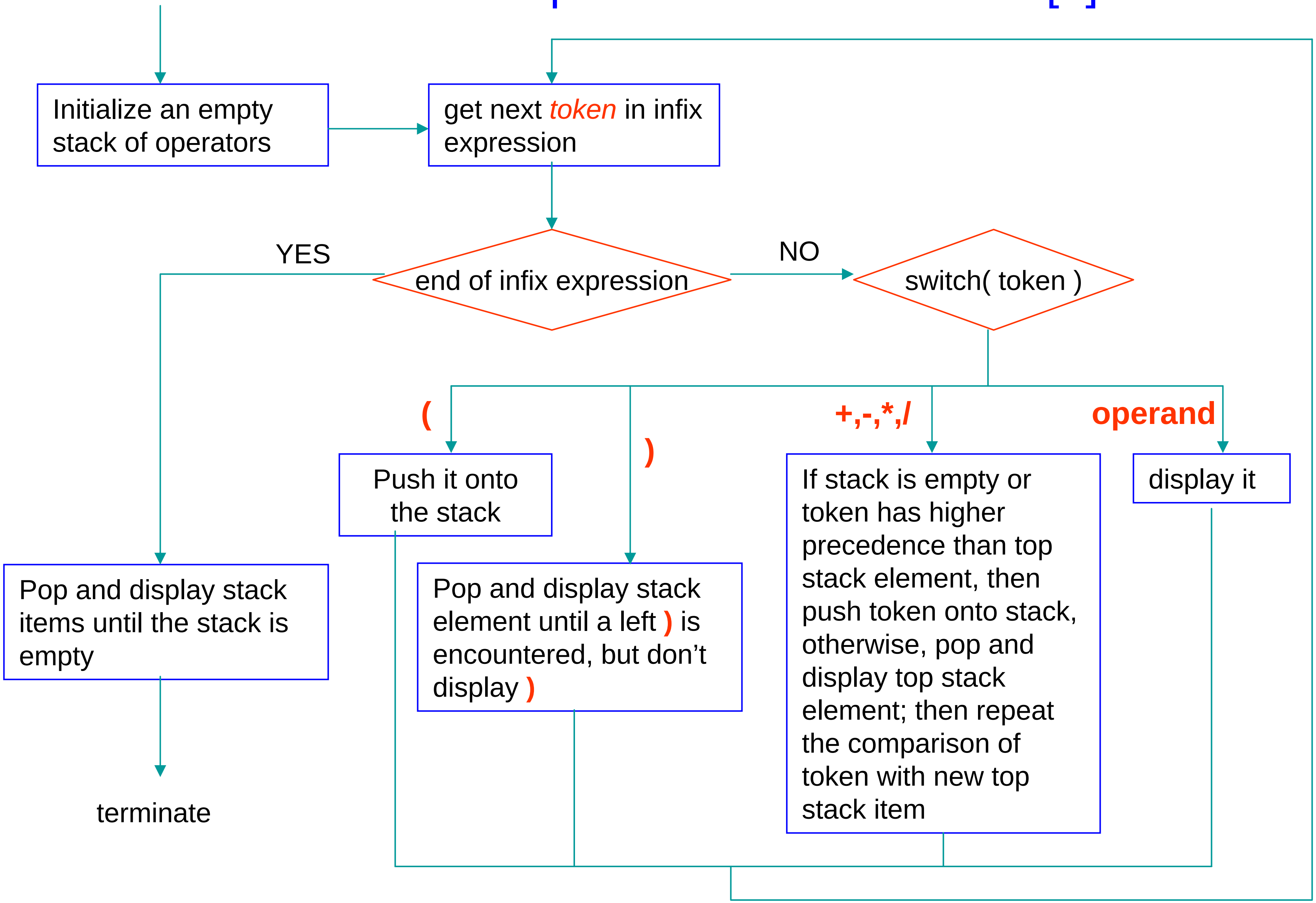
convert infix to postfix [4]

expression	stack	output	comments
	$-$ ← top	$78\times 23+$	pop **(**, sub-expression is exhausted
no token	← top	$78\times 23+-$	No token is read, it means that right child of **–** is exhausted, so pop **–** and display it.

$$postfix: 78\times 23+-$$

Question: What is general procedure?

convert infix to postfix: flow chart [5]



convert infix to postfix: [6]

main.cpp

```
#include "stack.h"

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

#define  MAXBUFFER     128

void  RPN( char* expr, char* RPNexpr ) ;

int main( int argc, char* argv[] )
{
//   char expr[] = "7*8-(2+3)" ;
     char expr[] = "(a + b)*(c /(d -e))" ;
     char RPNexpr[MAXBUFFER] ;

     RPN( expr, RPNexpr ) ;

     printf("RPNexpr=%s\n", RPNexpr );

     return 0 ;
}
```

stack.h

```
#ifndef STACK_H
#define STACK_H

#define STACK_CAPACITY  5  // maximum size of stack
typedef int stackEle ; // integer stack

typedef struct{
    stackEle myArray[ STACK_CAPACITY ] ;
    int myTop ;
} stack ;

// construct (initialize) an empty stack
stack* stack_init( void ) ;

// return 1  if stack is empty and
//        0  otherwise
int  empty( stack* ) ;

// retrieve top element of the stack
stackEle top( stack* ) ;

// add an element at the top of the stack
void  push( stack*, stackEle ) ;

// remove the top element of the stack
void  pop( stack* ) ;

#endif // STACK_H
```

Assumption: every token is a non-space character

RPN.cpp

```
#include "stack.h"

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

// assume token is a non-space character
void  RPN( char* expr, char* RPNexpr )
{
    assert( expr ) ; assert( RPNexpr ) ;
    int i ;
    char token, topToken ;
    stack  *opStack = stack_init() ;
    int n = strlen(expr) ;

    for( i = 0 ; i < n ; i++ ){
        token = expr[i] ;
        if ( ' ' == token ) { continue ; }
// token is a non-space character
        switch( token ){
        case '(' :
// push it onto the stack
            push( opStack, token ) ;
            break ;

        case ')' :
// pop and display stack element until a left ')'
// is encountered, but don't display ')'
            while(1){
                assert( !empty(opStack) ) ;
                topToken = top( opStack ) ;
                pop( opStack ) ;
                if ( '(' == topToken ) { break ; }
                *RPNexpr++ = ' ' ;
                *RPNexpr++ = topToken ;
            }
            break ;
```

```
        case '+' : case '-' : case '*' : case '/' :
            while(1){
// if (1) stack is empty or
//    (2) top stack element is '(', delimiter of sub-expression
//    (2) token has higher precedence than top stack element
// then push token onto stack
                if ( empty(opStack) ||
                     ( '(' == (topToken = top(opStack)) ||
                     (('*' == token    ) || ('/' == token    )) &&
                     (('+' == topToken) || ('-' == topToken))
                   ){
                    push( opStack, token ) ;
                    break ;
                }else{
// otherwise, pop and display top stack element
                    topToken = top(opStack) ;
                    pop( opStack ) ;
                    *RPNexpr++ = ' ' ;
                    *RPNexpr++ = topToken ;
                }
            }// Repeat the comparison with new top stack item
            break ;

        default: // operand
// display it
            *RPNexpr++ = ' ' ;
            *RPNexpr++ = token ;
            break ;
        }//switch( token)
    }// for each token
// pop remaining operations on the stack
    while(!empty(opStack)){
        topToken = top( opStack ) ;
        pop( opStack ) ;
        if ( '(' == topToken ){
            printf("Error in infix expression \n");
            exit(1) ;
        }else{
            *RPNexpr++ = ' ' ;
            *RPNexpr++ = topToken ;
        }
    }// for each top stack element
    *RPNexpr = '\0' ; // terminating character
}
```

Exercise

- Implement function RPN and test it
- We assume that token is a non-space character in first version of function RPN, remove this assumption, consider token as an identify or integer or double.
  for example:
  (delta + 5)/z – y
  3.75 * z / pi
- We only take binary operator in our example, how to deal with unary operator, or in general, a function with *N* arguments.
  for example
  max(add(x, y) + c, d)
  5.0 + sin(7.2 * cos(y))

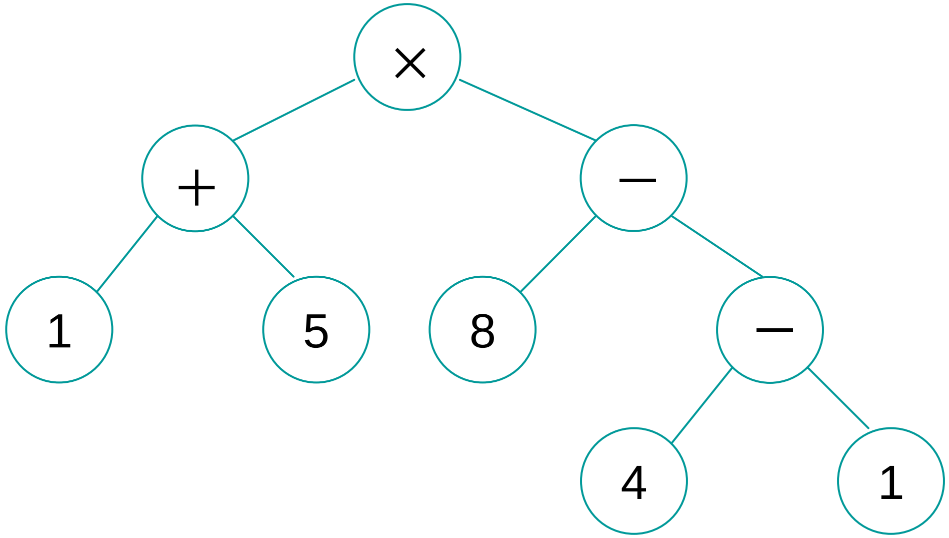
OutLine

- LIFO: from base-10 to base-2
- Array-based stack implementation
- Application 1: railroad switching yard
- Application 2: expression evaluation
  - infix to postfix
  - Reverse Polish Notation

Reverse **P**olish **N**otation: postfix order

- Precedence of multiplication is higher than addition, we need parenthesis to guarantee execution order. However in the early 1950s, the Polish logician Jan Lukasiewicz observed that parentheses are not necessary in postfix notation, called RPN (Reverse Polish Notation).

- The Reverse Polish scheme was proposed by F. L. Bauer and E. W. Dijkstra in the early 1960s to reduce computer memory access and utilize the stack to evaluate expressions .



infix: $(1+5)\times(8-(4-1))$

$postfix:\ 15+841--\times$

parenthesis free

Evaluate **RPN** expression [1]

$1\ 5\ +\ 8\ 4\ 1\ -\ -\ \times$

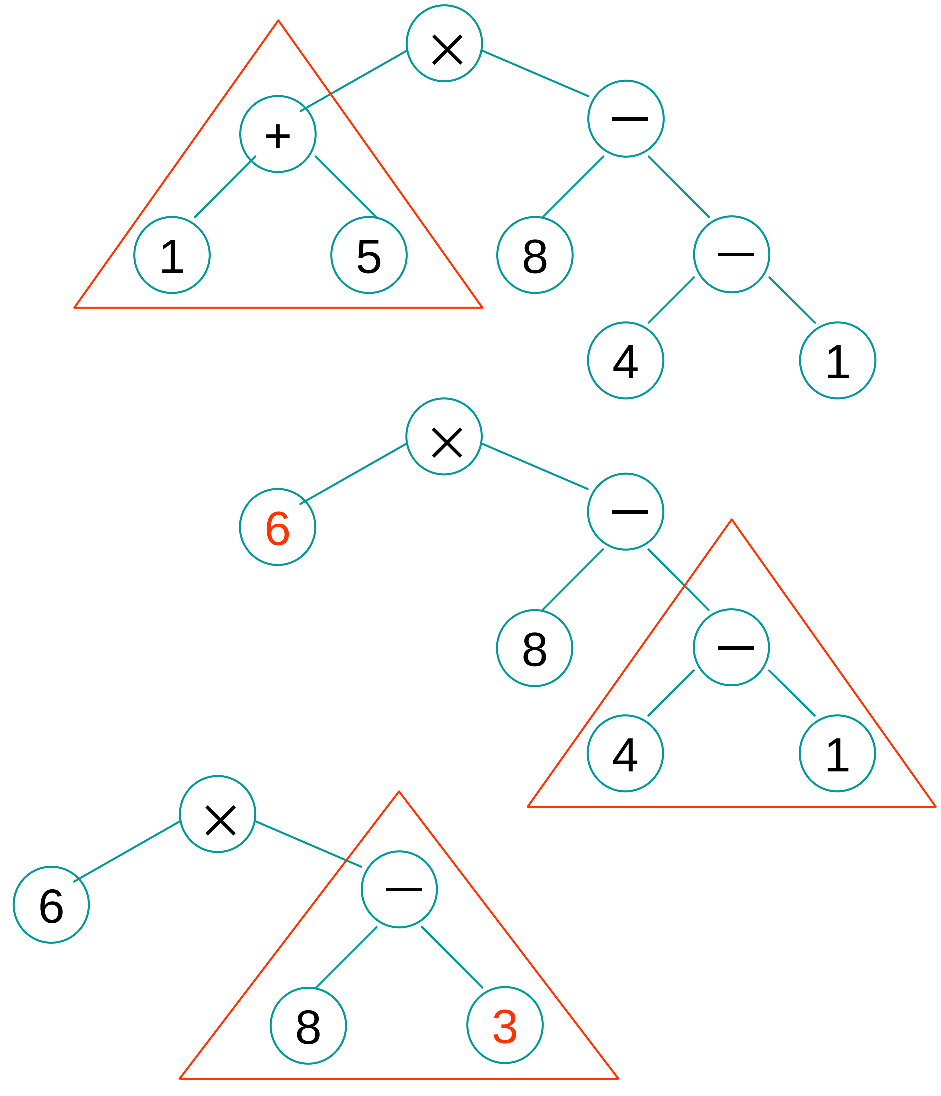
$1+5=6$

$6\ 8\ 4\ 1\ -\ -\ \times$

$4-1=3$

$6\ 8\ 3\ -\ \times$

$8-3=5$

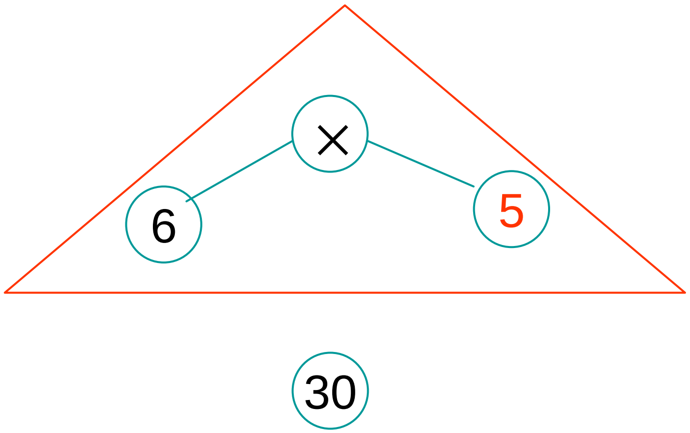


Evaluate **RPN** expression [2]

$6\ 5\ \times$

$6 \times 5 = 30$

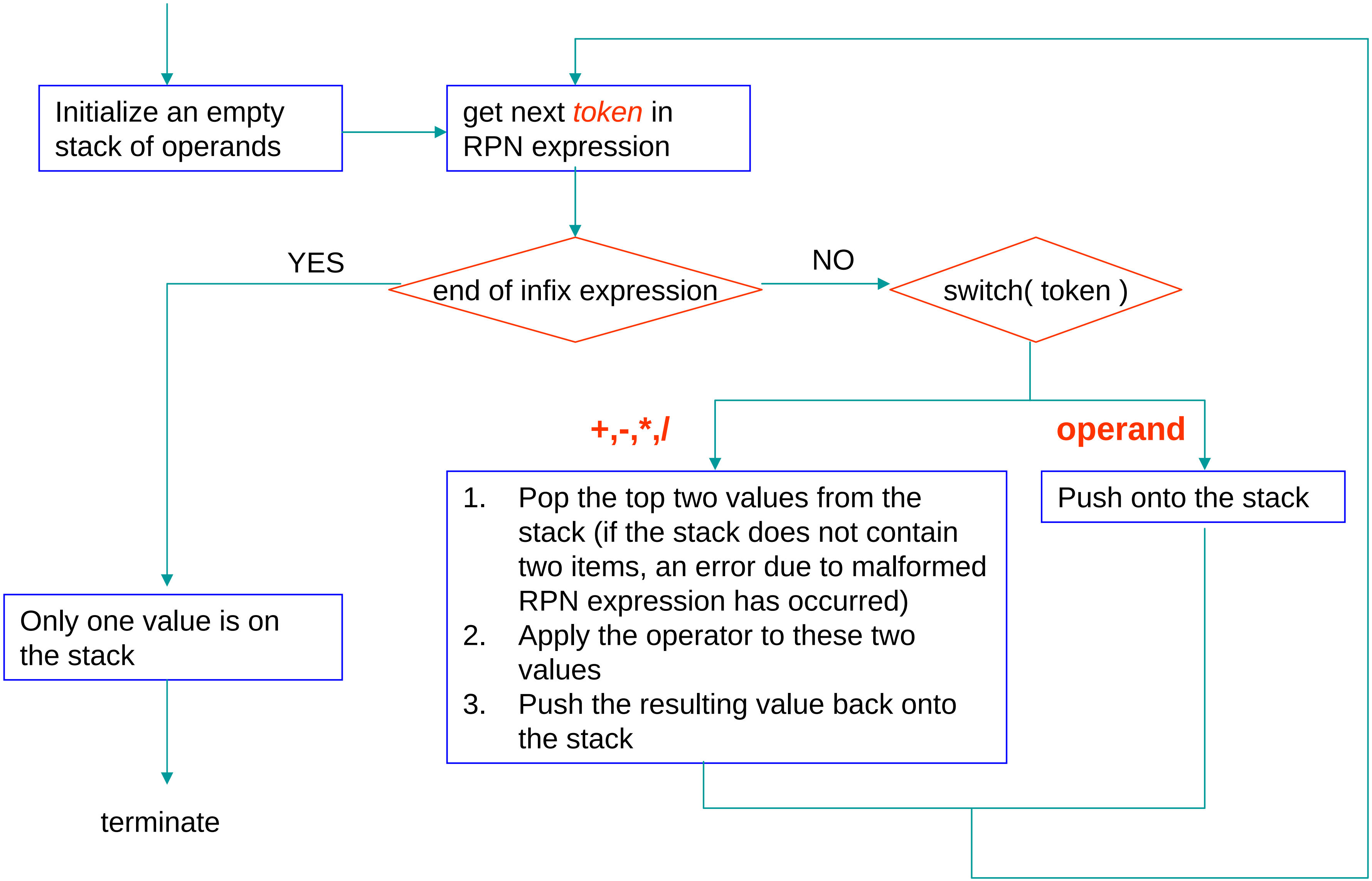
30



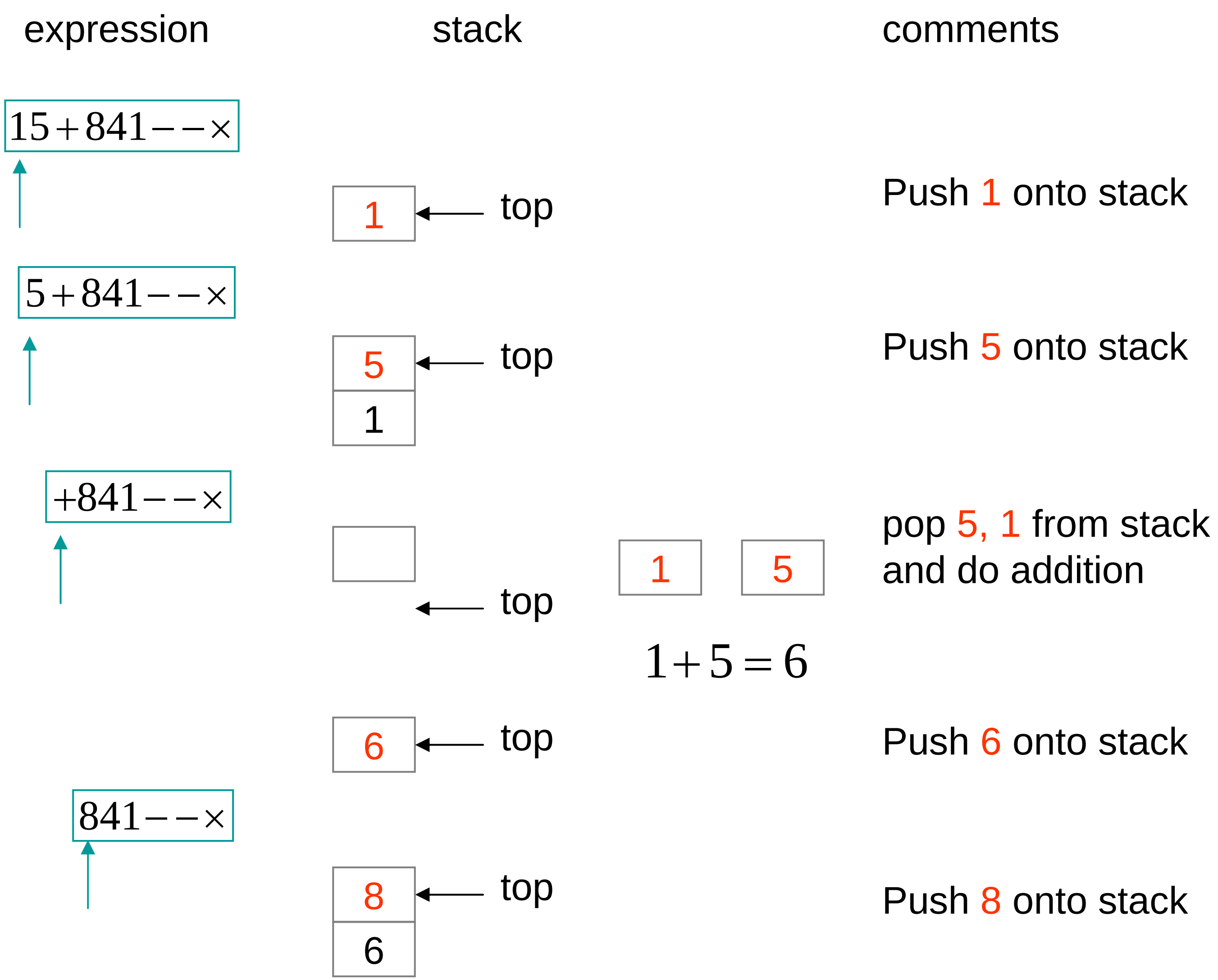
- Scanned from left to right until an operator is found, then the last two operands must be retrieved and combined.

- Order of operands satisfy Last-in, First-out, so we can use stack to store operands and then evaluate *RPN* expression

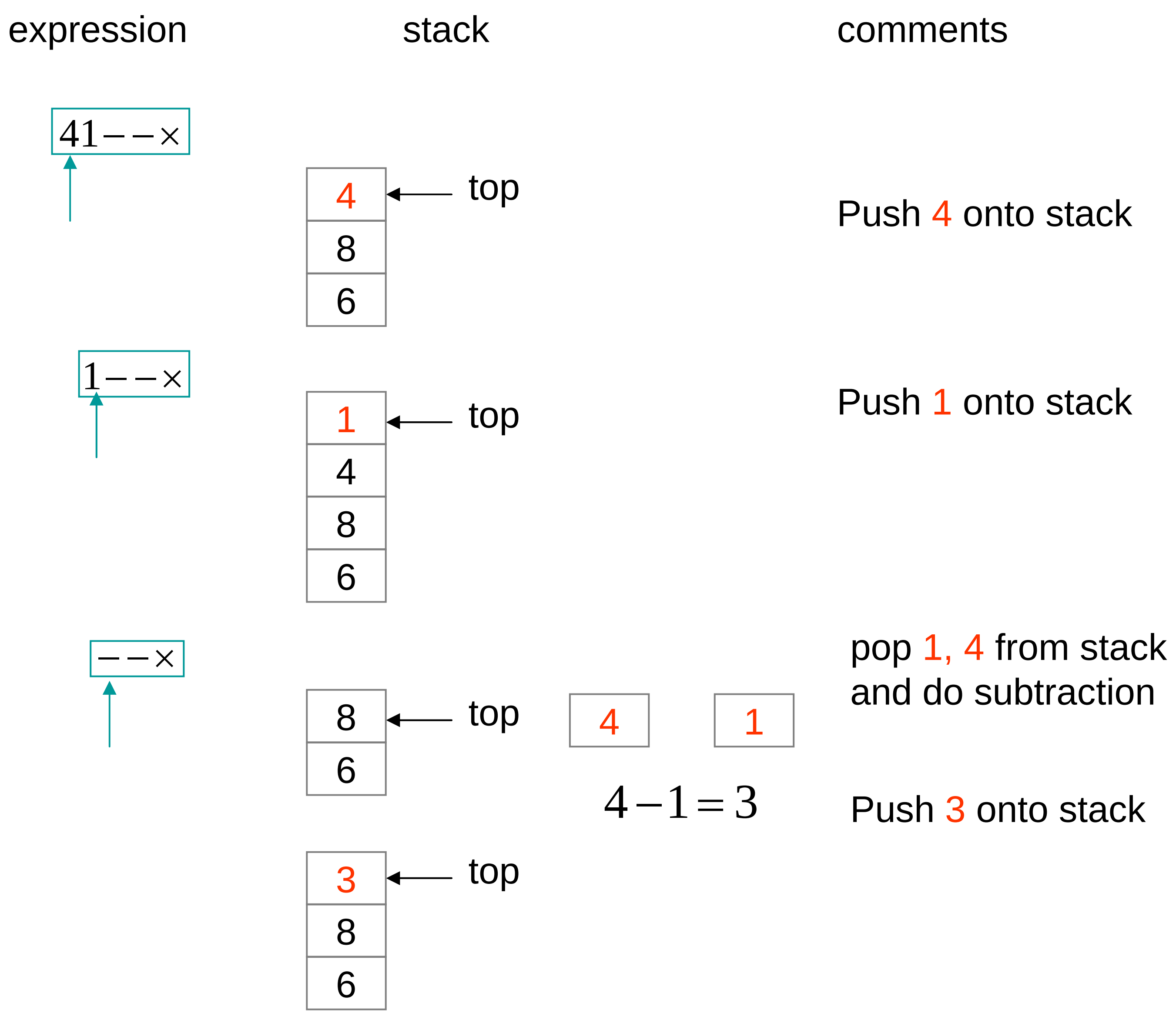
Evaluate RPN expression: flow chart [3]



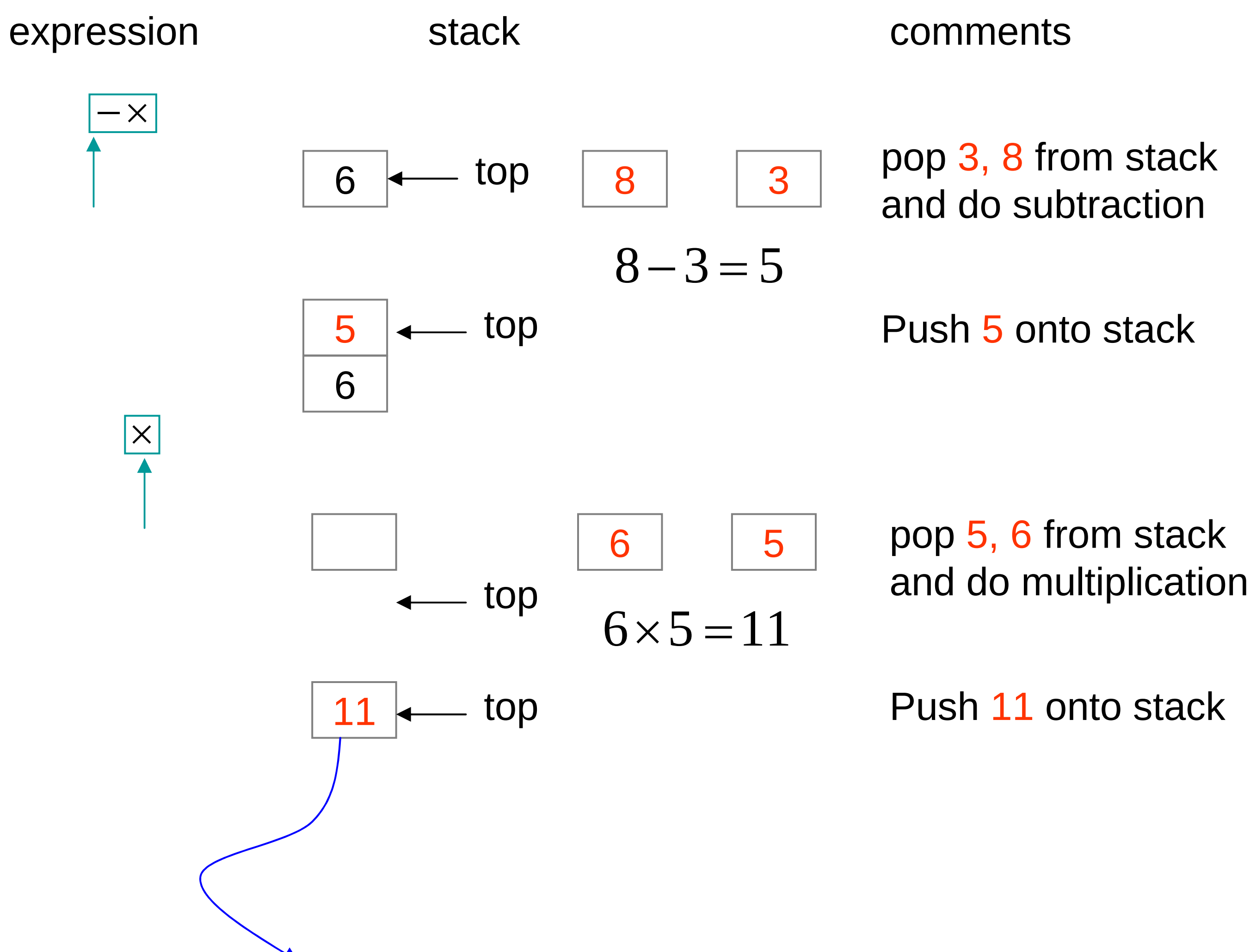
Evaluate **RPN** expression [4]



Evaluate **RPN** expression [5]



Evaluate **RPN** expression [5]



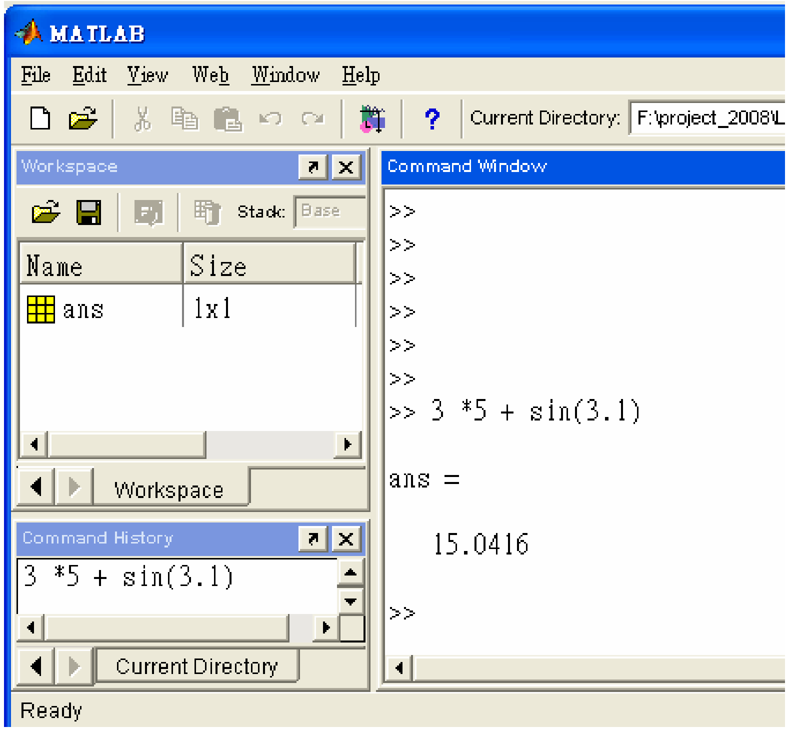
Only one value on the stack, this
value is final result

Exercise

- Implement flow chart of evaluating RPN expression, where RPN expression comes from function *RPN* we have discussed.
  You can focus on binary operator first.

- Can you extend to unary operator and general function?

- Think about How does MATLAB do when you type an expression. Can you write a MATLAB?



- survey
  -stack-oriented programming language

  - RPN calculator