Programación II

Tema 2. Pilas (parte 1)

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

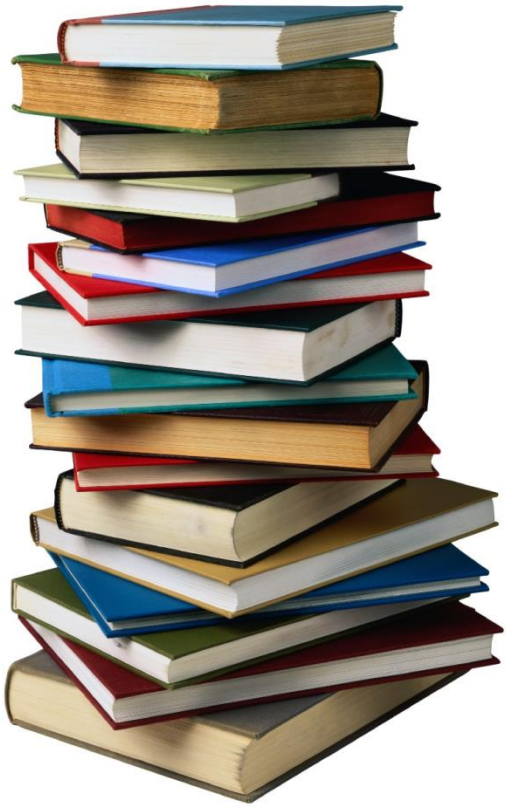
- El TAD Pila
- Estructura de datos y primitivas de Pila
- Implementación en C de Pila
  - Implementación con top de tipo entero
- Ejemplos de aplicación de Pila
  - Balanceo de paréntesis
  - Evaluación de expresiones posfijo
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

Contenidos

- **El TAD Pila**
- Estructura de datos y primitivas de Pila
- Implementación en C de Pila
  - Implementación con top de tipo entero
- Ejemplos de aplicación de Pila
  - Balanceo de paréntesis
  - Evaluación de expresiones posfijo
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

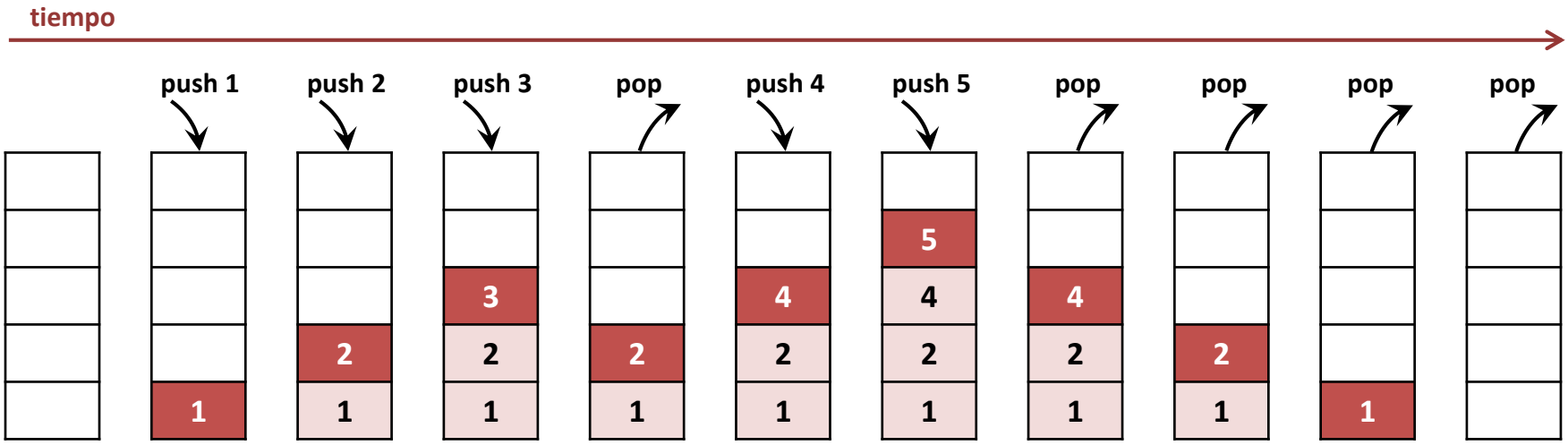
El TAD Pila

- **Pila** (***stack*** en inglés)
  - Colección de elementos LIFO - *Last In, First Out*: “el último que entra, el primero que sale”



El TAD Pila

- **Definición de Pila**
  - Contenedor de elementos que son insertados y extraídos siguiendo el principio de que el último que fue insertado será el primero en ser extraído (*LIFO – Last In, First Out*)
    - Los elementos se insertan de uno en uno: **push** (*apilar*)
    - Los elementos se extraen de uno en uno: **pop** (*desapilar*)
    - El último elemento insertado (que será el primero en ser extraído) es el único que se puede "observar" de la pila: **top** (*tope*, *cima*)



El TAD Pila

- **Funciones incluidas en la interfaz (primitivas)**

`Stack stack_init()`: crea, inicializa y devuelve una pila

`void stack_free(Stack s)`: libera (la memoria ocupada por) la pila

`Boolean stack_isEmpty(Stack s)`: devuelve *true* si la pila está vacía y *false* si no

`Boolean stack_isFull(Stack s)`: devuelve *true* si la pila está llena y *false* si no

`Status stack_push(Stack s, Element e)`: inserta un dato en una pila

`Element stack_pop(Stack s)`: extrae el dato que ocupa el top de la pila

`Element stack_top(Stack s)`: devuelve el dato que ocupa el top de la pila sin extraerlo de ella

El TAD Pila

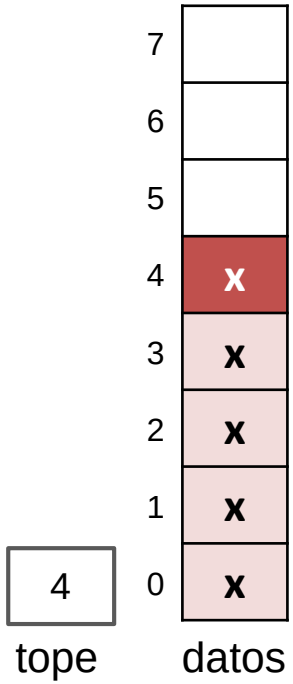
- **Aplicaciones reales de las pilas**
  - En general, todas aquellas aplicaciones que conlleven:
    - **estrategias "vuelta atrás"** (*back tracking*): la acción deshacer/*undo*
    - **algoritmos recursivos**
  - Ejemplos:
    - **Editores de texto**: pila con los últimos cambios realizados sobre un documento
    - **Navegadores web**: pila de direcciones con los sitios web más recientemente visitados
    - **Pila de programa**: zona de memoria de un programa donde se guardan temporalmente los argumentos de entrada de funciones
    - **Comprobación del balanceo de ()**, **{}**, **[]** en compiladores
    - ***Parsing* de código XML/HTML**, comprobando la existencia de etiquetas de comienzo <tag> y finalización </tag> de elementos en un documento XML
    - **Calculadoras con notación polaca inversa** (posfijo): se convierten expresiones "infijo" a "posfijo" soportando complejidad superior a la de calculadoras algebraicas (p.e., realizan cálculos parciales sin tener que pulsar "=")

Contenidos

- El TAD Pila
- **Estructura de datos y primitivas de Pila**
- Implementación en C de Pila
  - Implementación con top de tipo entero
- Ejemplos de aplicación de Pila
  - Balanceo de paréntesis
  - Evaluación de expresiones posfijo
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

Estructura de datos (una posible)

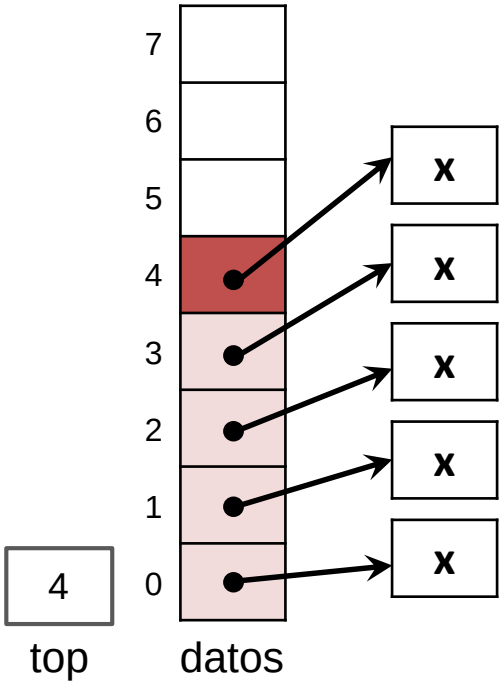
- **La pila estará formada por:**
  - **datos:** conjunto de elementos, en general del mismo tipo, almacenados de forma secuencial y accesibles desde un único punto: el *tope*
  - **tope**: indicador de la posición del último elemento insertado; da lugar a una ordenación LIFO (*last in, first out*)



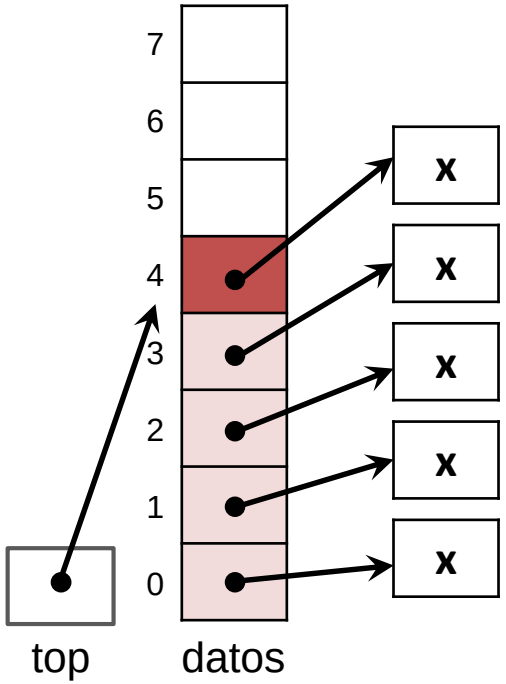
(en este dibujo asumimos que la pila tiene tamaño máximo de 8, pero no tiene por qué ser siempre ese valor)

Estructura de datos y primitivas de Pila

- **EdD en C**
  - **datos**: en este tema será un array de punteros: `Element * datos[];`
  - **top**: en este tema se declarará de 2 maneras (versiones) distintas
    - V1: Como un entero: `int top;`
    - V2: Como un puntero a un elemento del array: `Element **top;`



Versión 1



Versión 2 (en anexo)

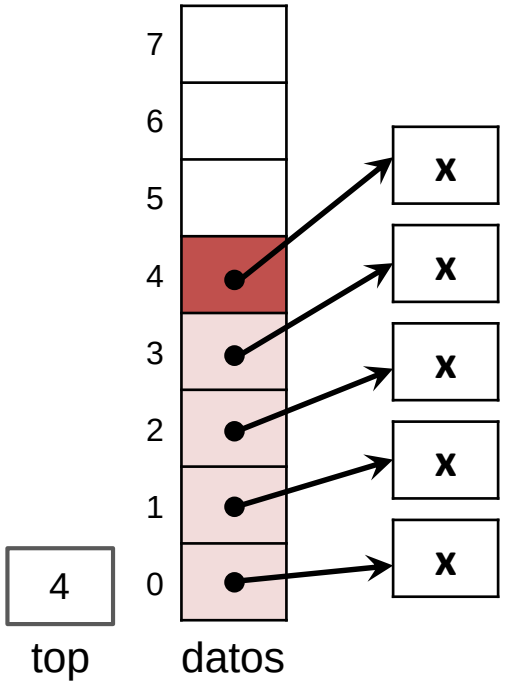
Contenidos

- El TAD Pila
- Estructura de datos y primitivas de Pila
- **Implementación en C de Pila**
  - **Implementación con top de tipo entero**
- Ejemplos de aplicación de Pila
  - Balanceo de paréntesis
  - Evaluación de expresiones posfijo
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element
  - **EdD de Pila mediante un array**

```
// En stack_h
typedef struct _Stack Stack;

// En stack_c
#define STACK_MAX 8
struct _Stack {
   Element * datos[STACK_MAX];
   int     top;
};
```



Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element que, entre otras, incluye en su interfaz las funciones *free* y *copy*:

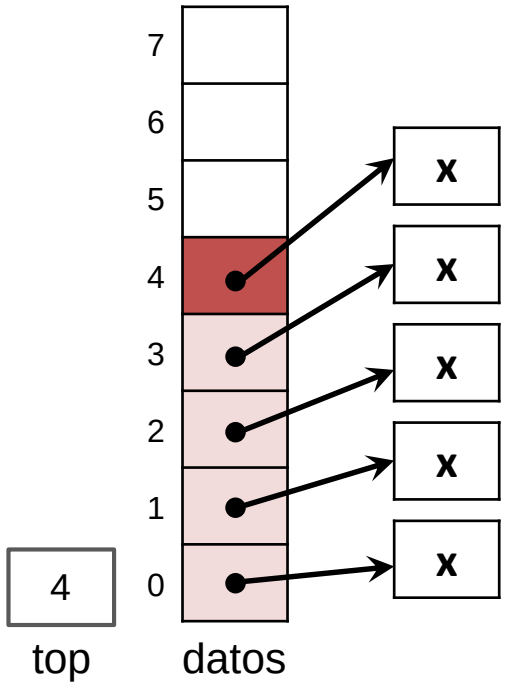
```
void element_free(Element *pe);
Element *element_copy(const Element *pe);
```

  - **Interfaz (prototipos en stack.h)**

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);
Element *stack_pop(Stack *ps);
Element *stack_top(Stack *ps);
```

  - **Estructura de datos (en stack.c)**

```
struct _Stack {
   Element * datos[STACK_MAX];
   int       top;
};
```



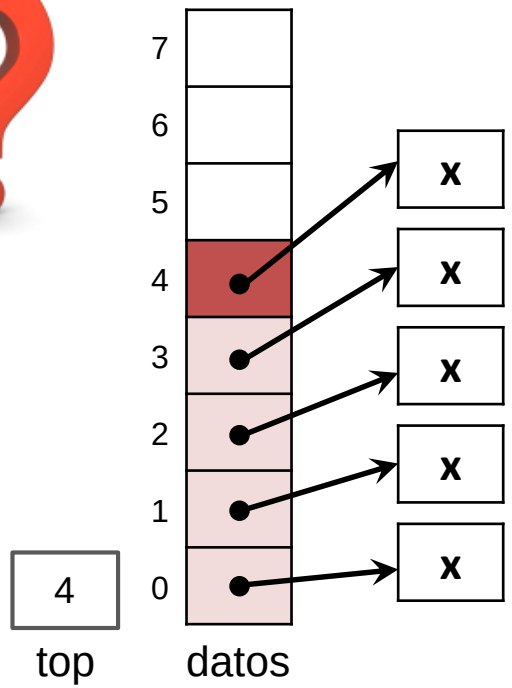
Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element que, entre otras, incluye en su interfaz las funciones *free* y *copy*:

```
void element_free(Element *pe);
Element *element_copy(const Element *pe);
```

  - **Interfaz (prototipos en stack.h)**

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);
Element *stack_pop(Stack *ps);
Element *stack_top(Stack *ps);
```



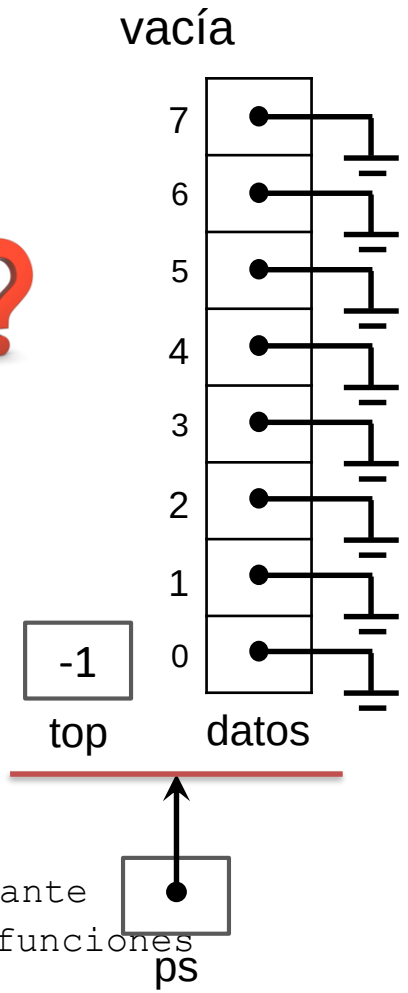
  - **Estructura de datos (en stack.c)**

```
struct _Stack {
   Element * datos[STACK_MAX];
   int       top;
};
```

Implementación en C de Pila

- **Implementación con top de tipo entero**

```
Stack *stack_init() {
   Stack *ps = NULL;
   int i;

   ps = (Stack *) malloc(sizeof(Stack));
   if (ps == NULL) {
      return NULL;
   }

   for (i=0; i<STACK_MAX; i++)
      ps->datos[i] = NULL;

   ps->top = -1;    // (1)
   return ps;
}
```



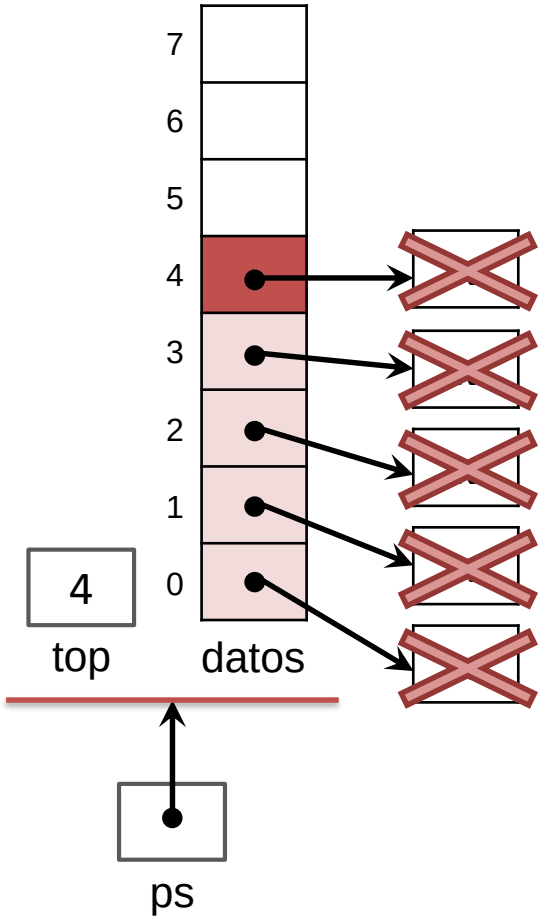
(1) Hay otras posibilidades (e.g., ps->top = 0). Lo importante es ser coherente con la implementación del resto de las funciones

Implementación en C de Pila

- **Implementación con top de tipo entero**

Existe: `void element_free (Element *pe);`

```
void stack_free (Stack *ps) {
   int i;

   if (ps != NULL){
      for (i=0; i<=ps->top; i++) {
         element_free(ps->datos[i]);
         ps->datos[i] = NULL;
      }


   }
}
```

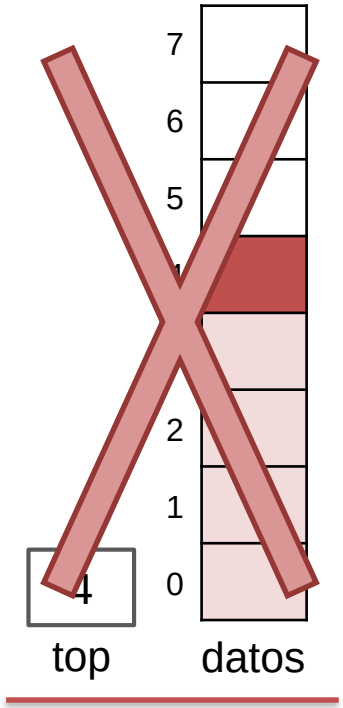


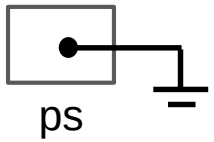
Implementación en C de Pila

- **Implementación con top de tipo entero**

Existe: `void element_free (Element *pe);`

```
void stack_free (Stack *ps) {
   int i;

   if (ps != NULL){
      for (i=0; i<=ps->top; i++) {
         element_free(ps->datos[i]);
         ps->datos[i] = NULL;

      free(ps);
   // ps = NULL; se hace fuera,
   // tras llamar a pila_liberar
   }
}
```





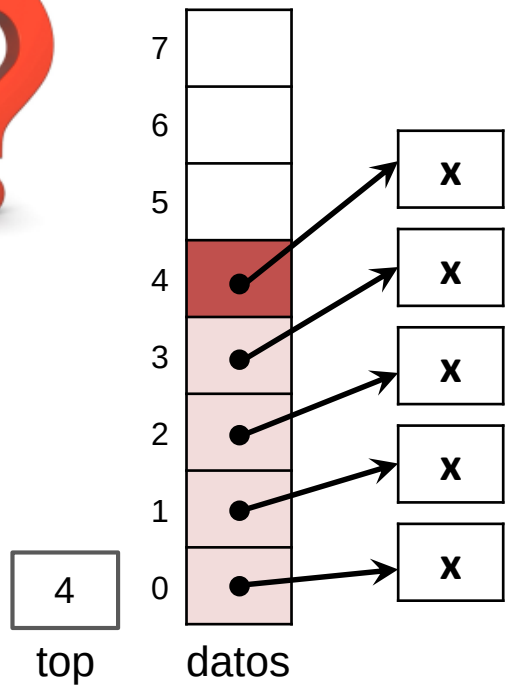
Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element que, entre otras, tiene asociadas las primitivas *liberar* y *copiar*:

```
void elemento_free(Element *pe);
Element *element_copy(const Element *pe);
```

  - **Interfaz (prototipos en stack_h)**

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);
Element *stack_pop(Stack *ps);
Element *stack_top(Stack *ps);
```



  - **Estructura de datos (en pila.c)**

```
struct _Stack {
   Element  *datos[PILA_MAX];
   int      top;
};
```

Implementación en C de Pila

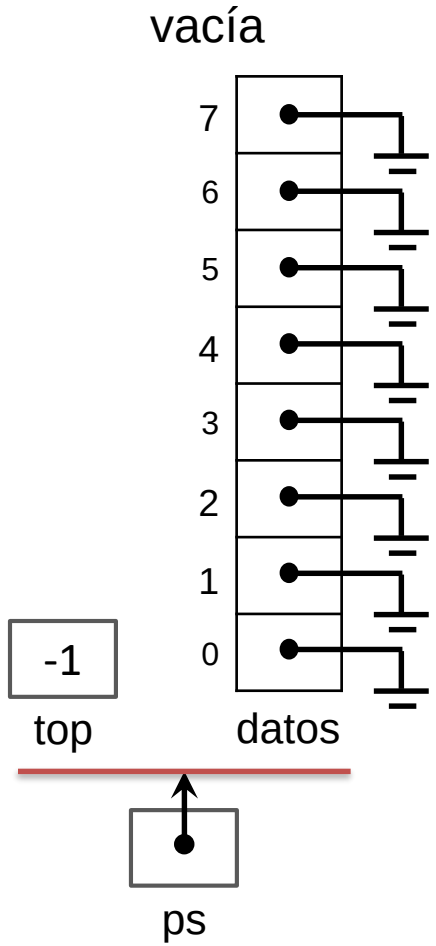
- **Implementación con top de tipo entero**

```
Boolean stack_isEmpty(const Stack *ps) {
   if (ps == NULL)
      return TRUE;

   if (ps->top == -1)
      return TRUE;

   return FALSE;
}
```

Nota:

Llamaremos a stack_isEmpty cuando queremos hacer pop, para comprobar si hay elementos en la pila. Si la pila está vacía, no se puede extraer nada.

Por eso, si el puntero ps que recibe stack_isEmpty es NULL, la función devuelve TRUE indicando que la pila "está" vacía, con el fin de que luego no se haga pop.



Implementación en C de Pila

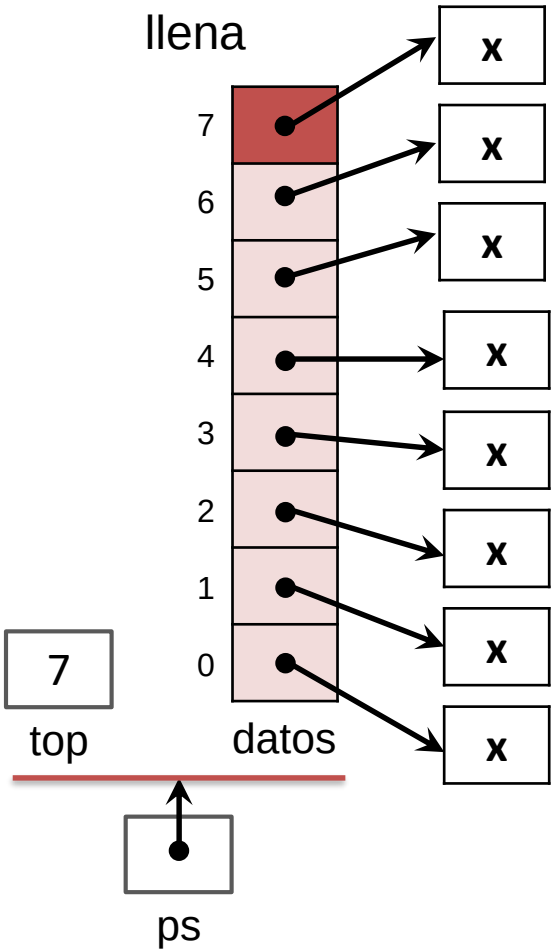
- **Implementación con top de tipo entero**

```
Boolean stack_isFull(const Stack *ps) {
   if (ps == NULL)
      return TRUE;

   if (ps->top == STACK_MAX-1) {
      return TRUE;
   }
   return FALSE;
}
```

Nota:

Llamaremos a stack_isEmpty cuando queramos hacer push, para comprobar si la pila tiene o no capacidad. Si la pila está llena, no se puede hacer el push.

Por eso, si el ps que recibe stack_isEmpty es NULL, esta función devuelve TRUE indicando que la pila "está" llena, con el fin de que luego no se haga push.



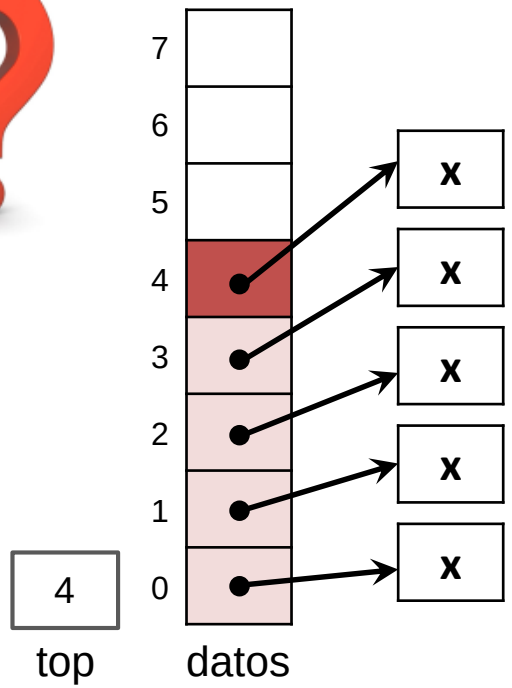
Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element que, entre otras, incluye en su interfaz las funciones *free* y *copy*:

```
void element_free(Element *pe);
Element *element_free(const Element *pe);
```

  - **Interfaz (prototipos en stack_h)**

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);
Element *stack_pop(Stack *ps);
Element *stack_top(Stack *ps);
```



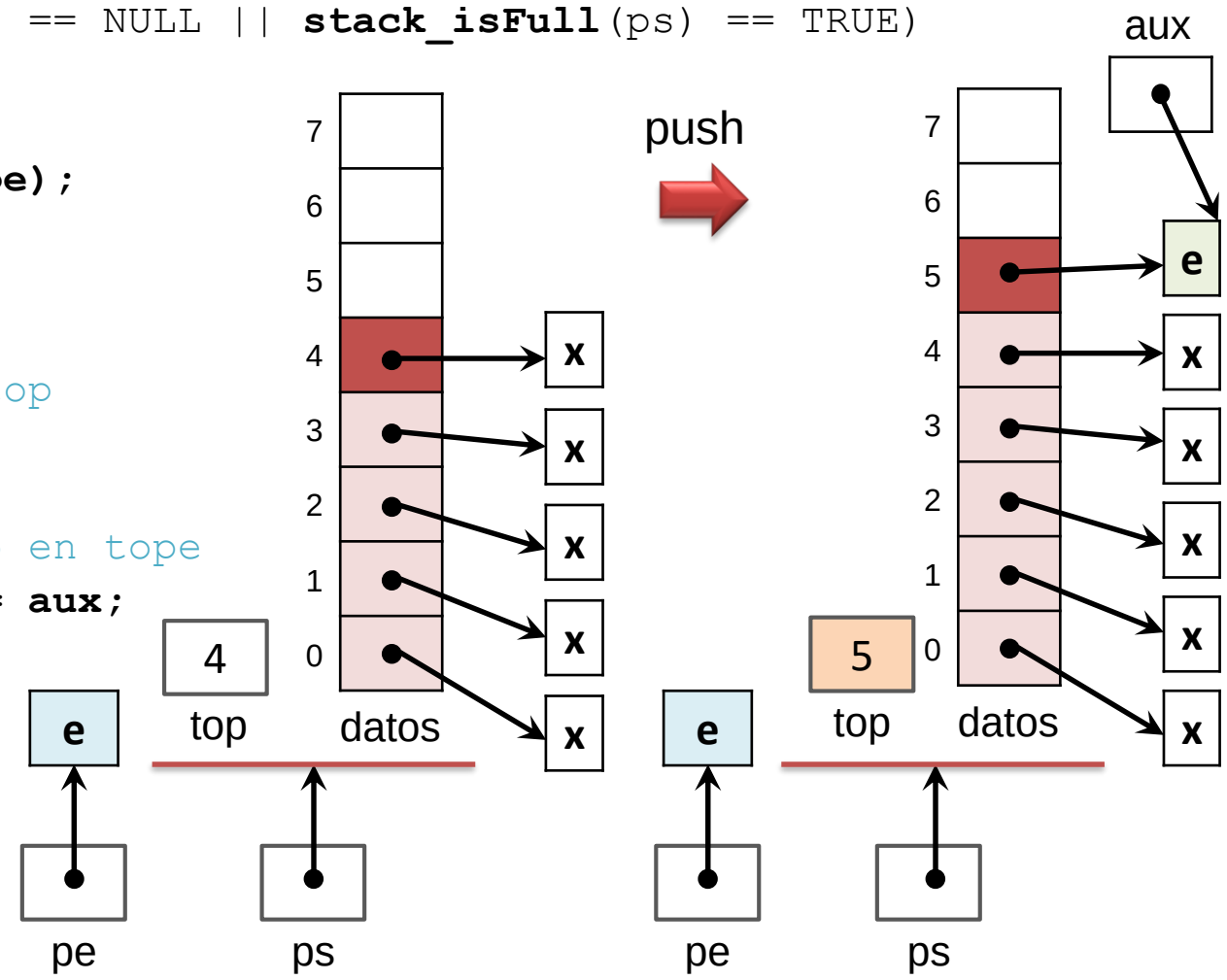
  - **Estructura de datos (en stack_c)**

```
struct _Stack {
   Element  *datos[STACK_MAX];
   int      top;
};
```

Implementación en C de Pila

- **Implementación con top de tipo entero**

```
status stack_push(Stack *ps, const Element *pe) {
   Element *aux = NULL;

   if (ps == NULL || pe == NULL || stack_isFull(ps) == TRUE)
      return ERR;

   aux = element_copy(pe);
   if (aux == NULL)
      return ERR;

   // Actualizamos el top
   ps->top++;

   // Guardamos el dato en tope
   ps->datos[ps->top] = aux;

   return OK;
}
```



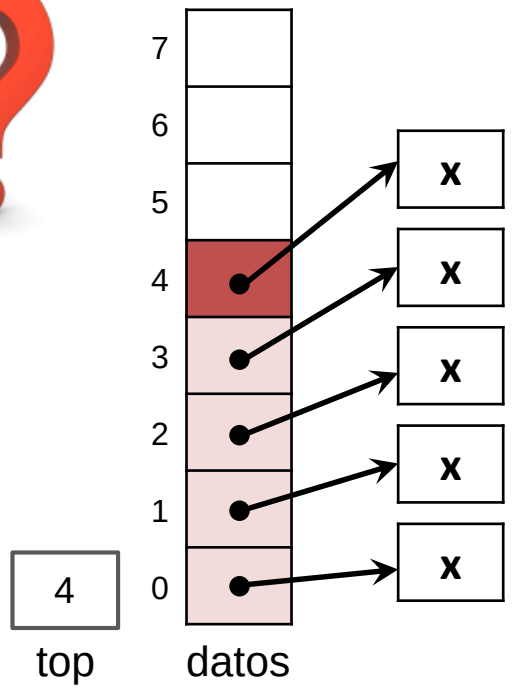
Implementación en C de Pila

- **Implementación con top de tipo entero**
  - Asumimos la existencia del TAD Element que, entre otras, incluye en su interfaz las funciones *free* y *copy*:

```
void element_free(Element *pe);
Element *element_copy(const Element *pe);
```

  - **Interfaz (prototipos en stack_h)**

```
Stack *stack_init();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Boolean stack_isFull(const Stack *ps);
Status stack_push(Stack *ps, const Element *pe);
Elemento *stack_pop(Stack *ps);
Element stack_top(Stack *ps);
```



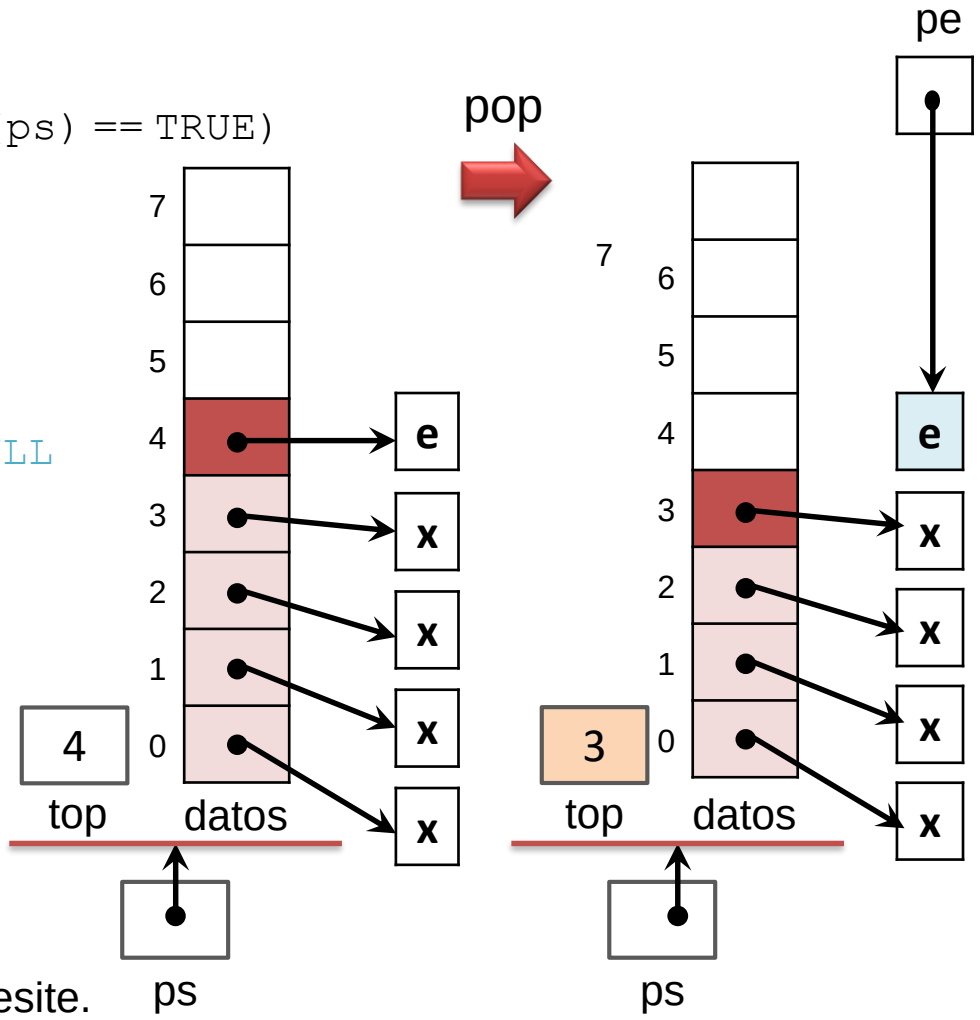
  - **Estructura de datos (en stack_c)**

```
struct _Stack {
   Element  *datos[STACK_MAX];
   int      top;
};
```

Implementación en C de Pila

- **Implementación con top de tipo entero**

```
Element *stack_pop(Stack *ps){
    Element *pe = NULL;

    if (ps == NULL || stack_isEmpty(ps) == TRUE)
        return NULL;

    // Recuperamos el dato del top
    pe = ps->datos[ps->top];

    // Asignamos el Elemento* a NULL
    ps->datos [ps->top] = NULL;

    // Actualizamos el top
    ps->top--;

    return pe;
}
```



Nota: el elemento cuyo puntero se devuelve ha de liberarse fuera, en algún momento después de hacer pop, cuando ya no se necesite.

Implementación en C de Pila (usando Element)

Interfaz del TAD Element (element.h)

```
Element * element_init (void *);
void  element_free (Element *);

Element * element_copy (Element *);

Status element_set (Element *, void *);
void * element_get (Element *ele);
```

```
#include “stack_h”

void clean_up (Element *ele, Stack *s, Status ret) {
   element_free (ele);
   stack_free (s);
   exit (ret);
}
void main() {
   Stack *s = NULL;
   Element *ele = NULL;
   int i = 3;
   Status ret;

   // crea un elemento de pila
   ele = element_init (&i);
   if (!ele) return ERROR;

    // inicializa la pila
    s = stack_init ();
    if (!s) clean_up (ele, s, ERROR);

    // inserta el elemento en la pila (hace copia de ele)
    ret  = stack_push (s, ele);
    if (ret == ERROR) clean_up (ele, s, ERROR);

    //libera el elemento (push insertó una copia del mismo)
    element_free (ele);

    // extrae un elemento de la pila
    ele = stack_pop (s);
    fprintf (stdout, “ %d ” , *(int *)element_get (ele));

    // libera los recursos
    clean_up (ele, s, OK);
}
```

Contenidos

- El TAD Pila
- Estructura de datos y primitivas de Pila
- Implementación en C de Pila
  - Implementación con top de tipo entero
- **Ejemplos de aplicación de Pila**
  - **Balanceo de paréntesis**
  - Evaluación de expresiones posfijo
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

Balanceo de paréntesis

- Determinar si una expresión (p.e., una operación aritmética) contiene el mismo número de paréntesis de apertura que de cierre

  `( 3 * 4 / ( 6 - 1 ) )` → CORRECTA

  `2 + ( 5 / ( 4 + 7 )` → INCORRECTA

  `6 * 4 + 9 )` → INCORRECTA

Balanceo de paréntesis (Idea y algoritmo)

EJEMPLOS

```
Input :   (A+B)+C;
Output : TRUE

Input :   (A+B))+C(;
Output : FALSE

Input :   A)+B+C;
Output: FALSE
```

- Ejemplo 1: `( A * B / ( C - D ) ) ;`

Símbolo;	Pila de paréntesis
(	(
A	(
*	(
B	(
/	(
(	((
C	((
-	((
D	((
)	(
)	
;	

ALGORITMO

S 1: Inicializar pila auxiliar

S 2: Recorrer la expresión de izquierda a derecha:

S 2.1: si el símbolo leído es un paréntesis de apertura, insertarlo en la pila.

S 2.2: si el símbolo leído es un paréntesis de cierre, extraer un elemento de la pila; si la pila estaba vacía (no se pudo extraer), la expresión **no** está balanceada.

S 3: Si después de recorrer la expresión la pila **no** queda vacía, la expresión **no** está balanceada.

S 4: Liberar la pila auxiliar.

Balanceo de paréntesis (Pseudocódigo sin CdE)

Sin control de errores

```
Boolean balancedExpression (String str)
   ret = TRUE
  i = 0
  s = stack_init ()

  // recorremos la expresión
  while  str[i] ≠  EoS :
        if str[i] == ‘(‘ :
                stack_push (s, str[i])

        else  if str[i] == ')' :
                stack_pop (s)

        i = i + 1

  if stack_isEmpty (s) == FALSE :
        ret = FALSE

  stack_free (s)
  return ret
```

- Ejemplo 1: `( 3 * 4 / ( 6 - 1 ) ) ;`

Símbolo	Pila de paréntesis
(	(
3	(
*	(
4	(
/	(
(	((
6	((
-	((
1	((
)	(
)	

¿Control de errores?

Balanceo de paréntesis (Pseudocódigo sin CdE)


```
Boolean balancedExpression (String str)
  ret = TRUE
  i = 0
  s = stack_init ()  (1)

  while  str[i] ≠  EoS :
        if str[i] == '(' :
              stack_push (s, str[i])  (2)
        else  if str[i] == ')' :
              stack_pop (s)  (A)
        i = i + 1

  if stack_isEmpty (s) == FALSE :  (B)
        ret = FALSE

  stack_free (s)
  return ret
```

- Ejemplo 1: `( 3 * 4 / ( 6 - 1 ) ) ;`

Símbolo	Pila de paréntesis
(	(
3	(
*	(
4	(
/	(
(	((
6	((
-	((
1	((
)	(
)	

Posibles fuentes de error (funciones que pueden devolver ERROR, *overflow*): 1 2

Errores de balanceo se detectan en: A B

- Ejemplo 2: `2 + ( 5 / ( 4 + 7 )`
- Ejemplo 3: `6 * 4 + 9 )`

Balanceo de paréntesis (Pseudocódigo con CdE v1)


```
Boolean balancedExpression (String str)
  st = OK
  i = 0

  if (s = stack_init ()) == NULL :  1
      return FALSE

  while  str[i] ≠ EoS:
        if str[i] == '(' :
            if stack_push (s, str[i]) == ERROR :  2
                stack_free (s)
                return FALSE
        else  if str[i] == ')' :
             if stack_pop (s) == ERROR :  A
                stack_free (s)
                return FALSE
        i = i + 1
  if stack_isEmpty (s) == FALSE :  B
        st = ERROR
  stack_free (s)
  if st == ERROR : return FALSE
  return TRUE
```

V1: si hay error, libera pila y devuelfe FALSE

(A * B / (C - D)) ;

Símbolo	Pila de paréntesis
(	(
A	(
*	(
B	(
/	(
(	((
C	((
-	((
D	((
)	(
)	
;	

¿Puedes mejorarlo para distinguir en el retorno las fuentes de error?

Balanceo de paréntesis (Pseudocódigo con CdE v2)


```
Boolean balancedExpression (String str)
  st = OK
  i = 0

  if (s = stack_init ()) == NULL :  1
      return FALSE

  while  str[i] ≠ EoS AND st == OK : //si ERROR, sale
        if str[i] == '(' :
            st = stack_push (s, str[i])  2

        else  if str[i] == ')' :
                if stack_pop (s) == ERROR :  A
                    st = ERROR
        i = i + 1

  if st == OK AND  stack_isEmpty (s) == FALSE :  B
        st = ERROR

  stack_free (s)

  if st == ERROR : return FALSE
  return TRUE
```

(A * B / (C - D)) ;

Símbolo	Pila de paréntesis
(	(
A	(
*	(
B	(
/	(
(	((
C	((
-	((
D	((
)	(
)	
;	

V2:
Si hay error, st=ERROR
Se libera y se retorna abajo del todo

¿Puedes mejorarlo para distinguir en el retorno del algoritmo las fuentes de error?

Balanceo de símbolos

- Determinar si una expresión (p.e. una operación aritmética) contiene el mismo número de paréntesis de apertura que de cierre, **distinguiendo '()', '{}', '[]'**

  `{ 3 * 4 / ( 6 - 1 ) }` → CORRECTA

  `2 + ( 5 / [ 4 + 7 ) )` → INCORRECTA

  `6 * 4 + 9 }` → INCORRECTA

Balanceo de símbolos


```
Boolean balancedExpression (String str)
  st = OK
  i = 0

  if (s = stack_init ()) == NULL :
      return FALSE

  while  str[i] ≠ EoS AND st == OK : //si ERROR, sale
        if str[i] == '(' :
            st = stack_push (s, str[i])

        else  if str[i] == ')' :
                if stack_pop (s) == ERROR :
                    st = ERROR
        i = i + 1

  if st == OK AND stack_isEmpty (s) == FALSE :
        st = ERROR

  stack_free (s)

  if st == ERROR : return FALSE
  return TRUE
```

¿Qué hay que cambiar para comprobar distintos símbolos?

Balanceo de símbolos

ALGORITMO

S 1: Inicializar una pila auxiliar.
S 2: Recorrer la expresión de izquierda a derecha :
S 2.1: si el símbolo leído es de apertura, insertarlo en la pila.
S 2.2: si el símbolo leído es de cierre y es pareja con el símbolo del tope de la pila, extraerlo de la pila; si la pila estaba vacía (no se pudo mirar símbolo) o los símbolos no eran pareja, la expresión **no** está balanceada
S 3: Si después de recorrer la expresión la pila **no** queda vacía, la expresión **no** está balanceada.
S 4: Liberar la pila auxiliar.

- Ejemplo : `[ ( A * B ) + C * D ] ;`

Símbolo	Pila de paréntesis
[	[
(	[(
A	[(
*	[(
B	[(
)	[
+	[
C	[
*	[
D	[
]	
;	

Balanceo de símbolos


```
Boolean balancedSymbol (String str)
        i = 0
        st = OK

        if (s = stack_init ()) == ERROR :  1
             return FALSE

        while str[i] ≠  EoS AND st == OK :
             if isOpenSymbol (str[i]) == TRUE :
                 st = stack_push (s, str[i])  2

             else  if isCloseSymbol (str[i])  == TRUE :
                 if arePaired (str[i], stack_top (s)) == TRUE :
                          stack_pop (s)                          A
                 else :
                          st = ERROR  B

             i = i + 1

        if st == OK AND stack_isEmpty (s) == FALSE :  C
             st = ERROR

        stack_free (s)
        if st == ERROR : return FALSE
        return TRUE
```

Notas:

- stack_top no devuelve algo válido si la pila está vacía
- arePaired devuelve FALSE si alguno de los elementos no son válidos.

Contenidos

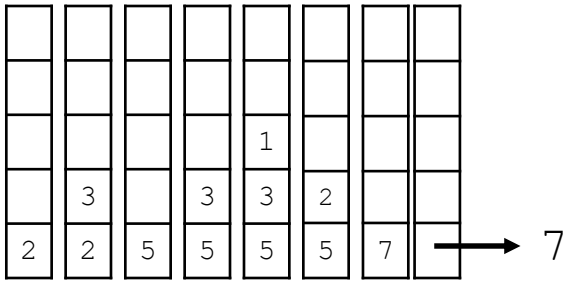
- El TAD Pila
- Estructura de datos y primitivas de Pila
- Implementación en C de Pila
  - Implementación con top de tipo entero
- **Ejemplos de aplicación de Pila**
  - Balanceo de paréntesis
  - **Evaluación de expresiones posfijo**
  - Conversión entre notaciones infijo, posfijo y prefijo
- Anexos

Evaluación de expresiones posfijo

- **Notación infijo**
  - Los operadores se indican entre operandos: **`A+B`**
  - La habitual
- **Notación posfijo (o sufijo)**
  - Los operadores se indican después de los operandos: **`AB+`**
  - La más "práctica" para un ordenador (algoritmo de evaluación más rápido)
- **Notación prefijo**
  - Los operadores se indican antes de los operandos: **`+AB`**
- Ejemplo:
  - En infijo: `2 - 3 * (4 + 1)`
  - En posfijo: `2 3 4 1 + * -`
  - En prefijo: `- 2 * 3 + 4 1`

Evaluación de expresiones posfijo

- Ejemplo: `2 3 + 3 1 - + ;`



- Ejemplo: `1 2 * 4 + 3 4 + * ;`

ALGORITMO

E 1: Inicializar una pila auxiliar.

E 2: Recorrer la expresión de izquierda a derecha :

E 2.1: si el símbolo leído es un operando, insertarlo en la pila.

E 3.2: si el símbolo leído es un operador, extraer 2 elementos de la pila (primero el operando2 y después el operando1), evaluar la expresión formada por los operandos extraídos y el operador (operando1 operador operando2) e insertar el resultado en la pila.

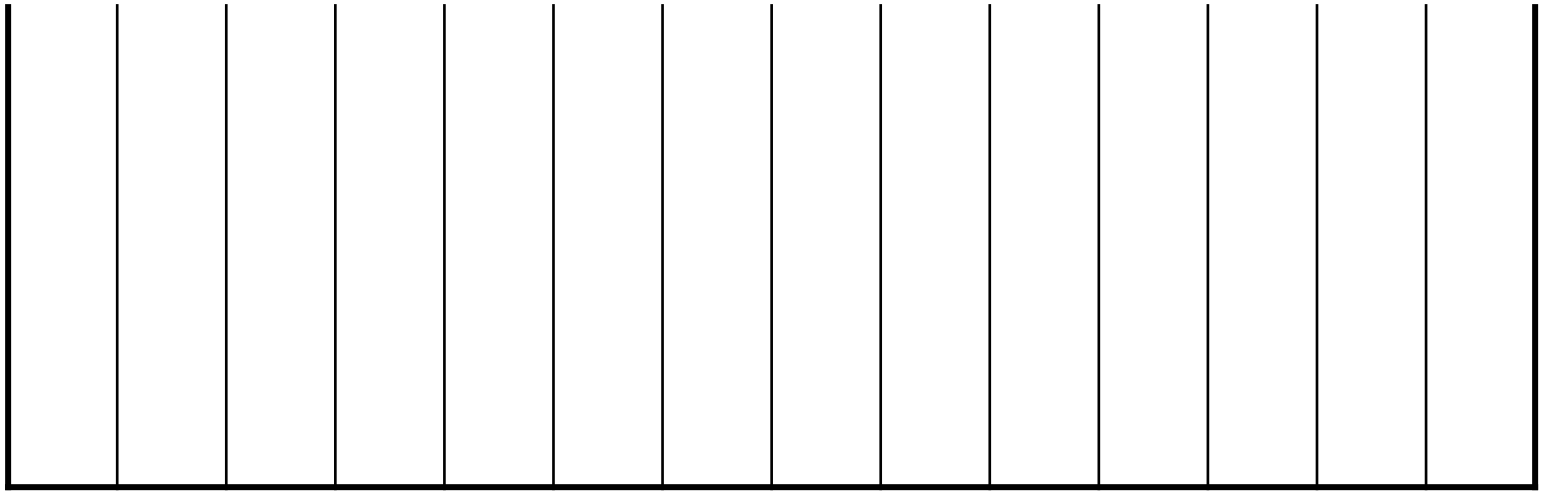
E 3: Extraer el resultado de la expresion de la pila. Si tras extraer la pila no queda vacía, la expresión es incorrecta.

E 4: Liberar la pila auxiliar.

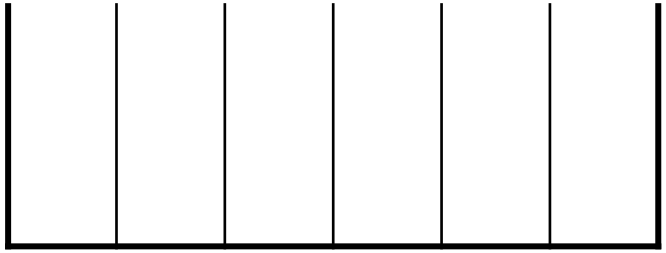
Evaluación de expresiones posfijo

- **Ejemplos**

6 2 / 3 2 * 5 1 - 3 * + - ;



A B * C - ;



1 2 * 4 2 / 3 4 + 5 * - + ; → – 31

Evaluación de expresiones posfijo

- **Pseudocódigo (sin CdE)**

```
Status evalPosfix (String S, Integer ret)
    st = OK
    p = stack_init ()
    while s[i] ≠ EoS :
         if isOperand (s[i]) :
              stack_push (p, s[i])
         else if isOperator (s[i]) :
              op2 = stack_pop (p)
              op1 = stack_pop (p)
              ret = eval (op1, s[i], op2)
              stack_push (p, ret)
         i = i + 1

    ret = stack_pop (p) //resultado almacenado en ret

    if stack_isEmpty (p) = FALSE : //si la pila no queda vacía, incorrecto
        st = ERROR

    stack_free(p)
    return st
```

¿Control de errores?

Ejemplo: `1 6 * 2 1 ;`

Ejemplo: `1 6 * 2 + / ;`

Evaluación de expresiones posfijo (con CdE)


```
Status evalPosfix (String S, Integer ret)
    st = OK

    if (p = stack_init ()) == ERROR
           return ERROR                          // ERROR 1: la pila está llena (overflow)

    while s[i] ≠ EoS AND st == OK :

           if isOperand (s[i]) :
                 st = stack_push (p, s[i]) ) // ERROR 2: la pila está llena (overflow)

           else if isOperator (s[i]) :
                 op2 = stack_pop(p)
                 op1 = stack_pop(p)
                 if op1 ≠ NULL  AND  op2 ≠ NULL :
                     ret = eval(op1, s[i], op2)
                     st = stack_push (p, ret) // ERROR 3: la pila está llena (overflow)
                 else :
                     st = ERROR  // ERROR 4: no hay suficientes operandos en la pila

           i = i + 1

    if st == OK AND (ret = stack_pop (p)) == NULL : //ERROR 5: expresión incorrecta
           st = ERROR

    if st == OK AND stack_isEmpty (p) == FALSE : //ERROR 6: pila no queda vacía, sobran operandos
           st = ERROR

    stack_free(p)
    return st
```

Ejemplo: `1 6 * 2 1 ;`

Ejemplo: `1 6 * 2 + / ;`

Contenidos

- El TAD Pila
- Estructura de datos y primitivas de Pila
- Implementación en C de Pila
  - Implementación con top de tipo entero
- **Ejemplos de aplicación de Pila**
  - Balanceo de paréntesis
  - Evaluación de expresiones posfijo
  - **Conversión entre notaciones infijo, posfijo y prefijo**
- Anexos

Conversión de expresiones: infijo-posfijo

- **Infijo → Posfijo**
- Conversión de A + B * C
  - A +(B * C)=> ~~AB+C*~~
  - A + B * C => ABC*+
- Hay que tener en cuenta:
  - Precedencia de operadores
  - Asociatividad de operaciones (a igual precedencia, de izquierda a derecha)

precedencia

\+

\-

operador	asociatividad
() [] . ->	izquierda-derecha
^	izquierda-derecha
* / %	
+ -	
== ≠	
AND OR	
=	derecha-izquierda

Conversión de expresiones: infijo-posfijo

- **Idea inicial (¿correcta o errónea?)**

`A + B * C ; => ABC*+`

Símbolo	Traducción parcial	Pila de OPERADORES
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
;	ABC*+	

- **Posible algoritmo** ¿Funciona para `A * B + C / E ;` ?

1. Si el símbolo actual it es operando: “imprimir” (añadir it a cadena de traducción parcial)
2. Si it es operador: pila_push(pila, it)
3. Al final: vaciar pila e imprimiendo elementos en cadena traducción en el orden en que se van extrayendo

Conversión de expresiones: infijo-posfijo

- **Otro ejemplo**

```
A * B + C / E ; => AB*CE/+
```

Símbolo	Traducción parcial	Pila de OPERADORES
A	A	
*	A	*
B	AB	*
+	AB	* +
C	ABC	* +
/	ABC	* + /
E	ABCE	* + /
;	ABCE/+* MAL	

- **Ajustes al algoritmo anterior para que funcione:**
  - Los operadores de mayor precedencia salen antes de pila
  - A igual precedencia, sale antes el que se leyó antes

Conversión de expresiones: infijo-posfijo

- Teniendo en cuenta precedencia de operadores

```
A * B + C / E ; => AB*CE/+
```

Símbolo	Traducción parcial	Pila de OPERADORES
A	A	
*	A	*
B	AB	*
+	AB*	+
C	AB*C	+
/	AB*C	+ /
E	AB*CE	+ /
;	AB*CE/+	

- **Idea para el algoritmo**

1. Si it es operando, imprimir it
2. Un operador leído saca de la pila todos los operadores de mayor o igual precedencia, en el orden correspondiente (pop) , y luego se mete él en pila

   >: el más preferente sale antes

   =: por la regla de asociatividad izquierda-derecha, también sale antes
3. Al final, vaciar la pila de operadores e imprimir

Conversión de expresiones: infijo-posfijo

Algoritmo

S1: Inicializar pila auxiliar

S2: Leer la expresión de izquierda a derecha:

S2.1: si el símbolo leído es operando, añadirlo a la expresión de salida

S2.2: si el símbolo leído es operador:

mientras la prec (operador leído) ≤ prec (tope(pila))

extraer elemento de la pila y añadirlo a la expresión de salida

insertar operador leído en la pila

S3: Mientras la pila no esté vacía:

extraer elemento de la pila y añadirlo a la expresión

S4 : Liberar la pila auxiliar

```
A + B * C – D ;
=> ABC*+D-
```

Símbolo	Traducción parcial	Pila de OPERADORES
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
-	ABC*+	-
D	ABC*+D	-
;	ABC*+D-	

Conversión de expresiones: infijo-posfijo (sin CdE)


```
Status infix2posfix (String s1, String s2)
 i = 0
 j = 0

(1) p = stack_init()

 while  s1[i] ≠ EOS:
    if isOperand (s1[i]) == TRUE :
       s2[j] = s1[i]
       j = j + 1

    else if isOperator (s1[i]) == TRUE :
       while (stack_isEmpty(p) == FALSE) AND (prec (s1[i]) ≤ prec(stack_top(p)):
             e = stack_pop(p)
             s2[j] = e
              j = j + 1
    (2) stack_push (p, s1[i])

    i = i + 1

 while stack_isEmpty (p) ==  FALSE :
       e = stack_pop (p)
       s2[j] = e
       j = j + 1
 stack_free (p)
 return OK
```

prec(op): Devuelve el nivel de precedencia de un operador

Conversión de expresiones: infijo-posfijo


```
Status infix2posfix (String s1, String s2)
 i = 0
 j = 0
 st = TRUE

1 if (p = stack_init()) == ERROR :
    return ERROR

 while  s1[i] ≠ EOS AND  st == OK :
     if isOperand (S1[i]) == TRUE :
         s2[j] = s1[i]
         j = j + 1

     else if isOperator (s1[i]) == TRUE :
         while stack_isEmpty(p) == FALSE AND prec (s1[i]) ≤ prec(stack_top(p):
             e = stack_pop(p)
             s2[j] = s[i]
             j = j + 1
2        st = stack_push (p, s1[i])

     i = i + 1

 if st == OK :
     while stack_isEmpty (p) ==  FALSE :
        e = stack_pop (p)
        s2[j] = e
        j = j + 1
 stack_free (p)
 return st
```

Si tuviéramos dentro del while alguna posible situación de error (en otro código diferente) meteríamos la comprobación de st en el while:

```
while st==OK AND condicion_del_bucle:
   //operaciones, alguna de las cuales podría
   //producir error, en cuyo caso:
   // if (x == ERROR) st = ERROR
…
```

Conversión de expresiones: infijo-posfijo

- ¡Ojo! A diferencia de la evaluación, el algoritmo de conversión no detecta errores sintácticos
  - A+B* → AB*+
  - ABC* → ABC*
- **Función de precedencia**

```
int prec(char op)
   switch(op) {
      case '+': case '-': return 1;
      case '*': case '/': return 2;
      case '^': return 3;
   }
   return 0;  // Error
}
```

Conversión de expresiones: infijo-posfijo

- **Con paréntesis…**

```
A * B / (C + D) * (E – F);
```

- Clave: tratar las expresiones entre paréntesis como expresiones en si misma → se traducen antes de seguir la traducción general
  - Ejemplo: `A * B / [CD+] * [EF-] => AB*CD+/EF-*`
- **En la práctica:**
  - ‘(‘ se introduce en la pila
  - cuando se encuentra ‘)’ se sacan todos los operadores hasta ‘(‘
- ¿Hay que tener en cuenta el balanceado de paréntesis?
  - El algoritmo de conversión lo tendrá en cuenta

Conversión de expresiones: infijo-posfijo

- Con paréntesis…

```
A * B / (C + D) * (E – F) ;
```

Símbolo	Traducción parcial	Pila de OPERADORES
A	A	
*	A	*
B	AB	*
/	AB*	/
(	AB*	/ (
C	AB*C	/ (
+	AB*C	/ (+
D	AB*CD	/ (+
)	AB*CD+	/
*	AB*CD+/	*
(	AB*CD+/	* (
E	AB*CD+/E	* (
–	AB*CD+/E	* (–
F	AB*CD+/EF	* (–
)	AB*CD+/EF–	*
;	AB*CD+/EF–*	

Conversión de expresiones: posfijo-prefijo

- Se puede seguir la misma estrategia de evaluación para otras conversiones
- **Posfijo → Prefijo**
  1) Algoritmo de evaluación de expresiones posfijo
  2) En vez de "operar", crear expresiones parciales de tipo prefijo (e introducirlas en la pila, como se hacía al evaluar con el resultado)
- Ejemplo: **`AB*CD+/`**

Conversión de expresiones: posfijo-infijo

- **Posfijo → Infijo**
  1) Algoritmo de evaluación de expresiones posfijo
  2) En vez de "operar", crear expresiones parciales de tipo infijo e introducirlas, entre paréntesis, en la pila
- Ejemplo: **`AB/CD+/EF-*`**

Conversión de expresiones: infijo-prefijo

- **Infijo→ prefijo**

  Algoritmo: infijo → sufijo → prefijo

- Ejemplos:

```
A+B-C;                    (A+B)*(C-D);
```

Repaso

- Evaluar la siguiente expresión sufijo

  `2 1 / 4 2 * + 6 5 - 8 2 / + + ;`

- Convertir la siguiente expresión infijo en sufijo

  `A + B / C - D * F ;`

- Convertir la siguiente expresión infijo en sufijo

  `(A + B) / C * (D - E) + F ;`

- Convertir la siguiente expresión sufijo en infijo

  `A B C / + D F * - ;`

- Convertir la siguiente expresión sufijo en prefijo

  `A B C / + D F * - ;`

- Evaluar la siguiente expresión prefijo

  `/ / 6 5 - 9 3 ;`