Estructuras de Datos

Pilas

Índice

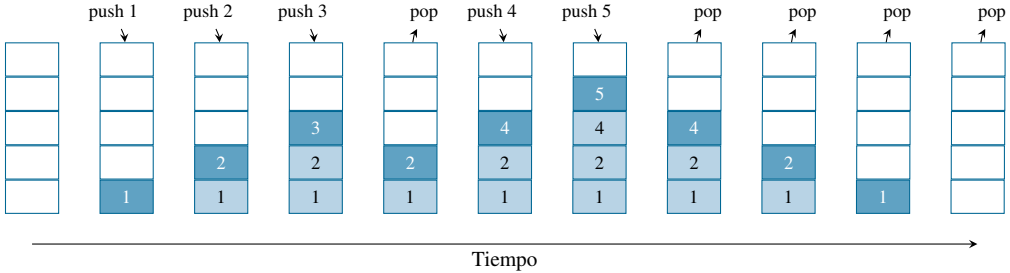
Pila

Una **Pila** es una colección de elementos que son insertados y extraídos siguiendo el principio *Last In, First Out* (LIFO): «el último que entra en la colección es el primero que sale»

TAD Pila

***Last In, First Out* (LIFO):**

- 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 de la pila que se puede «observar»: top (tope, cima)



Aplicaciones de las pilas

En general, todas aquellas aplicaciones que conlleven:

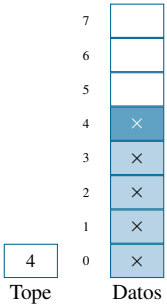
- Algoritmos de **backtracking**, p. ej., búsqueda en profundidad
- Recursión

Algunos **ejemplos**:

- Editores de texto: pila con los últimos cambios realizados
- Navegadores web: pila con las direcciones web visitadas
- Pila de programa: gestiona las llamadas a las funciones
- *Parsing* de código XML/HTML: comprueba el correcto anidamiento de etiquetas `<tag>` `</tag>`
- Comprobación de balanceo de `( )`, `{ }`, `[ ]` en compiladores
- Conversión de expresiones algebraicas

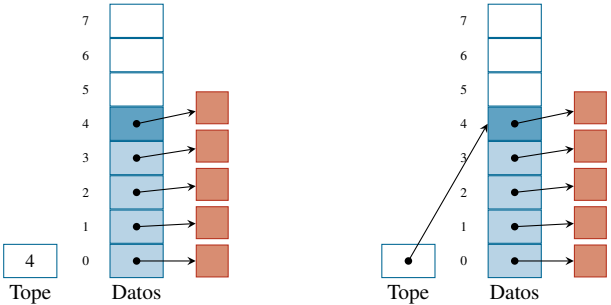
Estructura de datos

- **Array**: conjunto de elementos del mismo tipo almacenados de forma secuencial y con acceso aleatorio (directo)
- **Tope**: Indicador de la posición del último elemento insertado (índice o puntero)



Estructura de datos en C

- **Array**: consideraremos dos posibilidades:
  - Array de punteros de tamaño fijo: `void *data[];`
  - Array de tamaño variable: `void **data;`
- **Tope**: consideraremos dos posibilidades:
  - Índice entero: `int top;`
  - Puntero: `void **top;`



Primitivas

- `Stack stack_new`: crea e inicializa una pila
- `stack_free(Stack s)`: libera la memoria asociada a la pila
- `Boolean stack_isEmpty(Stack s)`: devuelve `True` si la pila está vacía y `False` si no; no modifica la pila
- `Status stack_push(Stack s, Element e)`: inserta un elemento en una pila; modifica la pila cuando devuelve `OK`
- `Element stack_pop(Stack s)`: extrae el dato que ocupa el tope de la pila
- `Element stack_top(Stack s)`: devuelve el dato que ocupa el tope de la pila **sin** extraerlo
- `Integer stack_print(Stream d, Stack s)`: Imprime la pila en un dispositivo y devuelve el número de caracteres impresos

Primitivas en C++

Implementación en C++: `std::stack`

`(constructor)`	Construct stack (public member function)
`empty`	Test whether container is empty (public member function)
`size`	Return size (public member function)
`top`	Access next element (public member function)
`push`	Insert element (public member function)
`emplace`	Construct and insert element (public member function)
`pop`	Remove top element (public member function)
`swap`	Swap contents (public member function)

Implementación en C (I)

EdD para Pila:

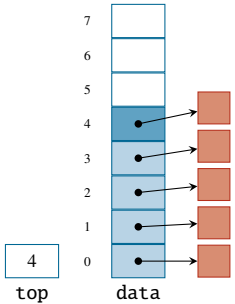
- Implementación con `int top;`

Tipo de dato en `stack.h`

```
typedef struct _Stack Stack;
```

Estructura en `list.c`

```
#define MAX_STACK 8

struct _Stack {
  void *data[MAX_STACK];
  int top;
};
```



Implementación en C (II)

Primitivas:

Cabeceras de las primitivas en `stack.h`

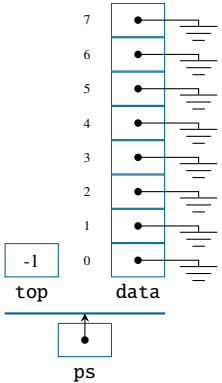
```
Stack *stack_new();
void stack_free(Stack *ps);
Boolean stack_isEmpty(const Stack *ps);
Status stack_push(Stack *ps, const void *e);
void *stack_pop(Stack *ps);
void *stack_top (const Stack *ps);
int stack_size(const Stack *ps);
int stack_print(FILE *fp, const Stack *ps, int (*ele_print)(FILE *, const void *));
```

Implementación en C (III)

Crear e inicializar Pila:

Función `stack_new` en `stack.c`

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

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

  for (i = 0; i < MAX_STACK; i++) {
    ps->data[i] = NULL;
  }
  ps->top = -1;  /* Alt: ps->top = 0; */
  return ps;
}
```



Implementación en C (IV)

Destrucción de Pila:

Función `stack_free` en `stack.c`

```
void stack_free(Stack *ps) {
  free(ps);
}
```

- En esta implementación:
  - La función que reservó la memoria para los elementos *debería* encargarse de liberarlos
  - La asignación `ps = NULL;` debe hacerse después de la llamada a `stack_free(ps)`

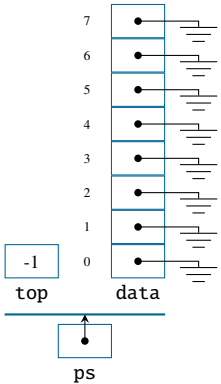
Ejercicio: ¿Y si el prototipo fuese `void stack_free(Stack **pps);`?

Implementación en C (V)

Comprobar si la pila está vacía:

Función `stack_isEmpty` en `stack.c`

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

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

  return FALSE;
}
```



- Esta función debe llamarse antes de extraer
- ¿Es razonable que `ps == NULL` devuelva `TRUE`?

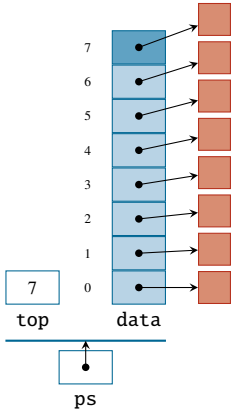
Implementación en C (VI)

Comprobar si la pila está llena:

Función _stack_isFull en stack.c

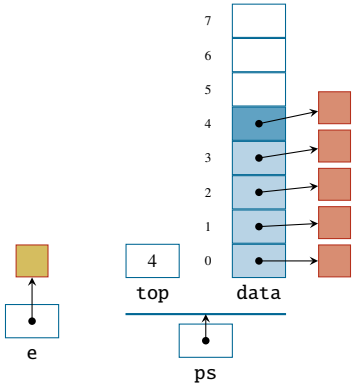
```
Boolean _stack_isFull(const Stack *ps) {
  if (ps == NULL) {
    return TRUE;
  }

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

  return FALSE;
}
```

Nota: Función privada; debe llamarse antes de insertar un elemento para prevenir desbordamiento de la memoria (*stack buffer overflow*)



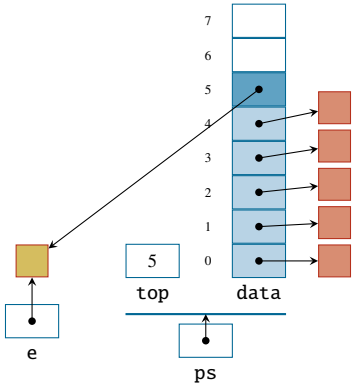
Implementación en C (VII)

Insertar elemento:



Implementación en C (VII)

Insertar elemento:



Implementación en C (VII)

Insertar elemento:

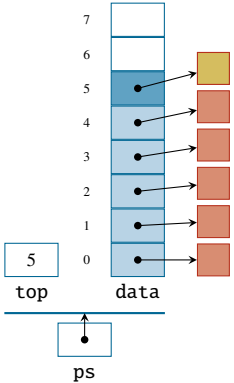
Función `stack_push` en `stack.c`

```
Status stack_push(Stack *ps, const void *e) {
  if (ps == NULL || e == NULL || _stack_isFull(ps) == TRUE) {
    return ERROR;
  }

  ps->top++;
  ps->data[ps->top] =(void *)e;
  /* Alt: ps->data[++ps->top] = (void *)e; */

  return OK;
}
```

Ejercicio: Implementar las funciones `_stack_isFull` y `stack_push` suponiendo `s->top = 0` en `stack_new`

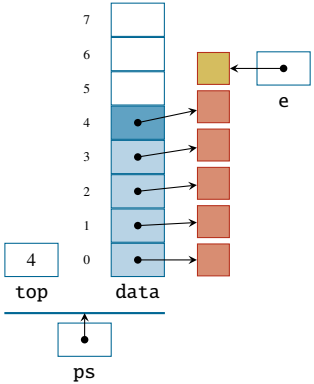
Implementación en C (VIII)

Extraer elemento:



Implementación en C (VIII)

Extraer elemento:



Implementación en C (VIII)

Extraer elemento:

Función `stack_pop` en `stack.c`

```
void *stack_pop(Stack *ps) {
  void *e = NULL;

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

  }

  e = ps->data[ps->top];
  ps->data[ps->top] = NULL;
  ps->top--;
  return e;
}
```

Implementación en C (IX)

Obtener elemento del tope (sin modificar la pila):

Función `stack_top` en `stack.c`

```
void *stack_top(const Stack *ps) {
  if (ps == NULL || stack_isEmpty(ps) == TRUE) {
    return NULL;
  }

  return ps->data[ps->top];
}
```

Implementación en C: Características de la EdD (I)

- Implementación genérica: no depende del tipo de elemento almacenado en la pila: `void *`
- Reutilización de la memoria tras extraer un elemento
- Excepto `stack_new`, todas las operaciones tienen coste (complejidad) $\mathcal{O}(1)$

NO USAR: Inserción ineficiente con complejidad $\mathcal{O}(n)$

```
Status stack_push_inefficient(Stack *ps, const void *e) {
  int i = 0;

  if (ps == NULL || e == NULL || _stack_isFull(ps) == TRUE) {
    return ERROR;
  }

  /* Traverse the array. */
  while (ps->data[i] != NULL) i++;
  /* Insert the element. */
  ps->data[i] = (void *)e;
  /* Increase the top. */
  ps->top++;

  return OK;
}
```

Implementación en C: Características de la EdD (II)

- El tamaño de la pila (`STACK_SIZE`) es constante (array estático)
- La memoria de los elementos se gestiona fuera de la pila

ATENCIÓN: Comportamiento no definido

```
int main() {
  Stack *s = NULL;
  int *ele;

  if (!(ele = (int *)malloc(sizeof(int)))) {
    return EXIT_FAILURE;
  }
  if (!(s = stack_new())) {
    free(ele);
    return EXIT_FAILURE;
  }

  stack_push(s, ele);
  free(ele); /* The stack is corrupted */
  print("%d", stack_size(s));
  /* ... Additional code ... */
}
```

Implementación en C: Ejemplo de uso (I)

```c
#include "stack.h"
#define N_ELE 100

int clean_up(int *ele, Stack *s, Status st) {
  int ret;
  free(ele);
  stack_free(s);
  ret = (st == OK) ? EXIT_SUCCESS : EXIT_FAILURE;
  return ret;
}

int int_print(FILE *f, const void *p) {
  return fprintf(f, "%d ", *((int *)p));
}
```

...

Implementación en C: Ejemplo de uso (II)

```
                                                                    ...
int main() {
  int *ele = NULL, i;
  Status st = OK;
  Stack *s = NULL;

  if (!(s = stack_new())) return clean_up(ele, s, ERROR);

  /* Alloc memory for the elements. */
  if (!(ele = (int *)malloc(N_ELE * sizeof(int)))) return clean_up(ele, s, ERROR);

  /* Assign and push elements in the stack. */
  for (i = 0; i < N_ELE && st == OK; i++) {
    ele[i] = i;
    st = stack_push(s, ele + i); /* Why? Depends on the stack implementation. */
  }

  /* Print stack. */
  stack_print(stdout, s, int_print);

  /* Free memory. */
  return clean_up(ele, s, st);
}
```

Balanceo de paréntesis (I)

Objetivo:

- Determinar si una expresión (p. ej., una operación aritmética) está correctamente balanceada

Correcto:

```
( 3 * 4 / ( 6 - 1 ) )
```

Incorrecto:

```
2 + ( 5 / ( 4 + 7 )
```

```
6 * 4 + 9 )
```

```
6 * ) 4 + 9 (
```

Balanceo de paréntesis (II)

Algoritmo: Comprobación de paréntesis (sin control de errores)

```
   input  : String, expr = {x_1 x_2 ··· x_n}
   output : Boolean, ret
 1 ret = TRUE
 2 s = stack_new()
 3 foreach x in expr do
 4 |   if isOpeningParenth(x) == TRUE then
 5 |   |   stack_push(s, x)
 6 |   else if isClosingParenth(x) == TRUE then
 7 |   |   stack_pop(s)
 8 if stack_isEmpty(s) == FALSE then
 9 |   ret = FALSE
10 stack_free(s)
11 return ret
```

Balanceo de paréntesis (III)

Algoritmo: Comprobación de paréntesis (con control de errores)

```
   input  : String, expr = {x_1 x_2 ··· x_n}
   output : Boolean, ret
 1 st = OK
 2 s = stack_new()
 3 if s == NULL then st = ERROR
 4 foreach x in expr AND st == OK do
 5 |   if isOpeningParenth(x) == TRUE then
 6 |   |   st = stack_push(s, x)
 7 |   else if isClosingParenth(x) == TRUE then
 8 |   |   if stack_pop(s) == NULL then
 9 |   |   |   st = ERROR
10 if st == OK AND stack_isEmpty(s) == FALSE then
11 |   st = ERROR
12 stack_free(s)
13 if st == OK then
14 |   return TRUE
15 else
16 |   return FALSE
```

Balanceo de paréntesis (IV)

Ejercicio:

- Modificar el código anterior para verificar si los símbolos de apertura y cierra `( )`, `[ ]` y `{ }` de una expresión aritmética se hayan correctamente anidados

Correcto:

```
{ 3 * 4 / ( 6 - 1 ) }
```

Incorrecto:

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

```
6 * 4 + 9 }
```

Balanceo de paréntesis (V)

Algoritmo: Comprobación de paréntesis (con diferentes símbolos)

```
   input  : String, expr = {x_1 x_2 ··· x_n}
   output : Boolean, ret
1  st = OK
2  s = stack_new()
3  if s == NULL then st = ERROR
4  foreach x in expr AND st == OK do
5  |   if isOpeningParenth(x) == TRUE then
6  |   |   st = stack_push(s, x)
7  |   else if isClosingParenth(x) == TRUE then
8  |   |   if arePaired(x, stack_top(s)) ==TRUE then
9  |   |   |   stack_pop(s)
10 |   |   else
11 |   |   |   st = ERROR
12 if st == OK AND stack_isEmpty(s) == FALSE then
13 |   st = ERROR
14 stack_free(s)
15 if st == OK then
16 |   return TRUE
17 else
18 |   return FALSE
```

Expresiones algebraicas

Notación infija:

- El operador se coloca entre los operandos
- `A + B`
- `2 - 3 * ( 4 + 1 )`

Notación prefija (polaca):

- El operador precede a los operandos
- `+ A B`
- `- 2 * 3 + 4 1`

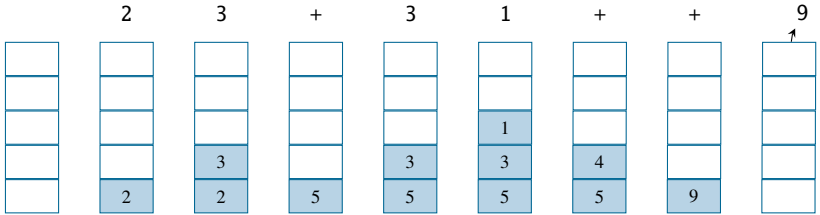
Notación postfija:

- Los operandos preceden al operador
- `A B +`
- `2 3 4 1 + * -`

Evaluación de expresiones posfijas (I)

Ejemplos:

- `2 3 + 3 1 + +`



- `1 2 * 4 + 3 4 + *`

Evaluación de expresiones posfijas (II)

Algoritmo: Evaluación de expresiones posfijas

```
   input  : String, postfix_expr = {x_1 x_2 ··· x_n} ; Integer, r
   output : Status
 1 s = stack_new()
 2 st = OK
 3 foreach x in postfix_expr do
 4 |   if isOperand(x) == TRUE then
 5 |   |   stack_push(s, x)
 6 |   else if isOperator(x) == TRUE then
 7 |   |   arg2 = stack_pop(s)
 8 |   |   arg1 = stack_pop(s)
 9 |   |   e = evaluate(arg1, arg2, x)
10 |   |   stack_push(s, e)
11 r = stack_pop(s)
12 if stack_isEmpty(s) == FALSE then
13 |   st = ERROR
14 stack_free(s)
15 return st
```

Conversión de infijo a posfijo (I)

Algoritmo: Conversión de infijo a posfijo (sin control de errores)

input : String, *infix_expr* = $\{x_1\, x_2\, \cdots x_n\}$; String, *postfix_expr* = $\emptyset$
output : Status

```
1  s = stack_new()
2  foreach x in infix_expr do
3  |   if isOperand(x) == TRUE then
4  |   |   concat(postfix_expr, x)
5  |   else
6  |   |   while prec(x) ≤ prec(stack_top(s)) do
7  |   |   |   e = stack_pop(s)
8  |   |   |   concat(postfix_expr, e)
9  |   |   stack_push(s, x)
10 while stack_isEmpty(s) == FALSE do
11 |   e = stack_pop(s)
12 |   concat(postfix_expr, e)
13 stack_free(s)
14 return OK
```

Conversión de infijo a posfijo (II)

Algoritmo: Conversión de infijo a posfijo (con control de errores)

```
   input  : String, infix_expr = {x_1 x_2 ··· x_n} ; String, postfix_expr = ∅
   output : Status
 1 st = OK
 2 s = stack_new()
 3 if s == NULL then st = ERROR
 4 foreach x in infix_expr AND st == OK do
 5 |   if isOperand(x) == TRUE then
 6 |   |   concat(postfix_expr, x)
 7 |   else
 8 |   |   while stack_isEmpty(s) == FALSE AND prec(x) ≤ prec(stack_top(s)) do
 9 |   |   |   e = stack_pop(s)
10 |   |   |   concat(postfix_expr, e)
11 |   |   st = stack_push(s, x)
12 if st == OK then
13 |   while stack_isEmpty(s) == FALSE do
14 |   |   e = stack_pop(s)
15 |   |   concat(postfix_expr, e)
16 stack_free(s)
17 return st
```

Conversión de infijo a posfijo (III)

Expresiones con **paréntesis**:

- Cuando se encuentra un `(`, se inserta en la pila
- Cuando se encuentra un `)`, se extraen todos los operadores hasta el correspondiente `(` y se añaden a la expresión posfija (el símbolo `(` se extrae también de la pila)
- Ejemplo:
  - Infijo: `A * B / ( C + D ) * ( E - F )`
  - Posfijo: `A B * C D + / E F - *`

Conversión de posfijo a prefijo

- Similar al algoritmo para evaluar una expresión posfijo
- Cuando se lee un operador, en vez de evaluar la expresión e insertar el resultado en la pila, se construye la expresión prefija correspondiente y se inserta en la pila
- Ejemplo:
  - Posfijo: `A B * C D + /`
  - Prefijo: `/ * A B + C D`

Conversión de posfijo a infijo

- Similar al algoritmo para evaluar una expresión posfijo
- Cuando se lee un operador, en vez de evaluar la expresión e insertar el resultado en la pila, se construye la expresión infija con paréntesis correspondiente y se inserta en la pila
- Ejemplo:
  - Posfijo: `A B / C D + / E F - *`
  - Infijo: `( ( A / B ) / ( C + D ) ) * ( E - F )`

Conversión de infijo a prefijo

- Primero se convierte la expresión infija a posfija
- Después se convierte la expresión posfija a prefija
- Ejemplo:
  - Infijo: `( A + B ) * ( C - D )`
  - Prefijo: `* + A B - C D`

Ejercicios

- Evaluar: `2 1 / 4 2 * + 6 5 - 8 2 / + +`
- Convertir a posfijo: `A + B / C - D * F`
- Convertir a posfijo: `( A + B ) / C * ( D - E ) + F`
- Convertir a infijo: `A B C / + D F * -`
- Convertir a prefijo: `A B C / + D F * -`
- Evaluar: `/ / 6 5 - 9 3`