Tema 3.1

Estructuras de Datos y Algoritmos

Tipos de datos lineales. Pilas

Prof. Dr. P. Javier Herrera

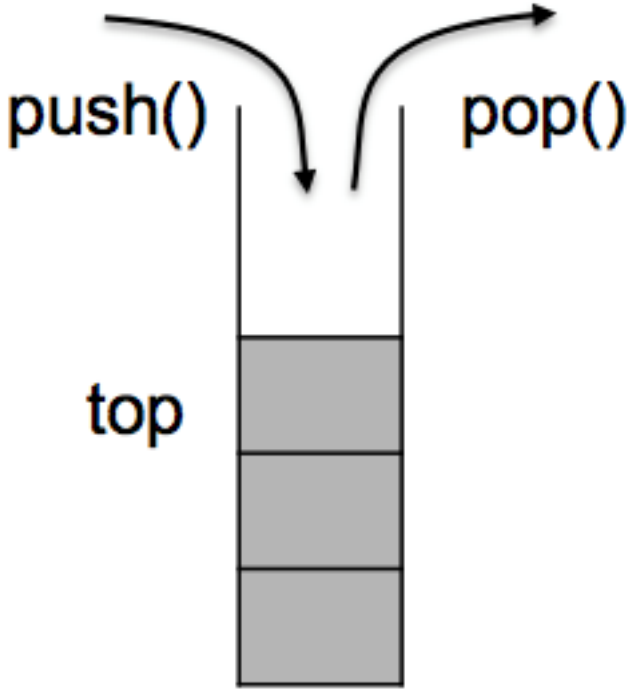
Universidad Francisco de Vitoria
UFV Madrid

Grado en Ingeniería Informática
Escuela Politécnica Superior

Contenido

- Pilas: Conceptos generales
- Operaciones básicas
- Especificación algebraica
- Implementación estática
- Repaso punteros
- Implementación dinámica
- Ejercicio: Evaluación de expresiones en forma postfija

Pila (*Stack*)



Pilas: Conceptos generales

- Estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en orden inverso al de su almacenamiento, siguiendo el criterio de *el último en entrar es el primero en salir*.

- Se las denomina estructuras LIFO (*Last In, First Out*) o pilas. El comportamiento de las pilas es completamente independiente del tipo de los datos almacenados en ellas, por lo que se trata de un tipo de datos parametrizado.

- La ventaja de las pilas es que el acceso a la estructura, tanto para su modificación (inserción y borrado) como para la consulta de los datos almacenados, se realiza en un único punto (la *cima* de la pila), lo que facilita implementaciones sencillas y eficientes.

- A pesar de su sencillez, se trata de una estructura con múltiples aplicaciones en el diseño de algoritmos, como la evaluación de expresiones o la implementación de la recursión.

Operaciones básicas

- El TAD de las pilas cuenta con las siguientes operaciones:
  - crear la pila vacía,
  - apilar un elemento,
  - desapilar el elemento en la cima,
  - consultar el elemento en la cima, y
  - determinar si la pila es vacía.

Especificación algebraica

especificación *PILAS*[*ELEM*]
usa *BOOLEANOS*
tipos *pila*
operaciones

pila-vacía	:		$\rightarrow$ *pila*	{ constructora }
apilar	:	*elemento pila*	$\rightarrow$ *pila*	{ constructora }
desapilar	:	*pila*	$\rightarrow_p$ *pila*	
cima	:	*pila*	$\rightarrow_p$ *elemento*	
es-pila-vacía?	:	*pila*	$\rightarrow$ *bool*	

- Como el orden de apilación es fundamental para la posterior consulta y eliminación, las constructoras son **libres** (no son necesarias ecuaciones de equivalencia).

Especificación algebraica

variables

e : elemento

p : pila

ecuaciones

desapilar(pila-vacía) = error

desapilar(apilar(*e*, *p*)) = *p*

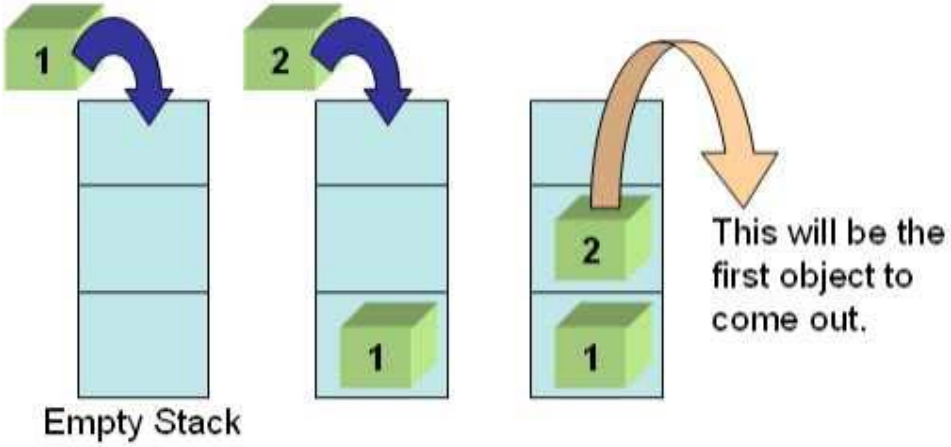
cima(pila-vacía) = error

cima(apilar(*e*, *p*)) = *e*

es-pila-vacía?(pila-vacía) = cierto

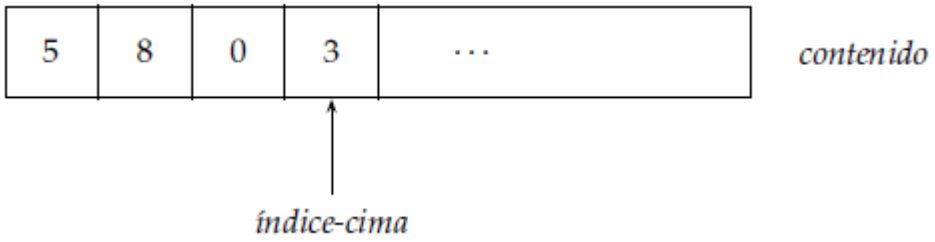
es-pila-vacía?(apilar(*e*, *p*)) = falso

fespecificación



Implementación estática

apilar(3, apilar(0, apilar(8, apilar(5, pila-vacía))))



tipos

pila = **reg**

contenido[1..*N*] **de** elemento

índice-cima : 0..*N*

freg

ftipos

- Almacenamos los elementos de la pila en un vector.
- Mediante un índice apuntamos a la cima.

Implementación estática

```
fun pila-vacía() dev p : pila      {      }
    p.índice-cima := 0             { Cuando la pila está vacía la cima vale 0}
ffun                                                                    O(1)

fun es-pila-vacía?(p : pila) dev b : bool   {      }
    b := (p.índice-cima = 0)                                            O(1)
ffun

proc apilar(e e : elemento, p : pila)  {      }
    si p.índice-cima = N entonces error(Espacio insuficiente)           O(1)
    si no
       p.índice-cima := p.índice-cima + 1
       p.contenido[p.índice-cima] := e
    fsi
fproc
```

Implementación estática

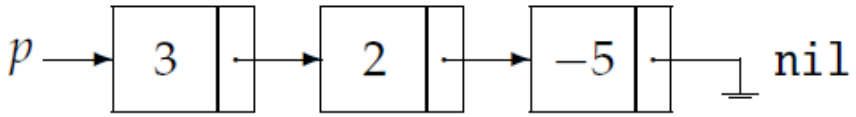
```
proc desapilar(p : pila)    {       }
    si p.índice-cima = 0 entonces error(Pila vacía)
    si no
        p.índice-cima := p.índice-cima − 1        { El elemento no se borra físicamente }
    fsi
fproc

fun cima(p : pila) dev e : elemento {       }
    si p.índice-cima = 0 entonces error(Pila vacía)
    si no
        e := p.contenido[p.índice-cima]
    fsi
ffun
```

$O(1)$

$O(1)$

Implementación dinámica



tipos

enlace-pila = **puntero a** nodo-pila

nodo-pila = **reg**

valor : elemento

sig : enlace-pila

freg

pila = enlace-pila

ftipos

- Representamos la pila como una lista de nodos.
- Cada nodo de la lista tiene dos campos: el contenido y la dirección del siguiente elemento de la lista.
- El campo *sig* del último elemento de la lista no contiene ninguna dirección (NIL).

Implementación dinámica

```
fun pila-vacía() dev p : pila     {      }
    p := nil                                  O(1)
ffun

fun es-pila-vacía?(p : pila) dev b : bool   {      }
    b := (p = nil)                            O(1)
ffun

proc apilar(e e : elemento, p : pila)  {      }
var q : enlace-pila                           O(1)
    reservar(q)
    q↑.valor := e ; q↑.sig := p
    p := q
fproc
```

Implementación dinámica

proc desapilar(*p* : pila) { }
var *q* : enlace-pila
  si *p* = nil **entonces** error(Pila vacía)
  si no
    q := *p* ; *p* := *p*↑.*sig* ; liberar(*q*)
  fsi
fproc

$O(1)$

fun cima(*p* : pila) **dev** *e* : elemento { }
  si *p* = nil **entonces** error(Pila vacía)
  si no *e* := *p*↑.*valor*
  fsi
ffun

$O(1)$

Bibliografía

- Martí, N., Ortega, Y., Verdejo, J.A. *Estructuras de datos y métodos algorítmicos*. Ejercicios resueltos. Pearson/Prentice Hall, 2003. Capítulo 3

- Peña, R.; *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005. Capítulo 6

(Estas transparencias se han realizado a partir de aquéllas desarrolladas por los profesores Clara Segura, Alberto Verdejo y Yolanda García de la UCM, y la bibliografía anterior)