

Estructuras de Datos y Algoritmos

TAD pila genérica (implementación dinámica)

LECCIÓN 8

☺ All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA



Índice

1 Implementación (dinámica)

- Representación de datos
- Implementación del módulo

2 Implementación en C++

3 Consideraciones finales sobre la implementación dinámica

Índice

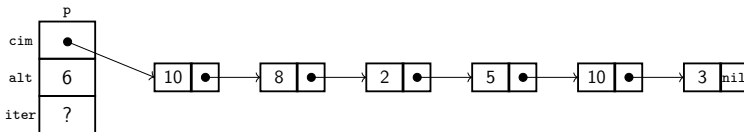
- 1 Implementación (dinámica)
 - Representación de datos
 - Implementación del módulo
- 2 Implementación en C++
- 3 Consideraciones finales sobre la implementación dinámica

TAD pila

Implementación dinámica – representación de datos

```
tipos ptDato = ↑unDato;  
      unDato = registro  
              dato: elemento;  
              sig: ptDato;  
      freg;  
pila = registro  
      cim: ptDato;  
      alt: natural;  
      iter: ptDato; {se utiliza para implementar el iterador  
      freg
```

Sea p una variable de tipo $pila$ de enteros:



TAD pila

Implementación dinámica – módulo genérico pilas

módulo genérico pilas

parámetro

tipo elemento

exporta

tipo pila *{Los valores del TAD pila representan secuencias de elementos con acceso LIFO (last in, first out), esto es, el último elemento añadido será el primero en ser borrado.}*

procedimiento crearVacía(sal p: pila)
{Devuelve en p la pila vacía, sin elementos}

procedimiento apilar(e/s p: pila; ent e: elemento)
{Devuelve en p la pila resultante de añadir e a p}

función esVacía(p: pila) devuelve booleano
{Devuelve verdad si y sólo si p no tiene elementos}

procedimiento cima(ent p: pila; sal e: elemento; sal error: booleano)
*{Si p es no vacía, devuelve en e el último elemento apilado en p y error=falso
Si p es vacía, devuelve error=verdad y e queda indefinido}*

procedimiento desapilar(e/s p: pila)
{Si p es no vacía, devuelve en p la pila resultante de eliminar de p el último elemento que fue apilado. Si p es vacía, la deja igual}

...

...

función altura(p: pila) **devuelve** natural
{Devuelve el número de elementos de p}

procedimiento duplicar(**sal** pilaSal: pila; **ent** pilaEnt: pila)
{Devuelve en pilaSal una pila igual a pilaEnt, duplicando la representación en memoria}

función iguales(pila1, pila2: pila) **devuelve** booleano
{Devuelve verdad si y sólo si pila1 y pila2 tienen los mismos elementos y en las mismas posiciones}

procedimiento liberar(**e/s** p: pila)
{Devuelve en p la pila vacía y además libera la memoria utilizada previamente por p}

...

...

{Las tres operaciones siguientes conforman un iterador interno para la pila}

procedimiento iniciarIterador(e/s p: pila)

{Prepara el iterador para que el siguiente elemento a visitar sea un primer elemento de p, si existe (situación de no haber visitado ningún elemento)}

función existeSiguiende(p: pila) **devuelve** booleano

{Devuelve falso si ya se han visitado todos los elementos de p; devuelve cierto en caso contrario}

procedimiento siguiente(e/s p: pila; **sal** e: elemento; **sal** error: booleano)

{Si existe algún elemento de p pendiente de visitar, devuelve en e el siguiente elemento a visitar y error=falso, y además avanza el iterador para que a continuación se pueda visitar otro elemento de p. Si no quedan elementos pendientes de visitar devuelve error=verdad y e queda indefinido}

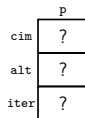
...

[ver implementación completa en el material de clase]

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```



TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);
```

	p
cim	nil
alt	0
iter	?

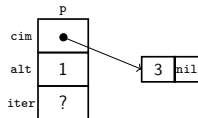
```
procedimiento crearVacía(sal p: pila)
principio
    p.cim := nil;
    p.alt := 0
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);
```



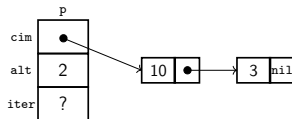
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: pDatao  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);
```



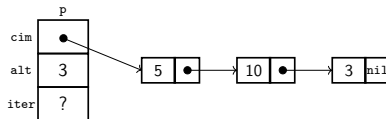
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: ptDato  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);
```



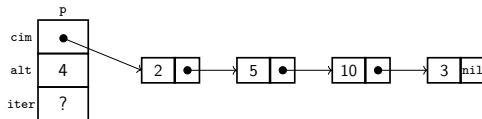
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: ptDato  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);  
apilar(p, 2);
```



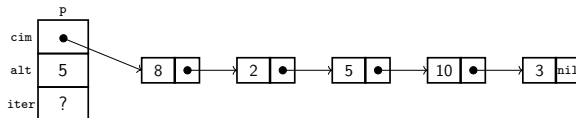
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: ptDato  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);  
apilar(p, 2);  
apilar(p, 8);
```



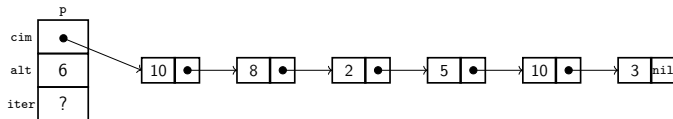
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: ptDato  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);  
apilar(p, 2);  
apilar(p, 8);  
apilar(p, 10);
```



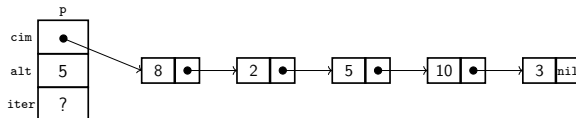
```
procedimiento apilar(e/s p: pila; ent e: elemento)  
variable aux: pDatao  
principio  
    aux := p.cim;  
    nuevoDato(p.cim);  
    p.cim↑.dato := e;  
    p.cim↑.sig := aux;  
    p.alt := p.alt + 1;  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);  
apilar(p, 2);  
apilar(p, 8);  
apilar(p, 10);  
desapilar(p);
```



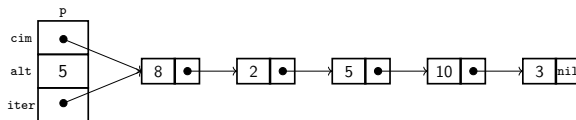
```
procedimiento desapilar(e/s p: pila)  
variable aux: ptDato  
principio  
    si not esVacía(p) entonces  
        aux := p.cim;  
        p.cim := p.cim↑.sig;  
        disponer(aux);  
        p.alt := p.alt - 1  
    fsi  
fin
```

TAD pila

Implementación dinámica – ejemplo

```
p: pilaDeEnteros;
```

```
crearVacía(p);  
apilar(p, 3);  
apilar(p, 10);  
apilar(p, 5);  
apilar(p, 2);  
apilar(p, 8);  
apilar(p, 10);  
desapilar(p);  
iniciarIterador(p);
```



```
procedimiento iniciarIterador(e/s p: pila)  
principio  
    p.iter := p.cim  
fin
```

Índice

1 Implementación (dinámica)

2 Implementación en C++

3 Consideraciones finales sobre la implementación dinámica

TAD pila

Implementación dinámica en C++

// Interfaz del TAD. Pre-declaraciones:

```
template <typename Elemento> struct Pila;

template <typename Elemento> void vacia(Pila<Elemento>& p);
template <typename Elemento> void apilar(Pila<Elemento>& p,
                                         const Elemento& dato);

template <typename Elemento> void desapilar(Pila<Elemento>& p);
template <typename Elemento> void cima(const Pila<Elemento>& p, Elemento& dato,
                                       bool& error);

template <typename Elemento> bool esVacia(const Pila<Elemento>& p);
template <typename Elemento> int altura(const Pila<Elemento>& p);
template <typename Elemento> void duplicar(const Pila<Elemento>& pOrigen,
                                           Pila<Elemento>& pDestino);

template <typename Elemento> bool operator== (const Pila<Elemento>& p1,
                                              const Pila<Elemento>& p2);

template <typename Elemento> void liberar(Pila<Elemento>& p);
template <typename Elemento> void iniciarIterador(Pila<Elemento>& p);
template <typename Elemento> bool existeSiguiente(const Pila<Elemento>& p);
template <typename Elemento> bool siguiente(Pila<Elemento>& p, Elemento& dato);

...
```

```

...
// Declaración

template <typename Elemento> struct Pila{
    friend void vacia<Elemento>(Pila<Elemento>& p);
    friend void apilar<Elemento>(Pila<Elemento>& p, const Elemento& dato);
    friend void desapilar<Elemento>(Pila<Elemento>& p);
    friend void cima<Elemento>(const Pila<Elemento>& p, Elemento& dato,
                               bool& error);
    friend bool esVacia<Elemento>(const Pila<Elemento>& p);
    friend int altura<Elemento>(const Pila<Elemento>& p);
    friend void duplicar<Elemento>(const Pila<Elemento>& pOrigen,
                                    Pila<Elemento>& pDestino);
    friend bool operator==<Elemento> (const Pila<Elemento>& p1,
                                       const Pila<Elemento>& p2);
    friend void liberar<Elemento>(Pila<Elemento>& p);
    friend void iniciarIterador<Elemento>(Pila<Elemento>& p);
    friend bool existeSiguiente<Elemento>(const Pila<Elemento>& p);
    friend bool siguiente<Elemento>(Pila<Elemento>& p, Elemento& dato);

    // Representación de los valores del TAD
private:
    struct unDato {
        Elemento dato;
        unDato* sig;
    };

    unDato* cim ;
    int alt;
    unDato* iter;
};

```

```
// Implementación de las operaciones
```

```
template<typename Elemento> void vacia(Pila<Elemento>& p) {  
    p.alt = 0;  
    p.cim = nullptr;  
}
```

```
template <typename Elemento> void apilar(Pila<Elemento>& p, const Elemento& e){  
    typename Pila<Elemento>::unDato* aux;  
    aux = new typename Pila<Elemento>::unDato;  
    aux -> dato = e;  
    aux -> sig = p.cim;  
    p.cim = aux;  
    p.alt++;  
}
```

```
template<typename Elemento> void iniciarIterador(Pila<Elemento>& p) {  
    p.itr = p.cim;  
}
```

```
// y más operaciones...
```

[ver implementación completa en el material de clase]

Índice

- 1 Implementación (dinámica)
- 2 Implementación en C++
- 3 Consideraciones finales sobre la implementación dinámica

TAD pila

Consideraciones finales sobre la implementación dinámica

- **Coste temporal** de todas las operaciones: $\mathcal{O}(1)$
 - Es decir, **son independientes de la altura** de la pila
 - Como en implementación estática, aunque aquí hay más instrucciones (y algo costosas)
 - Solicitar memoria suele ser *caro* (en tiempo)

TAD pila

Consideraciones finales sobre la implementación dinámica

- **Coste temporal** de todas las operaciones: $\mathcal{O}(1)$
 - Es decir, **son independientes de la altura** de la pila
 - Como en implementación estática, aunque aquí hay más instrucciones (y algo costosas)
 - Solicitar memoria suele ser *caro* (en tiempo)
- **Sin limitación** respecto al máximo de tamaño de la pila en implementación
 - Se reserva memoria con cada nuevo dato
 - La capacidad de almacenamiento depende de la cantidad de memoria disponible
- **Recuerda**: *no olvides sacar la basura...*
 - Recolección de basura manual en C++
- **Ya no es necesario implementar** `apilar` como operación parcial

Método de análisis de coste en 6 pasos

- 1 **Operación y tamaño:** fija la operación a analizar y **define cada variable** de tamaño ($n, k, h, \dots$)
- 2 **Caso:** declara el caso analizado (peor / promedio; con tablas de dispersión, distingue *siempre*)
- 3 **Descomposición:** enumera las sub-operaciones, **incluidas las de los TAD internos**
- 4 **Coste local:** asigna coste a **cada** sub-operación (ninguna se omite)
- 5 **Composición:** secuencia = suma; bucle = producto; alternativa = máximo
- 6 **Resultado:** exprésalo en notación asintótica **solo con variables definidas**, y justifícalo

Errores que evita (familia F4): olvidar sub-operaciones, confundir promedio con peor caso, usar variables sin definir.

TAD pila

El método en acción: apilar (implementación dinámica)

- 1 **Operación y tamaño:** `apilar(p,e)`; sea n la altura de la pila p
- 2 **Caso:** peor caso
- 3 **Descomposición:** pedir memoria para el nuevo registro; copiar e ; enlazar el registro con la antigua cima; actualizar el puntero a cima
- 4 **Coste local:** las cuatro sub-operaciones son $\mathcal{O}(1)$
- 5 **Composición:** secuencia $\Rightarrow$ suma de costes constantes
- 6 **Resultado:** $\mathcal{O}(1)$ en caso peor, **independiente de n** (por eso definimos n : para poder afirmarlo)

Rutina del curso: cerraremos **cada** implementación aplicando estos 6 pasos a sus operaciones

Estructuras de Datos y Algoritmos

TAD pila genérica (implementación dinámica)

LECCIÓN 8

☹ All wrongs reversed – bajo licencia [CC-BY-NC-SA 4.0](#)



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Grado en Ingeniería Informática

UNIVERSIDAD DE ZARAGOZA

