Tecnología de la programación
Sesión 19

Objetivos de la sesión

1. Empezaremos con el tema 6. Hay que estudiarse el TAD Pila (hasta la transparencia 29)
2. Implementaremos como usuarios del TAD Pila una función recursiva para comprobar si dos pilas son iguales (pudiendo destruir las pilas).
3. Deberes: función que compruebe si dos pilas son iguales (versión iterativa que además no destruya las pilas)

Guion

NOTAS PRELIMINARES:

Empezamos tema. Conviene que repaséis antes el tema de punteros, porque en este tema vamos a usar implementaciones dinámicas (basadas en punteros). Dinámico significa que la reserva de memoria se hace en tiempo de ejecución. No vamos a necesitar marcar un tamaño máximo (como necesitamos hacer con vectores), y de esta forma en límite para almacenar elementos va a ser aproximadamente el tamaño de la memoria RAM del equipo.

El tema es una mezcla de los dos anteriores: vamos a definir TADs que vamos a implementar con punteros. En concreto, vamos a definir tres TADs:

1. TAD Pila
2. TAD Cola
3. TAD Lista

Los tres TADs son muy conocidos y usados. Casi la mitad de los ejercicios del examen final tendrán que ver de una forma u otra con este tema. Si entendéis esta sesión del TAD Pila, tenéis la mayor parte del trabajo hecho, porque las ideas del TAD Cola y el TAD Lista son las mismas.

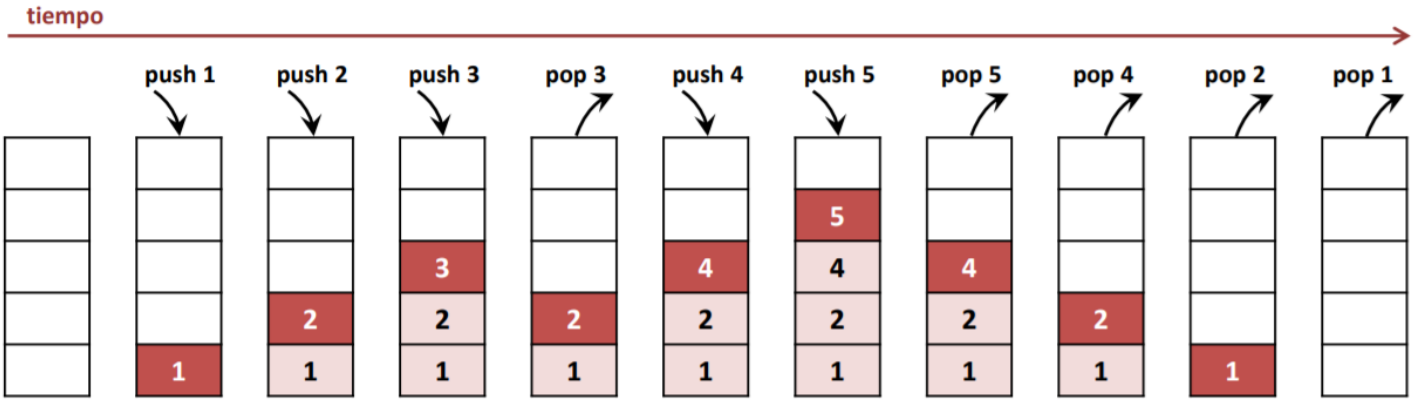
TAD PILA

De nuevo vamos a hacer la especificación, implementación y lo usaremos (sin necesidad de saber cómo está implementado, es decir, basándonos solo en la especificación).

Tenéis que imaginaros una pila como un bloque de libros, uno encima de otro. Solo podemos añadir de uno en uno (arriba del todo) y coger el que tengamos más arriba.

Es decir, el último elemento que llega es el primero que sale. Resumiendo:

- Los elementos se insertan de uno en uno (arriba del todo): operación apilar (push en inglés).
- Los elementos se extraen de uno en uno (arriba del todo): operación desapilar (pop en inglés)
- El último elemento insertado (que será el primero en ser extraído) es el único que se puede "observar" de la pila: operación cima (top en inglés)



Podemos crear pilas de cualquier cosa: pila de enteros, pila de booleanos, pila de caracteres o incluso pilas de cualquier otro TAD (por ejemplo, una pila de tEstudiantes, de libros o de números complejos). Para considerar el caso general, diremos que **nuestra pila es de elementos de tipo tElemento**.

Como operaciones del TAD, tendremos constructores (iniciar pila vacía, apilar), accesores (acceder a la cima), deconstructores (eliminar la cima, es decir, desapilar), operaciones básicas (mostrar, copiar, crear una pila a partir de una secuencia) y auxiliares (saber si una pila es vacía).

Es decir, nuestra especificación sería la siguiente:

ESPECIFICACIÓN TAD PILA

```
TAD Pila(tElemento) // Pila formada por elementos de tipo tElemento
usa
  tElemento
genero
  pila
```

Aquí ponemos que usa tElemento, para decir que es sobre un tipo genérico cualquiera

operaciones

```
  acción iniciarPila (sal pila p)
  {Pre: }
  {Post: inicia p como una pila vacía}

  acción apilar (e/s pila p, ent tElemento d)
  {Pre: p es una pila que ha sido iniciada previamente}
  {Post: apila en p el elemento d}

  función pilaVacia (pila p) dev booleano
  {Pre: p es una pila que ha sido iniciada previamente}
  {Post: dev verdad si p está vacía y falso en caso contrario}

  función cima (pila p) dev tElemento
  {Pre: p es una pila no vacía}
  {Post: devuelve el último elemento apilado, no modifica la pila}

  acción desapilar (e/s pila p)
  {Pre: p es una pila no vacía}
  {Post: modifica la pila p, eliminando el último elemento apilado}

  acción crearPila (s/ pila p)
  {Pre: el usuario introducirá por pantalla una secuencia de elementos
con una marca de finalización}
  {Post: la pila p contiene la secuencia elementos pedida al usuario}

  acción copiarPila(e/ pila p, s/ pila p2)
  {Pre: p es una pila que ha sido iniciada previamente}
  {Post: p2 es una copia de p, p no se destruye}

  acción mostrarPila(e/ pila p)
  {Pre: p es una pila que ha sido iniciada previamente}
  {Post: muestra los elementos de p en orden inverso al que han sido
apilados, p no se destruye}
```

fin_especificación

Vamos a implementar este TAD de dos formas:

1. Versión estática (basada en vectores)
2. Versión dinámica (basada en punteros)

NOTA: Con la primera versión estaremos limitados a poner un tamaño máximo a la pila.

Antes de nada, conviene recordar que cuando usamos un TAD en general, no sabemos cómo está implementado. Si utilizamos una representación dinámica (punteros) para el TAD Pila, desapilar liberará memoria. Por lo tanto, aunque trabajemos con parámetros de entrada, la pila podría quedar destruida.

Para las pilas, y en general cualquier TAD, siempre conviene usar parámetros de entrada/salida cuando trabajemos como usuarios.

Habrá veces que interesa mantener la pila, y para ello la reconstruiremos o haremos una copia antes. Si vamos a destruirla, es importante decirlo en la postcondición para evitar comportamientos inesperados.

IMPLEMENTACIÓN ESTÁTICA DEL TAD PILA

La implementación es sencilla. Es simplemente un registro con dos campos: un vector donde vamos a ir almacenando los elementos (hasta un máximo MAXPILA), y una variable de tipo entero donde se guarda el número de elementos que hay en la pila en ese momento. La implementación de los métodos es fácil: es simplemente trabajar con las componentes del vector.

```
Representación
constante
      MAXPILA = … // representa tamaño máximo de la pila
tipo
      tipodef tElemento : tvector[MAXPILA]
      pila = registro
            tvector datos
            entero numElem
      freg

Interpretación
La pila tiene numElem elementos almacenados en las numElem primeras
componentes del vector datos de forma que la cima está en la
componente de numElem-1

acción iniciarPila (sal pila p)
{Pre: }
{Post: inicia p como una pila vacía}
principio
      p.numElem = 0
fin

acción apilar (e/s pila p, ent tElemento d)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: apila en p el elemento d}
principio
      si p.numElem < MAXPILA entonces
            p.datos[p.numElem] = d
            p.numElem = p.numElem+1
    fsi
fin

función pilaVacia (pila p) dev booleano
{Pre: p es una pila que ha sido iniciada previamente}
{Post: dev verdad si p está vacía y falso en caso contrario}
principio
      dev(p.numElem == 0)
fin

función cima (pila p) dev tElemento
```

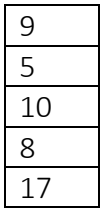
Esta sintaxis a veces os sorprende, es para hacer la comprobación dentro del propio devuelve.

```
{Pre: p es una pila no vacía}
{Post: devuelve el último elemento apilado, no modifica la pila}
principio
      dev(p.datos[p.numElem-1])
fin

acción desapilar (e/s pila p)
{Pre: p es una pila no vacía}
{Post: modifica la pila p, eliminando el último elemento apilado}
principio
      p.numElem = p.numElem-1
fin

acción crearPila (s/ pila p)
{Pre: el usuario introducirá por pantalla una secuencia de elementos
con una marca de finalización}
{Post: la pila p contiene la secuencia elementos pedida al usuario}
variables
     tElemento n
principio
      p.numElem = 0
      escribir("Introduce los elementos de la pila y finaliza con un
      0")
      leer(n)
      mientras que n != 0 AND p.numElem < MAXPILA hacer
          p.datos[p.numElem] = d
          p.numElem = p.numElem+1
          leer(n)
      fmq
fin

acción copiarPila(e/ pila p, e/s pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}
variables
    entero i
principio
    p2.numElem = p.numElem
    para i = 0 hasta numElem - 1 hacer
        p2.datos[i] = p.datos[i]
    fpara
fin

acción mostrarPila(e/ pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido
apilados, p no se destruye}
variables
    entero i
principio
    para i = p.numElem - 1 descendiendo hasta 0 hacer
        escribir (p.datos[i])
    fpara
fin
```

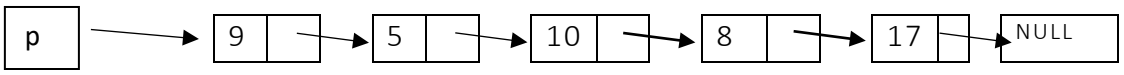
IMPLEMENTACIÓN DINÁMICA DEL TAD PILA

Para no tener la restricción de un número máximo de elementos (la constante MAXPILA), podemos hacer una implementación basada en punteros. La modelaremos como una especie de "lista enlazada de nodos" pero donde solo podamos añadir y eliminar al principio. Otra alternativa podría ser que solo pudiésemos añadir y eliminar al final, pero sería más ineficiente porque nos obligaría a recorrer toda la lista para añadir cada elemento (sería O(n), mientras que si añadimos al principio es O(1)).

Es decir, si queremos modelar una pila en la que hayamos apilado los elementos 17, 8, 10, 5 y 9 (en ese orden, es decir, el último apilado es el 9).



Entonces podremos representarlo con punteros y nodos de esta forma:



Es decir, lo único que necesitamos son nodos (que almacenen un tElemento y un puntero al siguiente nodo), y la pila será simplemente un puntero al primer nodo.

Ayuda para entenderlo: en la implementación estática, usábamos un vector donde guardábamos los elementos de izquierda a derecha según nos iban llegando. Aquí tendremos una lista enlazada de nodos y guardaremos de derecha a izquierda, porque vamos a ir añadiendo al principio.

```
implementación TAD Pila
Representación
tipo
      Nodo = registro
            tElemento dato
            puntero a Nodo sig
      freg
      pila = puntero a Nodo
```

Nodo es un registro con dos campos, el dato (de tipo tElemento) y sig, un puntero al siguiente nodo.

Pila es un puntero al primer nodo.

Interpretación

Pila es un puntero a Nodo que contiene la cima y un puntero que apunta a un nodo que contiene el dato que ha llegado anteriormente a la pila.

El puntero del nodo correspondiente al primer dato apilado apunta a NULL. Si la pila está vacía, el puntero apunta a NULL.

En implementaciones dinámicas, no os olvidéis de decir que lo último apunta a NULL, para poder saber cuándo acaba.

```
acción iniciarPila (sal pila p)
{Pre: }
{Post: inicia p como una pila vacía}
principio
      p = NULL
fin

acción apilar (e/s pila p, ent/ tElemento d)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: apila en p el elemento d}
variables
      puntero a Nodo nuevo
principio
      nuevo = reservar(Nodo)
      dest(nuevo).dato = d
      dest(nuevo).sig = p
      p = nuevo
fin
```

Es simplemente la operación de añadir al principio de una lista enlazada de nodos.

```
función pilaVacia (pila p) dev booleano
{Pre: p es una pila que ha sido iniciada previamente}
{Post: dev verdad si p está vacía y falso en caso contrario}
principio
      dev(p == NULL)
fin

función cima (pila p) dev tElemento
{Pre: p es una pila no vacía}
{Post: devuelve el último elemento apilado, no modifica la pila}
principio
      dev(dest(p).dato)
fin

acción desapilar (e/s pila p)
{Pre: p es una pila no vacía}
{Post: modifica la pila p, eliminando el último elemento apilado}
variables
      puntero a Nodo aux
principio
      aux = p
      p = dest(p).sig
      liberar(aux)
fin

acción crearPila(sal/ pila p)
variables
     tElemento n
principio
      p=NULL
```

```
      escribir("Introduce los elementos y finaliza con un 0")
      leer(n)
      mientras que n != 0 hacer
          nuevo = reservar(Nodo)
          dest(nuevo).dato = d
          dest(nuevo).sig = p
          p = nuevo
          leer(n)
      fmq
fin


acción copiarPila(e/ pila p, sal/ pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}
variables
    puntero a Nodo nuevo, aux
principio
    si p == NULL //Si la pila es vacía
        p2 = NULL
    si_no
        nuevo = reservar(Nodo)
        p2 = nuevo
        aux = p
        mientras que dest(aux).sig != NULL
            dest(nuevo).dato = dest(aux).dato
            aux = dest(aux).sig
            dest(nuevo).sig=reservar(Nodo)
            nuevo = dest(nuevo).sig
         fmq
         //Copiamos el último
         dest(nuevo).dato = dest(aux).dato
         dest(nuevo).sig = NULL
    fsi
fin


acción mostrarPila(e/ pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido
apilados, p no se destruye}
variables
    puntero a Nodo aux
principio
    aux = p
    mientras que aux != NULL
        escribir(dest(aux).dato)
        aux = dest(aux).sig
    fmq
fin


fin_implementación
```

Este método es el más difícil, échale un ojo con detalle y trata de entenderlo.

¿Podríamos haber implementado el **copiarPila, mostrarPila y crearPila** como usuarios, es decir, haciendo llamadas al resto de las funciones y acciones de la especificación del TAD (apilar, desapilar, …)?

Como esas operaciones pertenecen a la especificación, no es conveniente. La idea hubiese sido hacer lo siguiente:

```
acción crearPila(e/s pila p)
variables
     tElemento n
principio
      iniciarPila(p)
      escribir("Introduce los elementos y finaliza con un 0")
      leer(n)
      mientras que n != 0 hacer
          apilar(p,n)
          leer(n)
      fmq
fin


acción copiarPila(e/s pila p, e/s pila p2)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: p2 es una copia de p, p no se destruye}
variables
    pila aux
principio
    iniciarPila(aux)
    iniciarPila(p2)
    mientras que NOT pilaVacia(p) hacer
        apilar(aux,cima(p))
        desapilar(p)
    fmq
    mientras que NOT pilaVacia(aux) hacer
        apilar(p,cima(aux))
        apilar(p2,cima(aux))
        desapilar(aux)
    fmq
fin


acción mostrarPila(e/s pila p)
{Pre: p es una pila que ha sido iniciada previamente}
{Post: muestra los elementos de p en orden inverso al que han sido
apilados, p no se destruye}
variables
    pila aux
principio
    iniciarPila(aux)
    copiarPila(p,aux)
    mientras que NOT pilaVacia(aux)
        escribir(cima(aux))
        desapilar(aux)
```

```
    fmq
fin
```

Como veis, el código incluso se vuelve más entendible. Sin embargo, es algo no se debería hacer. Siempre que estemos implementando una operación que aparece en la especificación, deberíamos aprovecharnos al máximo de la representación elegida para hacer el código eficiente.

En las acciones anteriores, no nos estamos aprovechando de la implementación y esto provoca que sea menos eficiente. Por ejemplo, en el `copiarPila` necesitamos crear otra variable auxiliar de tipo pila, volcar los datos de la pila original a la auxiliar (el primer bucle, de esta forma los tendremos invertidos) y volver a volcarlos (segundo bucle) de la auxiliar tanto a la de destino y como a la original (para tener la copia y para reconstruir la original, porque habíamos desapilado).

Algo similar pasa en el método `mostrarPila`: con la versión de implementador podemos recorrerla directamente con los punteros. Sin embargo, con la versión de usuario tenemos que hacer primero una copia y luego recorrer la copia desapilando elementos.

En resumen: aunque la versión de usuario tiene mejor legibilidad del código, si una operación pertenece a la especificación es mejor implementarla directamente usando la representación: se podrá hacer más eficiente y además se ahorrarán llamadas al *call stack*.

¿En caso de que `copiarPila`, `mostrarPila` y `crearPila` no perteneciesen a la especificación, cómo las implementamos?

Habría que implementarlas como usuarios, es decir, sin saber la implementación del TAD Pila (no sabríamos si es con vectores, punteros u otra alternativa). Lo tendríamos que hacer precisamente como en el código mostrado en el punto anterior (donde usamos `apilar`, `desapilar`, `iniciarPila`, `pilaVacia`, etc).

¿Por qué `copiarPila` implementada como USUARIO, su primer parámetro es de entrada/salida y no únicamente de entrada, a pesar de que hayamos reconstruido la pila?

Estamos trabajando con punteros. Es cierto que la pila p primero la destruimos y luego la reconstruimos con los mismos elementos y en el orden original. **Pero esto SÍ que provoca que la pila original cambie**. La pila original p apuntaba a unas direcciones de memoria que hemos liberado (al desapilar), y luego hemos reservado otras direcciones de memoria donde hemos guardado los elementos originales. Lo mismo pasa con la acción `mostrarPila` implementada como usuario, porque nos basamos en el `copiarPila` implementado como usuario.

Es decir, cuando trabajamos como usuarios con un TAD, mejor ponerlos siempre como parámetros de entrada/salida, a no ser que estemos totalmente seguros. Por eso en la implementación como usuarios de `copiarPila`, `mostrarPila` y `crearPila`, todos los parámetros que sean pilas aparecen de entrada/salida.

¿Podríamos haber implementado recursivamente las operaciones?

Muchas de ellas sí. Por ejemplo, el `mostrarPila` (el siguiente ejemplo muestra la implementación recursiva cuando la operación pertenece a la especificación).

```
acción mostrarPilaR(e/ pila p)
principio
    si NOT pilaVacia(p)
        escribir(dest(p).dato)
        mostrarPilaR(dest(p).sig)
    fsi
fin
```

EJERCICIO: función recursiva COMO USUARIOS que compruebe que si dos pilas son iguales (pueden destruirse las pilas).

Recordad, la estamos implementando como usuarios. Solo podemos usar las operaciones del TAD y no sabemos qué implementación hay por debajo.

```
función iguales(pila p1, pila p2) dev booleano
{Pre: p1 y p2 son dos pilas iniciadas.}
{Post: devuelve verdad si ambas pilas tienen almacenados exactamente
los mismos elementos en el mismo orden y falso en caso contrario. Las
pilas p1 y p2 pueden quedar destruidas.}
principio
    si pilaVacia(p1) AND pilaVacia(p2)
        dev VERDAD
    si_no
        si pilaVacia(p1) AND NOT pilaVacia(p2)
            dev FALSO
        si_no
            si NOT pilaVacia(p1) AND pilaVacia(p2)
                dev FALSO
            si_no
                si cima(p1) != cima(p2)
                    dev FALSO
                si_no
                    desapilar(p1)
                    desapilar(p2)
                    dev iguales(p1,p2)
                fsi
            fsi
        fsi
    fsi
fin
```

¡¡Podríamos habernos cargado las pilas dependiendo de la implementación, porque hemos llamado al desapilar!! Por eso se recomienda trabajar con acciones y poner siempre que los TADs son de entrada/salida cuando trabajemos como usuarios.

Se debería adaptar la función anterior a una acción:

```
acción iguales(e/s pila p1, e/s pila p2, sal/ booleano resultado)
Principio
    si pilaVacia(p1) AND pilaVacia(p2)
        resultado = VERDAD
    si_no
```

```
        si pilaVacia(p1) AND NOT pilaVacia(p2)
            resultado = FALSO
        si_no
            si NOT pilaVacia(p1) AND pilaVacia(p2)
                resultado = FALSO
            si_no
                si cima(p1) != cima(p2)
                    resultado = FALSO
                si_no
                    desapilar(p1)
                    desapilar(p2)
                    iguales(p1,p2,resultado)
                fsi
            fsi
        fsi
    fsi
fin
```

Deberes: algoritmo iterativo para ver si dos pilas son o no iguales, sin destruir las pilas (haciendo copias)

Consejo final: al igual que en cualquier ejercicio de punteros, lo mejor es ir dibujando lo que hace cada paso para entenderlo bien. Sobre todo el copiarPila (la versión que pertenece a la especificación), porque es el más costoso de entender.