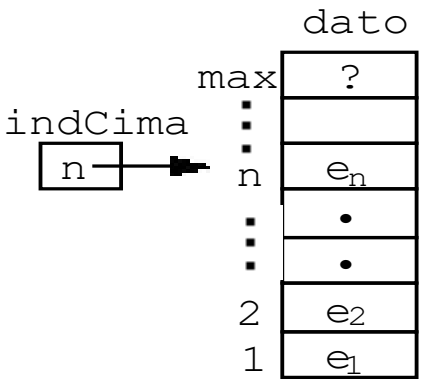
Representación estática de pilas genéricas e implementación de operaciones

La representación estática o **contigua** de una pila consiste en un vector con espacio para un máximo `max` de elementos y un contador `cima`, `0 ≤ cima ≤ max`, que indica el número de elementos válidos almacenados en el vector. Es decir:

```
constante max = ...
tipo pila = registro
              dato: vector[1..max] de elemento;
              indCima,iter: 0..max
            freg
```

Donde `elemento` es un tipo previamente definido. Si la pila `p` es no vacía, el elemento almacenado en `p.dato[p.indCima]` corresponde a la cima de la pila, mientras que el elemento `p.dato[1]` es el fondo de la pila (el elemento que fue apilado en primer lugar). La pila vacía se representa con `p.indCima=0`. El campo `iter` se utilizará para implementar las operaciones del iterador.



A continuación, presentamos una implementación del tipo:

```
módulo genérico pilas
parámetro
  tipo elemento
exporta
  constante max = 100[1] {Es la altura máxima de cualquier pila
                          representable en el tipo pila, limitada por
                          una decisión de implementación.}
  tipo pila
   {Los valores del TAD pila representan secuencias de longitud
   menor o igual que el valor de la constante 'max' de elementos
   con acceso LIFO (last in, first out), esto es, el último
   elemento añadido (o apilado) será el primero en ser borrado (o
   desapilado)}

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

  procedimiento apilar(e/s p:pila; ent e:elemento;
                       sal error:booleano)
  {Si altura(p)<max), entonces devuelve en el mismo parámetro p la
   pila resultante de añadir e a p, y error=falso. En caso
   contrario, devuelve error=verdad y no modifica p.}

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

[1] Es un valor arbitrario.

```
  función cima(p:pila) devuelve elemento
  {Pre: p es no vacía} {Devuelve el último elemento apilado en p}

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

  función altura(p:pila) devuelve natural
  {Devuelve el nº de elementos de p, 0 si no tiene elementos}

  procedimiento iniciarIterador(e/s p:pila)
  {Prepara el cursor del iterador para que el siguiente elemento a
   visitar sea la cima de la pila p, si existe (situación de no
   haber visitado ningún elemento)}

  función existeSiguiente(p:pila) devuelve booleano
  {Devuelve falso si ya se han visitado todos los elementos de p.
   Devuelve verdad en caso contrario.}

  procedimiento siguiente(e/s p:pila; sal e:elemento;
                          sal error:booleano)
  {Implementa las operaciones "siguiente" y "avanza" de la
   especificación, es decir: Si existeSiguiente(p), error toma el
   valor falso, e toma el valor del siguiente elemento de la pila,
   y se avanza el cursor del iterador al elemento siguiente de la
   pila. Si no existeSiguiente(p), error toma el valor verdad, e
   queda indefinido y p queda como estaba.}

implementación

  tipo pila = registro
                dato: vector[1..max] de elemento;
                indCima,iter: 0..max
              freg

  procedimiento crearVacía(sal p:pila)
  {Devuelve una pila vacía, sin elementos}
  principio
    p.indCima:=0
  fin

  procedimiento apilar(e/s p:pila; ent e:elemento;
                       sal error:booleano)
  {Si altura(p)<max), entonces devuelve en el mismo parámetro p la
   pila resultante de añadir e a p, y error=falso. En caso
   contrario, devuelve error=verdad y no modifica p.}
  principio
    si p.indCima<max entonces
      error:=falso;
      p.indCima:=p.indCima+1;
      p.dato[p.indCima]:=e
    sino
      error:=verdad
    fsi
  fin
```

```
procedimiento desapilar(e/s p:pila)
{Si p es no vacía, devuelve la pila resultante de eliminar de p
 el último elemento que fue apilado. Si p es vacía la deja igual}
principio
  si p.indCima>0 entonces
    p.indCima:=p.indCima-1
  fsi
fin
```

Versión NO robusta:

```
función cima(p:pila) devuelve elemento
{Pre: p es no vacía} {Devuelve el último elemento apilado en p}
principio
  devuelve(p.dato[p.indCima])
fin
```

Versión robusta:

```
procedimiento cima(ent p:pila; sal e:elemento;
                   sal error:booleano)
{Si p es vacía, error toma el valor verdad y se deja e
 indefinido. Si p no es vacía, error toma el valor falso y e toma
 el valor de la cima de p.}
principio
  si esVacía(p) entonces
    error:=verdad
  sino
    error:=falso;
    e:=p.dato[p.indCima]
  fsi
fin

función esVacía(p:pila) devuelve booleano
{Devuelve verdad si y sólo si p no tiene elementos}
principio
  devuelve(p.indCima=0)
fin

función altura(p:pila) devuelve natural
{Devuelve el nº de elementos de p, 0 si no tiene elementos}
principio
  devuelve p.indCima
fin

procedimiento iniciarIterador(e/s p:pila)
{Prepara el iterador para que el siguiente elemento a visitar sea
 la cima de la pila p, si existe (situación de no haber visitado
 ningún elemento)}
principio
  p.iter:=p.indCima
fin
```

```
  función existeSiguiente(p:pila) devuelve booleano
  {Devuelve falso si ya se han visitado todos los elementos de p.
   Devuelve verdad en caso contrario.}
  principio
    devuelve (p.iter>0)
  fin

  procedimiento siguiente(e/s p:pila; sal e:elemento;
                          sal error:booleano)
  {Implementa las operaciones "siguiente" y "avanza" de la
   especificación, es decir: Si existeSiguiente(p), error toma el
   valor falso, e toma el valor del siguiente elemento de la pila,
   y se avanza el iterador al elemento siguiente de la pila.
   Si no existeSiguiente(p), error toma el valor verdad, e queda
   indefinido y p queda como estaba.}
  principio
    si existeSiguiente(p) entonces
      error:=falso;
      e:=p.dato[p.iter];
      p.iter:=p.iter-1
    sino
      error:=verdad
    fsi
  fin

fin {del módulo}
```

El coste en tiempo de todas las operaciones es $O(1)$, es decir, independiente de la altura de la pila.

El mayor inconveniente de esta representación (estática) es que debe reservarse espacio (tamaño del vector) para el máximo previsto de elementos, aunque luego no se alcance al ejecutar el programa usuario. Además, **la operación `apilar` hay que implementarla como si fuese una operación parcial**, puesto que al intentar apilar un dato cuando `indCima = max`, se produce un error. A pesar de este problema, consideramos válida esta representación, de la misma forma que en los tipos predefinidos, como los enteros o los reales, toda implementación impone restricciones de tamaño o redondeo que no están presentes en la especificación del tipo (no se puede representar un tipo con un conjunto de valores de cardinal infinito en un espacio finito).

Además de las operaciones especificadas e implementadas anteriormente, toda implementación de un contenedor o colección de datos, debería incluir además operaciones básicas para **duplicar** la representación de un valor del TAD, es decir, hacer una copia de una variable del TAD en otra distinta y de **comparación de igualdad** entre dos valores del TAD. Para el tipo pila implementado aquí, serían las siguientes:

```
procedimiento duplicar(ent pEnt:pila; sal pSal:pila)
{Hace una copia en pSal de la pila almacenada en pEnt.}
Variable i:natural
principio
  pSal.indCima:=pEnt.indCima;
  si not esVacía(pEnt) entonces
    para i:= 1 hasta pEnt.indCima hacer
      pSal.dato[i]:=pEnt.dato[i]  (*)
    fpara
  fsi
fin

función iguales(p1,p2:pila) devuelve booleano
{Devuelve verdad si y sólo si p1 y p2 almacenan la misma pila.}
variables igual:booleano; i:natural
principio
  si esVacía(p1) and esVacía(p2) entonces
    devuelve verdad
  sino_si altura(p1)≠altura(p2) entonces
    devuelve falso
  sino {ambas pilas tienen el mismo número, no nulo, de elementos}
    igual:=verdad;
    i:=p1.indCima;
    mientrasQue igual and i>0 hacer
      igual:= (p1.dato[i]=p2.dato[i]); (**)
      i:=i-1
    fmq;
    devuelve igual
  fsi
fin
```

Debemos destacar que en la instrucción `(**)` de esta última función, estamos comparando dos datos de tipo "elemento", que no sabemos cuál es. Si no es un tipo predefinido (que permita esa comparación de igualdad con el operador "="), deberemos usar una función "`iguales(e1,e2)`", en este caso "`iguales(p1.dato[i],p2.dato[i])`" que debería existir para el tipo de dato "elemento", y que sirva para comparar valores de ese tipo.

Algo similar sucede con las operaciones en las que asignamos un elemento a una componente del vector campo "dato" del tipo "pila". Por ejemplo, cuando en el código de la operación duplicar escribimos `(*)`: `pSal.dato[i]:=pEnt.dato[i]`. Si el tipo elemento se particulariza en un tipo no predefinido, la operación de asignación (":=") entre datos de tipo "elemento" debería sustituirse por una llamada a un procedimiento para "duplicar elementos", similar al procedimiento de duplicar pila que hemos implementado aquí.

Una forma de resolver este problema es exigir al parámetro formal de tipo “elemento” que disponga de sendas operaciones “:=” e “=” (podrían llamarse alternativamente “duplicar” e “iguales”) para duplicar su representación (lo que habitualmente llamamos “asignación entre variables”) y para realizar la comparación de igualdad entre elementos, respectivamente. En ese caso, el inicio del módulo genérico de implementación de pilas sería el siguiente, exigiendo que el parámetro formal de tipo “elemento” disponga de esas dos operaciones:

```
módulo genérico pilas
parámetros
  tipo elemento
  con procedimiento duplicar(ent eEnt:elemento; sal eSal:elemento)
      {procedimiento que duplica la representación de eEnt en eSal}
  con función iguales(e1,e2:elemento) devuelve booleano
      {función que devuelve verdad si y sólo si e1 y e2 almacenan
       el mismo elemento}
exporta
  ...
```