11. PILAS

11.0 Introducción

En este tema y en el siguiente veremos las estructuras de datos lineales pilas y colas. Las pilas y las colas son dos de las estructuras de datos más utilizadas. Se trata de dos casos particulares de las estructuras lineales generales (secuencias o listas) que, debido a su amplio ámbito de aplicación, conviene ser estudiadas de manera independiente. En este tema veremos concretamente las pilas, su utilización y su implementación más habitual en C++.

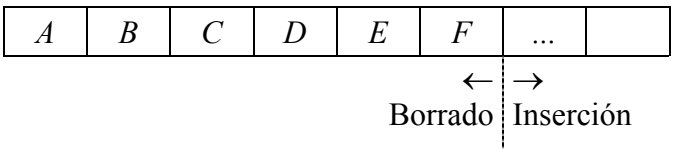
11.1 Fundamentos

La pila es una lista de elementos caracterizada porque las operaciones de inserción y eliminación de elementos se realizan solamente en un extremo de la estructura. El extremo donde se realizan estas operaciones se denomina habitualmente cima (*top* en la nomenclatura inglesa).

Dada una pila *P*, formada por los elementos *a*, *b*, *c*, ..., *k* (*P=(a,b,c,...,k)*), se dice que *a*, que es el elemento más inaccesible de la pila, está en el fondo de la pila (*bottom*) y que k, por el contrario, el más accesible, está en la cima (*top*).

Las restricciones definidas para la pila implican que si una serie de *elementos A, B, C, D, E, F* se añaden, en este orden, a una pila entonces el primer elemento que se elimine (borre) de la estructura deberá ser *E*. Por tanto, resulta que el último elemento que se inserta en una pila es el primero que se borra. Por esa razón, se dice que una pila es una lista o estructura lineal de tipo LIFO (*Last In First Out*, el último que entra es el primero que sale).

Estructura Pila



Un ejemplo típico de pila lo constituye un montón de platos: Cuando se quiere introducir un nuevo plato, éste se coloca en la posición más accesible, encima del último plato. Cuando se coge un plato, éste se extrae, igualmente, del punto más accesible, el último que se ha introducido. O, si somos más estrictos, otro ejemplo sería una caja llena de libros. Sólo podemos ver cuál es el libro que está más arriba en la caja, y si ponemos o cojemos un libro, sólo podremos actuar sobre este primer libro. No podemos siquiera saber el número total de libros guardados en la pila. Sólo sabremos el número de elementos de la pila de libros si previamente los sacamos hasta vaciar la caja.

Otro ejemplo natural de la aplicación de la estructura pila aparece durante la ejecución de un programa de ordenador, en la forma en que la máquina procesa las llamadas a los procedimientos. Cada llamada a un procedimiento (o función) hace que el sistema almacene toda la información

asociada con ese procedimiento (parámetros, variables, constantes, direción de retorno, etc...) de forma independiente a otros procedimientos y permitiendo que unos procedimientos puedan invocar a otros distintos (o a si mismos) y que toda esa información almacenada pueda ser recuperada convenientemente cuando corresponda. Como en un procesador sólo se puede estar ejecutando un procedimiento, esto quiere decir que sólo es necesario que sean accesibles los datos de un procedimiento (el último activado, el que está en la cima). De ahí que una estructura muy apropiada para este fin sea la estructura pila.

Como ya vimos con las estructuras de ejemplo del tema anterior, asociadas con la estructura pila existen una serie de operaciones necesarias para su manipulación. Éstas son:

Iniciación de la estructura:

- Crear la pila (CrearPila): La operación de creación de la pila inicia la pila como vacía.

Operaciones para añadir y eliminar información:

- Añadir elementos en la cima (Apilar): pondrá un nuevo elemento en la parte superior de la pila.
- Eliminar elementos de la cima (Desapilar): lo que hará será devolver el elemento superior de la cima y eliminarlo de la pila.

Operaciones para comprobar tanto la información contenida en la pila, como el propio estado de la cima:

- Comprobar si la pila está vacía (PilaVacia): Esta operación es necesaria para poder decidir si es posible eliminar elementos de la pila.
- Acceder al elemento situado en la cima (CimaPila): Nos indica el valor del elemento situado en la parte superior de la pila.

La especificación correcta de todas estas operaciones permitirá definir adecuadamente una pila.

Una declaración más formal de las operaciones definidas sobre la estructura de datos pila, y los axiomas que las relacionan pordrían ser las siguientes:

Estructura

Pila (Valor) /* Valor *será el tipo de datos que podremos guardar en la pila* */

Operaciones

CREAR_PILA ( ) → Pila
APILAR (Pila , Valor) → Pila
DESAPILAR (Pila) → Pila
CIMA_PILA (Pila) → Valor
PILA_VACIA (Pila) → Lógico

Axiomas

∀ *stack* ∈ *Pila, x* ∈ *Valor se cumple que:*

PILA_VACIA (CREAR_PILA ( )) → cierto
PILA_VACIA (APILAR (stack, x)) → falso
DESAPILAR (CREAR_PILA ( )) → error
DESAPILAR (APILAR (stack, x)) → stack
CIMA_PILA (CREAR_PILA ( )) → error
CIMA_PILA (APILAR (stack, x)) → x

11.2. Representación de las Pilas en C++

Los lenguajes de programación de alto nivel no suelen disponer de un tipo de datos pila. Aunque, por el contrario, en lenguajes de bajo nivel (ensambladores) es posible manipular directamente

alguna estructura pila propia del sistema. Por lo tanto, en general, es necesario representar la estructura pila a partir de otros tipos de datos existentes en el lenguaje.

En C++, lo primero que nos plantearemos serán los métodos a través de los que podremos acceder a la información contenida en la estructura.

Las operaciones van a ser básicamente las que hemos definido en el tipo abstracto de datos, pero ahora habrá que decidir la implementación concreta de las mismas. Como criterio general estableceremos que aquellas funciones que puedan producir error, devolverán un booleano que marque si se ha producido o no error en la operación. Si esas funciones tuvieran que devolver algún valor, lo devolverán por referencia.

Así, **`Desapilar`** y **`CimaPila`** tendrán el siguiente prototipo:

```
bool Desapilar (Valor &);
bool CimaPila (Valor &);
```

suponiendo **`Valor`**, como el tipo de dato guardado en la Pila.

La consulta del estado de la pila (**`PilaVacia`**) será:

```
bool PilaVacia (void);
```

Recordar que en C++, si la implementación se realiza mediante clases, no es necesario pasar como parámetro el objeto de la consulta, ya que es el propio objeto el que hace la llamada al método que realiza la consulta.

Respecto de Apilar, su prototipo será similar al de Desapilar, aunque según los axiomas, en principio no va a devolver ningún error. En la implementación, la memoria del ordenador es finita, y tendremos en cuenta la posibilidad de que llegue un punto en el que no podamos apilar más elementos.

```
bool Apilar (Valor);
```

Finalmente, nos queda el método de iniciación de la estructura CrearPila. Este método debería ser llamado cada vez que declarasemos un objeto de tipo pila, ya la pila siempre debe empezar sin contener ningún valor.

En las declaraciones de clases en C++, esto se soluciona mediante la inclusión de un **constructor por defecto** en la clase.

El constructor por defecto es un método especial que es llamado siempre que se instancia (declara) un nuevo objeto de una clase.

Si no existe el constructor por defecto, la instanciación se limita a reservar el espacio necesario para guardar la información del objeto.

Si existe el constructor por defecto, éste es llamado después de la reserva de memoria, y se suele utilizar para iniciar las estructuras con valores 'correctos' (y no los valores aleatorios que en principio habría en la memoria.)

El constructor por defecto es un método que tiene el mismo nombre que la clase, no devuelve ningún valor (ni siquiera **`void`**) y tiene como parámetro **`void`**.

Con todo lo dicho, la declaración de la parte pública de la clase, quedaría como sigue:

```
class Pila
{
   public:
      Pila (void);                  // Constructor por defecto
      bool Apilar (Valor);
      bool Desapilar (void);
      bool CimaPila (Valor &);
      bool PilaVacia (void);
   private:
      .??.
};
```

Una vez llegados a este punto, ya podemos utilizar, como usuarios, la clase Pila. De hecho, ya en la parte del TAD podíamos plantearnos la utilización de la pilas mediante las operaciones descritas.

A continuación veremos algunos ejemplos de utilización de pilas, con una primera aproximación a la solución en pseudocódigo y una solución en C++.

Ejemplo de utilización de pilas: Evaluación de expresiones algebraicas

Sabemos que, para evitar ambigüedad y saber en qué orden se deben evalúar las expresiones, se define en los lenguajes de programación una prioridad para cada posible operador. Además, si dos operadores tienen la misma prioridad se evita el conflicto evaluando éstos de izquierda a derecha. También está permitido el uso de paréntesis para establecer un orden concreto de evaluación, independientemente de la precedencia de los operadores. Todos estos condicionamientos son debidos al tipo de notación empleada, la llamada notación infija, donde los operadores se sitúan entre los operandos sobre los que actúa. De manera que, según sabemos, la siguiente expresión:

$x \leftarrow A/B-C+D*E-A*C$

se evaluaría como sigue:

$x \leftarrow (((A/B) - C) + (D*E)) - (A*C)$

El problema es cómo genera el compilador el código necesario para calcular esta expresión. La solución se facilita si se modifica el tipo de notación empleada para representar la expresión. Es conveniente pasar a notación posfija (el operador se sitúa detrás de los operandos sobre los que actúa) durante el proceso de compilación. Las ventajas de la notación posfija son varias: no hace falta paréntesis; no hace falta definir prioridad de operadores; y la evaluación final de la expresión se realiza fácilmente con un simple recorrido de izquierda a derecha de la expresión.

Siguiendo la notación posfija, la expresión del ejemplo anterior se podría escribir como:

$x \leftarrow A\ B\ /\ C - D\ E\ *\ +\ A\ C\ *\ -$

Para realizar el cálculo de las expresiones en notación posfija, hay que tener en cuenta que al leerlas de izquierda a derecha, lo primero que se lee es la información (operandos) y después el tipo de acción (operador) que se realiza con ella. Por ello, es necesario almacenar la información leida hasta que se determine que operador hace uso de ella. Además, los operadores actúan sobre los últimos operandos leidos. De manera que, conviene recuperar la información en sentido inverso a como se almacena. Por esa razón, parece natural emplear una pila como estructura de almacenamiento de información.

El esquema algorítmico para la evaluación de expresiones dadas en notación posfija consistiría en ir analizando secuencialmente la expresión. Si se detecta un operando, se inserta en la pila y si se detecta un operador, éste se evalúa utilizando los operandos necesarios de la pila y situando, de nuevo, el resultado en la pila, puesto que será un operando para otro operador posterior. Este proceso es mucho más simple que intentar la evaluación directa a partir de la expresión en notación infija.

Algoritmo para evaluar expresiones:
Inicio
leer la expresión
Mientras haya elementos en la expresión **hacer**
Obtener el siguiente elemento de la expresión
Si$_{(1)}$ el elemento es un identificador de operando **entonces**
Apilar (elemento)
Sino$_{(1)}$ *{* es un operador *}*
op ← número de operandos del operador
Repetir op veces
Desapilar y Almacenar
Fin_repetir
aplicar el operador sobre los operandos desapilados
Apilar (resultado de la operación)
Fin_si$_{(1)}$
Fin_mientras
Si$_{(2)}$ *Pila_Vacia* **entonces**
"*Error. Demasiados operadores*"
Sino$_{(2)}$
Si$_{(3)}$ la Pila tiene más de un valor apilado **entonces**
"Error. Demasiados operandos"
Sino$_{(3)}$
Devolver (*Desapilar*)
Fin_si$_{(3)}$
Fin_si$_{(2)}$
Fin

Este algoritmo plasmado en código C++ sería:

```
int Evaluar (Expresion exp)
{
    Pila s;
    int res;
    Elemento token, e1_aux, e2_aux;

    while (!exp.ExpresionVacia() )
    {
        token = exp.SigElemento ();
        if (token.EsOperando() )
            s.Apilar (token);
        else
        {
            s.CimaPila (e1_aux);
            s.Desapilar ();
            s.CimaPila (e2_aux);
            s.Desapilar ();

            res = token.Operar (e1_aux, e2_aux);
            s.Apilar (res);
        }
```

```
        }

        if (s.PilaVacia () )
        {
            res = 0;
            cerr << "Error: Demasiados operadores";
        }
        else
        {
            s.CimaPila (res);
            s.Desapilar ();
            if (!s.PilaVacia() )
                cerr << "Error: Demasiados operandos";
        }

        return res;
    }
```

Ejemplo de utilización de pilas: Paso de una expresión en notación infija a notación postfija

Asociado con el problema de las expresiones, ya comentado, estaría el problema de cambiar de notación la expresión. Cómo pasar de la notación infija empleada durante la escritura del programa (que es cómoda y habitual para el usuario) a la notación posfija, más conveniente para automatizar los procesos de cálculo. Para realizar este proceso, de nuevo resulta adecuada la utilización de una pila.

Para el proceso de traducción de la expresión hay que tener en cuenta una serie de aspectos: los operandos aparecen en el mismo orden en la notación infija y en la posfija, con lo que no hay que realizar ningún tipo de acción específica cuando, al leer la expresión, se detecta un operando, simplemente proporcionarlo como salida; los operandos; por su parte, sí que cambian de posición al cambiar la notación. En la notación infija, el operador se sitúa antes que uno de los operandos, mientras que enla notación posfija siempre va detrás. Por esa razón, ahora es conveniente almacenar los operadores, no los operandos, hasta el momento en que halla que proporcionarlos como salida. Además, hay que tener en cuenta que la entrada de datos es una expresión en la que los operadores tienen asignadas disintas prioridades, estas prioridades también se deben tener en cuenta en el momento de la traducción.

El siguiente algoritmo permite traducir una expresión escrita en notación infija a notación posfija. Para simplificar la tarea, se ha supuesto que la expresión de entrada no tiene paréntesis (lo que complicaría ligeramente el proceso) y que tenemos un proceso paralelo que permite extraer cada uno de los elementos de la expresión (identificadores de variable, constantes, funciones, etc...):

Algoritmo Infija_A_Posfija (sin paréntesis)
Entrada
    expresion: cadena
Inicio
    Iniciar_Pila (stack)
    Mientras haya elementos en la expresión **hacer**
        Obtener el siguiente elemento de la expresión
        Si$_{(1)}$ (el elemento es un operando) **entonces**
            Añadimos el elemento a la expresión traducida
        Sino$_{(1)}$ *{* x es operador, entonces se apila pero... *}*
            fin ← **FALSO**
            { ...antes de apilar, analizar prioridades de operadores *}*

```
                    Mientras(2) ( no Pila_Vacia ( stack ) ) y ( no fin ) hacer
                        Si(2) (la prioridad del elemento es menor que la prioridad del
                              elemento que está en la cima de la pila) entonces
                            y ← Desapilar ( stack )
                            Añadimos 'y' a la expresión traducida
                        Sino(2)
                            fin ← CIERTO
                        Fin_si(2)
                    Fin_mientras(2)
                    Apilar ( stack, x )
                Fin_si(1)
            fin_mientras(1)
            {* se ha terminado la expresión, vaciar la pila *}
            Mientras(3) ( no Pila_Vacia ( stack ) ) hacer
                y ← Desapilar ( stack )
                Añadimos 'y' a la expresión traducida
            Fin_mientras(3)
    fin
```

Al igual que hicimos en el ejemplo anterior, el algoritmo escrito en código C++:

```
Expresion Infijo2Posfijo (Expresion e)
{
    Pila s;
    Expresion traduccion;
    Elemento token, x;
    bool fin;

    while (!e.ExpresionVacia () )
    {
        token = e.SigElemento ();
        if (token.EsOperando () )
            traduccion.Concatenar (token);
        else
        {
            fin = false;
            while ( (!s.PilaVacia () ) && (!fin) )
            {
                s,CimaPila (x);
                if (token.Prioridad () <= x.Prioridad () )
                {
                    s.Desapilar ();
                    traduccion.Concatenar (x);
                }
                else
                    fin = true;
            }
            s.Apilar (token);
        }
    }
    while (!s.PilaVacia () )
    {
        s.CimaPila (x);
        s.Desapilar ();
        traduccion.Concatenar (x);
    }

    return traduccion;
}
```

Implementación mediante estructuras estáticas

La forma más simple, y habitual, de representar una pila es mediante un vector unidimensional. Este tipo de datos permite definir una secuencia de elementos (de cualquier tipo) y posee un eficiente mecanismo de acceso a la información contenida en ella.

Al definir un *array* hay que determinar el número de índices válidos y, por lo tanto, el número de componentes definidos. Entonces, la estructura pila representada por un array tendrá limitado el número de posibles elementos.

La parte privada de la clase, será pues un vector donde guardaremos la información.

El primer elemento de la pila se almacenará en **`info[0]`**, será el fondo de la pila, el segundo elemento en **`info[1]`** y así sucesivamente. En general, el elemento *i*-ésimo estará almacenado en **`info[i - 1]`**.

Como todas las operaciones se realizan sobre la cima de la pila, es necesario tener correctamente localizada en todo instante esta posición. Es necesaria una variable adicional, **`cima`**, que apunte al último elemento de la pila o nos diga cuantos elementos tenemos en ella.

Resumiendo, la clase **`Pila`** contendrá, en esta implementación, la siguiente parte privada:

```
class Pila
{
   public: ...

   private:
      Vector vect;
      int cima;
};
```

Donde **`Vector`** será:

```
typedef Valor Vector[MAX];
```

Suponiendo **`Valor`**, el tipo de dato que se puede almacenar en la pila, y **`MAX`** una constante que me limita el tamaño máximo de la pila.

Con estas consideraciones prácticas, se puede pasar a definir las operaciones que definen la pila.

Operación CREAR_PILA

La creación de la pila se realizará mediante el constructor por defecto. La tarea que deberá realizar será decir que no existen elementos en la pila:

```
stack: Pila

stack.Cima ← 0
```

En C++:

```
Pila::Pila (void)
{
   cima = 0;
}
```

Operación PILA_VACIA

Esta operación permitirá determinar si es posible eliminar elementos.

La pila estará vacia si la cima está apuntando al valor cero.

```
Algoritmo Pila_Vacia
Entrada
        stack: Pila
Salida
        (CIERTO, FALSO)
Inicio
        Si ( stack.Cima = 0 ) entonces
                Devolver ( CIERTO )
        Sino
                Devolver ( FALSO )
        Fin_si
Fin
```

```
bool Pila::PilaVacia (void)
{
   return cima == 0;
}
```

Operación de inserción de información (APILAR)

La operación de inserción normalmente se conoce por su nombre inglés *Push*, o Apilar. La operación aplicada sobre un pila y un valor x, inserta x en la cima de la pila. Esta operación está restringida por el tipo de representación escogido. En este caso, la utilización de un array implica que se tiene un número máximo de posibles elementos en la pila, por lo tanto, es necesario comprobar, previamente a la inserción, que realmente hay espacio en la estructura para almacenar un nuevo elemento. Con está consideración, el algoritmo de inserción sería:

```
Algoritmo Apilar
Entradas
        x: Valor                                    {* elemento que se desea insertar *}
        stack: Pila de Valor
Salidas
        stack
Inicio
        {* comprobar si en la pila se pueden insertar más elementos          *}
        {* esto es necesario por el tipo de representación de la estructura  *}
        Si ( stack.Cima = MAX ) entonces
                Error "pila llena"
        Sino
                stack.Cima ← stack.Cima + 1
                stack.Info [stack.Cima] ← x
        Fin_sino
Fin
```

En la implementación en C++, tendremos en cuenta que vamos a devolver mediante la función si se ha producido algún tipo de error o no, en vez de mostrar un mensaje por pantalla.

```
bool Pila::Apilar (Valor x)
{
   bool error;

   if (cima == MAX)
      error = true;
   else
   {
      error = false;
      info[cima] = x;
      cima++;
   }
   return error;
}
```

*

Operación de consulta de información (CIMA_PILA)

La operación de consulta de la información, sólo puede acceder al elemento que esté situado en la cima de la pila, y proporcionar su valor. El algoritmo se limitará a devolver el elemento que está situado en la posición cima de la pila, si existe información almacenada en la pila.

```
Algoritmo Cima_Pila
Entradas
        stack: Pila de Valor
Salidas
        Valor
Inicio
        {* comprobar si existe información en la pila                    *}
        {* esta operación no depende de la representación, siempre es necesaria  *}
        Si ( Pila_Vacia ( stack ) ) entonces
                Error "pila vacia"
        sino
                Devolver ( stack.Info [stack.Cima] )
        Fin_si
Fin
```

*

```
bool Pila::CimaPila (Valor & x)
{
   bool error;

   if (cima == 0)
      error = true;
   else
   {
      error = false;
      x = info[cima - 1];
   }
   return error;
}
```

*

Operación de eliminación de información (DESAPILAR)

La operación de borrado elimina de la estructura el elemento situado en la cima. Normalmente recibe el nombre de *Pop* en la bibliografía inglesa. El algoritmo de borrado sería:

```
Algoritmo Desapilar
Entradas
        stack: Pila de Valor
Salidas
        stack
        x: Valor
Inicio
        {* comprobar si se pueden eliminar elementos de la pila          *}
        {* esta operación no depende de la representación, siempre es necesaria  *}
        Si ( Pila_Vacia ( stack ) ) entonces
                Error "pila vacia"
        sino
                stack.Cima ← stack.Cima - 1
        Fin_si
Fin
```

```
bool Pila::Desapilar (void)
{
   bool error;

   if (cima == 0)
      error = true;
   else
   {
      error = false;
      cima--;
   }
   return error;
}
```

Implementación mediante cursores

Implementación mediante estructuras dinámicas

Uno de los mayores problemas en la utilización de estructuras estáticas, estriba en el hecho de tener que determinar, en el momento de la realización del programa, el valor máximo de elementos que va a poder contener la estructura.

Una posible solución a este problema es la utilización de estructuras dinámicas enlazadas (utilización de punteros) tal y como se explicó en el primer cuatrimestre.

Por tanto, la clase **`Pila`** contendrá, en esta implementación, la siguiente parte privada:

```
class Pila
{
   public: ...

   private:
      Puntero cima;
};
```

Donde **`Puntero`** será:

```
typedef struct Nodo * Puntero;
```

Y el tipo **Nodo** será:

```
struct Nodo
{
   Valor info;
   Puntero sig;
};
```

Suponiendo **Valor**, el tipo de dato que se puede almacenar en la pila.

La implementación en este caso de los métodos de la clase Pila será la siguiente.

Operación CREAR_PILA

---

stack: Pila

stack.Cima ← NULL

En C++:

```
Pila::Pila (void)
{
   cima = NULL;
}
```

Operación PILA_VACIA

Esta operación permitirá determinar si es posible eliminar elementos.

La pila estará vacia si la cima está apuntando al valor cero.

Algoritmo Pila_Vacia
Entrada
stack: Pila
Salida
(**CIERTO**, **FALSO**)
Inicio
Si (stack.Cima = NULL) **entonces**
Devolver (**CIERTO**)
Sino
Devolver (**FALSO**)
Fin_si
Fin

```
bool Pila::PilaVacia (void)
{
   return cima == NULL;
}
```

Operación de inserción de información (APILAR)

Con la representación enlazada de la pila, la estructura tiene una menor limitación en cuanto al posible número de elementos que se pueden almacenar simúltaneamente. Hay que tener en cuenta que la representación de la pila ya no requiere la especificación de un tamaño máximo, por lo que mientras exista espacio libre en memoria se va a poder reservar espacio para nuevos elementos. Por esa razón, se va suponer en el siguiente algoritmo que la condición de pila llena no se va a dar y, por lo tanto, no será necesaria su comprobación.

Algoritmo Apilar
Entrada
stack: Pila de Valores
x: Valor
Salida
stack
Variable
p_aux: Puntero_a_Nodo_pila
Inicio
p_aux ← **Crear_Espacio**
p_aux^.Info ← x
p_aux^.Sig ← stack.Cima
stack.Cima ← p_aux
Fin

Para hacer compatible este método con el método definido para el caso estático, devolveremos siempre '**FALSE**' (no se ha producido ningún error).

```
bool Pila::Apilar (Valor x)
{
   bool error;
   Puntero p_aux;

   error = false;

   p_aux = new Nodo;
   p_aux->info = x;
   p_aux->sig = cima;
   cima = p_aux;

   return error;
}
```

Operación de consulta de información (CIMA_PILA)

Al elemento "visible" de la pila se puede acceder fácilmente a través del puntero que le referencia, *cima*, que siempre debe existir y ser adecuadamente actualizado.

```
Algoritmo Cima_Pila
Entradas
        stack: Pila de Valor
Salidas
        Valor
Inicio
        {* comprobar si existe información en la pila                    *}
        {* esta operación no depende de la representación, siempre es necesaria  *}
        Si ( Pila_Vacia ( stack ) ) entonces
                Error "pila vacia"
        sino
                Devolver (Cima^.Info)
        Fin_si
Fin
```

```
bool Pila::CimaPila (Valor & x)
{
   bool error;

   if (cima == NULL)
      error = true;
   else
   {
      error = false;
      x = cima->info;
   }
   return error;
}
```

Operación de eliminación de información (DESAPILAR)

La único que hay que tener en cuenta a la hora de diseñar un algoritmo para esta operación es la utilización eficiente de la memoria, de forma que el espacio ocupado por el nodo borrado vuelva a estar disponible para el sistema. Recordar que si la pila está vacía no se puede desapilar.

```
Algoritmo Desapilar
Entrada
        stack: Pila de Valor
Salida
        stack
        x: Valor
Variable
        p_aux: Puntero_a_Nodo_pila
Inicio
        Si (Pila_Vacia (stack) ) entonces
                Error "pila_vacia"
        Sino
                p_aux ← stack.Cima
                stack.Cima ← p_aux^.Sig

                Liberar_Espacio (p_aux)
        Fin_si
Fin
```

En C++, el método quedaría como sigue:

```
bool Pila::Desapilar (void)
{
   bool error;
   Puntero p_aux;

   if (cima == NULL)
      error = true;
   else
   {
      error = false;

      p_aux = cima;
      cima = cima->sig;
      delete p_aux;
   }
   return error;
}
```