INTRODUCCIÓN

En este capítulo se estudian en detalle la estructura de datos *Pila* utilizada frecuentemente en la resolución de algoritmos. La *Pila* es una estructura de datos que almacena y recupera sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, First-Out, último en entrar primero en salir*). El desarrollo de las pilas como tipos abstractos de datos es el motivo central de este capítulo.

Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones. Una aplicación interesante es la evaluación de expresiones algebraicas mediante pilas.

11.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por su parte superior, la **cima** de la pila. Éste es el caso de una pila de platos, una pila de libros, etc.

Definición

Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado cima.

Cuando se dice que la pila está ordenada, se quiere decir que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador "*menor que*" y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

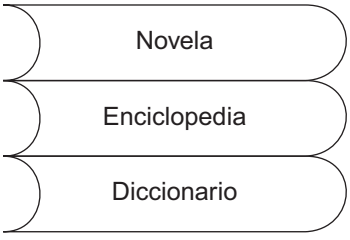


Figura 11.1. Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica "*último en entrar, primero en salir*" se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*).

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **`Insertar`** (`push`) añade un elemento en la cima de la pila y la operación **`Quitar`** (pop) elimina o saca un elemento de la pila. La Figura 11.2 muestra una secuencia de operaciones *Insertar* y *Quitar*.

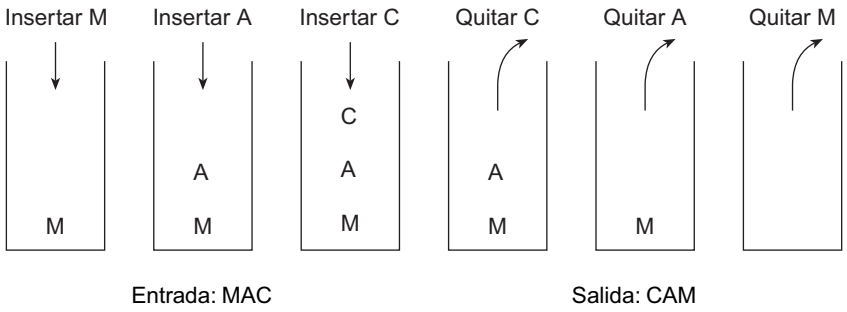


Figura 11.2. Insertar y quitar elementos de la pila.

La operación `insertar` (*push*) sitúa un elemento en la cima de la pila y `quitar` (*pop*) elimina o extrae el elemento cima de la pila.

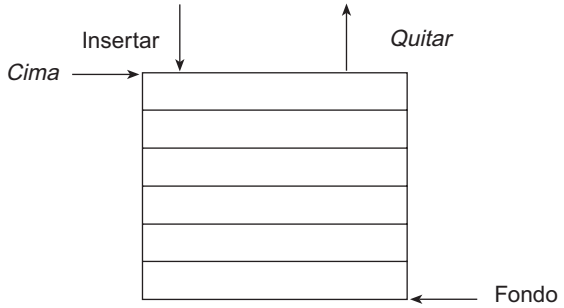


Figura 11.3. Operaciones básicas de una pila.

La pila se puede implementar guardando los elementos en un array, en cuyo caso su dimensión o longitud es fija. Otra forma de implementación consiste en construir una lista enlazada, cada elemento de la pila forma un nodo de la lista; la lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica y no existe limitación en su tamaño excepto la memoria del ordenador.

Una pila puede estar *vacía* o *llena* (en la representación con un array, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una *excepción*, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila *llena* se produce un error, una *excepción*, de **desbordamiento** (*overflow*) o *rebosamiento*. Para evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

11.1.1. Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes.

Tipo de dato	Dato que se almacena en la pila
Operaciones	
CrearPila	Inicia la pila.
Insertar (push)	Pone un dato en la pila.
Quitar (pop)	Retira (saca) un dato de la pila.
Pilavacía	Comprobar si la pila no tiene elementos.
Pilallena	Comprobar si la pila está llena de elementos.
Limpiar pila	Quita todos sus elementos y dejar la pila vacía.
CimaPila	Obtiene el elemento cima de la pila.
Tamaño de la pila	Número de elementos máximo que puede contener la pila.

La operación *Pilallena* sólo se implementa cuando se utiliza un array para almacenar los elementos. Una pila puede crecer indefinidamente si se implementa con una estructura dinámica.

11.2. TIPO DE DATO PILA IMPLEMENTADO CON ARRAYS

La implementación con un array es *estática* porque el array es de tamaño fijo. La clase *Pila*, con esta representación, además del array, utiliza la variable `cima` para apuntar (índice) al último elemento colocado en la pila. Es necesario controlar el tamaño de la pila para que no exceda al número de elementos del array, y la condición *Pilallena* será significativa para el diseño.

El método usual de introducir elementos en la pila es definir el *fondo* en la posición `0` del array y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array de modo que el primer elemento se introduce en una pila vacía y en la posición `0`, el segundo elemento en la posición `1`, el siguiente en la posición `2` y así sucesivamente. Con estas operaciones el índice que apunta a la cima de la pila va incrementando en `1` cada vez que se añade un nuevo elemento. Los algoritmos de insertar (*push*) y quitar (*pop*) datos de la pila son::

Insertar (*push*)

```
1. Verificar si la pila no está llena.
2. Incrementar en 1 el apuntador (cima) de la pila.
3. Almacenar el elemento en la posición del apuntador de la pila.
```

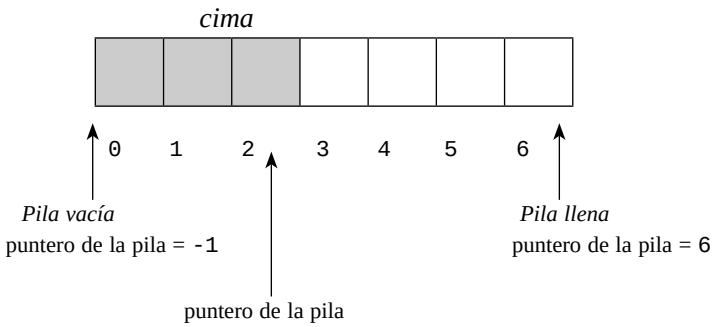
Quitar (*pop*)

```
1. Si la pila no está vacía.
2. Leer el elemento de la posición del apuntador de la pila.
3. Decrementar en 1 el apuntador de la pila.
```

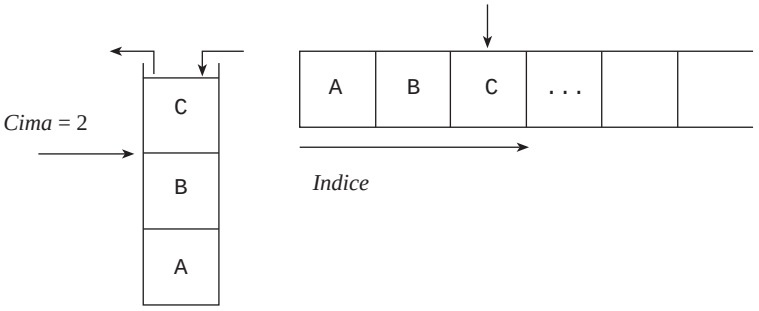
El rango de elementos que puede tener una pila varía de `0` a `TAMPILA-1`, en el supuesto de que el array se defina de tamaño `TAMPILA` elementos. De modo que *en una pila llena* el

apuntador (índice del array) de la pila tiene el valor `TAMPILA-1`, y *en una pila vacía* tendrá `-1` (el valor `0` es el índice del primer elemento).

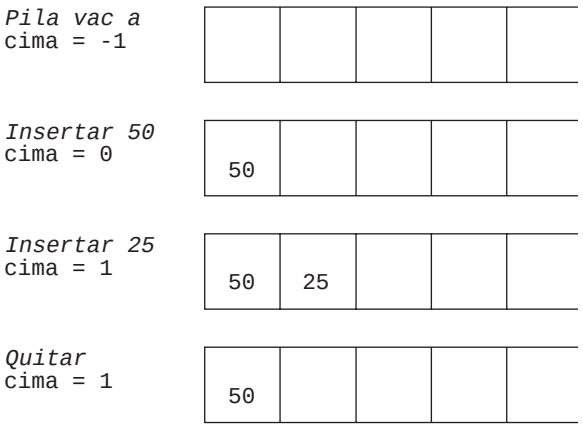
EJEMPLO 11.1. Una pila de 7 elementos se puede representar g ráficamente así:



Si se almacenan los datos `A, B, C`, ... en la pila se puede representar gráficamente de alguna de estas formas



A continuación, se muestra la imagen de una pila según diferentes operaciones realizadas.



11.2.1. Especificación de la clase `Pila`

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las operaciones. En el *TAD* `Pila` los datos pueden ser de cualquier tipo y las operaciones las ya citadas en 11.1.1.

1. Datos de la pila (`TipoDato` es cualquier tipo de dato primitivo o tipo clase).
2. `crearPila` inicializa una pila. Crear una pila sin elementos, por tanto, vacía.
3. Verificar que la pila no está llena antes de `insertar` o poner ("*push*") un elemento en la pila; verificar que una pila no está vacía antes de `quitar` o sacar ("*pop*") un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error (una *excepción*) y el programa debe terminar.
4. `pilaVacia` devuelve *verdadero* si la pila está vacía y *falso* en caso contrario.
5. `pilaLlena` devuelve *verdadero* si la pila está llena y *falso* en caso contrario. Estas dos últimas operaciones se utilizan para verificar las precondiciones de *insertar* y *quitar*.
6. `limpiarPila` vacía la pila, dejándola sin elementos y disponible para otras tareas.
7. `cimaPila`, devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que la pila queda intacta.

Declaración de la clase `Pila`

```
typedef tipo TipoDeDato;  // tipo de los elementos de la pila
// archivo de cabecera pilalineal.h
const int TAMPILA = 49;
class PilaLineal
{
  private:
    int cima;
    TipoDeDato listaPila[TAMPILA];
  public:
    PilaLineal()
    {
      cima = -1;  // condición de pila vacía
    }
    // operaciones que modifican la pila
    void insertar(TipoDeDato elemento);
    TipoDeDato quitar();
    void limpiarPila();
    // operación de acceso a la pila
    TipoDeDato cimaPila();
    // verificación estado de la pila
    bool pilaVacia();
    bool pilaLlena();
};
```

La declaración realizada está ligada al tipo de los elementos de la pila. Para alcanzar la máxima abstracción, se declara la clase genérica *PilaLineal* de tal forma que el tipo de dato de los elementos se especifica al crear el objeto pila.

EJEMPLO 11.2. Escribir un programa que cree una pila de enteros. Se realicen operaciones de añadir datos a la pila, quitar ...

Se supone implementada la clase pila con el tipo primitivo `int`. El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave `-1`), a continuación, extrae los elementos de la pila hasta que se vacía. En pantalla deben escribirse los números leídos en orden inverso por la naturaleza de la pila. El bloque de sentencias se encierra en un bloque `try` para tratar errores de desbordamiento de la pila.

```
#include <iostream>
using namespace std;
typedef int TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;      // crea pila vacía
  TipoDeDato x;
  const TipoDeDato CLAVE = -1;

  cout << "Teclea elemento de la pila(termina con -1)" << endl;
  try {
    do {
      cin >> x;
      pila.insertar(x);
    }while (x != CLAVE);

    // proceso de la pila
    cout << "Elementos de la Pila: " ;
    while (!pila.pilaVacia())
    {
      x = pila.quitar();
      cout << x << " ";
    }
  }
  catch (const char * error)
  {
    cout << "Excepción: " << error;
  }
  return 0;
}
```

11.2.2. Implementación de las operaciones sobre pilas

Las funciones de la clase `Pila` son sencillas de implementar, teniendo en cuenta la característica principal de esta estructura: *inserciones y borrados se realizan por el mismo extremo, la cima de la pila*.

La operación de `insertar` un elemento en la pila, incrementa el apuntador `cima` y asigna el nuevo elemento a la lista. Cualquier intento de añadir un elemento en una pila llena genera una excepción o error debido al "*Desbordamiento de la pila*".

```
void PilaLineal::insertar(TipoDeDato elemento)
{
```

```
  if (pilaLlena())
  {
    throw "Desbordamiento pila";
  }
  //incrementar puntero cima y copia elemento
  cima++;
  listaPila[cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila copiando, en primer lugar, el valor de la cima de la pila en una variable local, `aux`, y a continuación, decrementa el puntero de la pila. `quitar()` devuelve la variable `aux`, es decir, el elemento eliminado por la operación. Si se intenta eliminar o borrar un elemento en una pila vacía se produce error, se lanza una excepción.

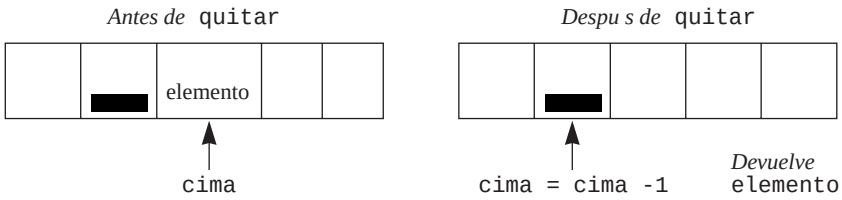


Figura 11. 4. Extraer elemento cima.

```
TipoDeDato PilaLineal::quitar()
{
  TipoDeDato aux;
  if (pilaVacia())
  {
    throw "Pila vacía, no se puede extraer.";
  }
    // guarda elemento de la cima
    aux = listaPila[cima];
    // decrementar cima y devolver elemento
    cima--;
    return aux;
}
```

La operación `cimaPila` devuelve el elemento que se encuentra en la cima de la pila, no se modifica la pila, únicamente accede al elemento.

```
TipoDeDato PilaLineal::cimaPila()
{
  if (pilaVacia())
  {
    throw "Pila vacía, no hay elementos.";
  }
  return listaPila[cima];
}
```

11.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben el estado de la pila: *pila vacía* o *pila llena*. Asimismo, se ha

de definir una operación, `LimpiarPila`, que restaure la condición inicial de la pila, cima igual a -1.

La función `pilaVacia` comprueba si la cima de la pila es -1. En cuyo caso la pila está vacía y devuelve *verdadero*.

```
bool PilaLineal::pilaVacia()
{
  return cima == -1;
}
```

La función `pilaLlena` comprueba si la cima es `TAMPILA-1`; en cuyo caso la pila está llena y devuelve *verdadero*.

```
bool PilaLineal::pilaLlena()
{
  return cima == TAMPILA - 1;
}
```

Por último, `limpiarPila()` pone la cima de la pila a su valor inicial.

```
void PilaLineal::limpiarPila()
{
  cima = -1;
}
```

EJERCICIO 11.1. Escribir un programa que utilice una `Pila` para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. Nota: una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual v alor: **alila**, es un palíndromo; **cara (arac)** no es un palíndromo.

La palabra se lee con la función `gets()` y se almacena en un `string`; cada carácter de la palabra se pone en una pila de caracteres. Una vez leída la palabra y construida la pila, se compara el primer carácter del `string` con el carácter que se extrae de la pila, si son iguales sigue la comparación con el siguiente carácter del `string` y de la pila; así sucesivamente hasta que la pila se queda vacía o hay un carácter no coincidente.

Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones de caracteres se realizan en orden inverso: primero con último … .

No es necesario volver a implementar las operaciones de la clases `Pila`, simplemente se cambia el tipo de dato de los elementos, en esta ocasión `char`.

```
#include <iostream>
using namespace std;
#include <string.h>
typedef char TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pilaChar;   // crea pila vacía
  TipoDeDato ch;
  bool esPal;
  char pal[81];
```

```
  cout << "Teclea la palabra verificar si es palíndromo: ";
  gets(pal);
  for (int i = 0; i < strlen(pal); )
  {
    char c;
    c = pal[i++];
    pilaChar.insertar(c);
  }
  // se comprueba si es palíndromo

  esPal = true;
  for (int j = 0; esPal && !pilaChar.pilaVacia(); )
  {
    char c;
    c = pilaChar.quitar();
    esPal = pal[j++] == c;
  }
  pilaChar.limpiarPila();
  if (esPal)
    cout << "La palabra " << pal << " es un palíndromo \n";
  else
    cout << "La palabra " << pal << " no es un palíndromo \n";
  return 0;
}
```

EJEMPLO 11.3. Llenar una pila con números leídos del teclado. A continuación vaciar la pila de tal forma que se muestren los valores positivos.

En este ejemplo el tipo de los elementos de la pila es `double`. El número de elementos que tendrá la pila se solicita al usuario; en un bucle *for* se lee el elemento y se inserta en la pila. Para vaciar la pila se diseña un bucle, *hasta pila vacía*; cada elemento que se extrae se escribe si es positivo.

```
#include <iostream>
using namespace std;
typedef double TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;
  int x;

  cout << "Teclea número de elementos: ";
  cin >> x;
  for (int j = 1; j <= x; j++)
  {
    double d;
    cin >> d;
    pila.insertar(d);
  }
  // vaciado de la pila
```

```
  cout << "Elementos de la Pila: ";
  while (!pila.pilaVacia())
  {
    double d;
    d = pila.quitar();
    if (d > 0.0)
      cout << d << " ";
  }
  return 0;
}
```

11.3. PILA GENÉRICA CON LISTAS ENLAZADA

La realización dinámica de una pila utilizando una lista enlazada almacena cada elemento de la pila como un nodo de la lista. Como las operaciones de *insertar* y *extraer* en el *TAD Pila* se realizan por el mismo extremo (cima de la pila), las acciones correspondientes con la lista se realizarán siempre por el mismo extremo de la lista.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente al número de elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria para guardar el campo de enlace entre nodos consecutivos. La Figura 11.5 muestra la imagen de una pila implementada con una lista enlazada.

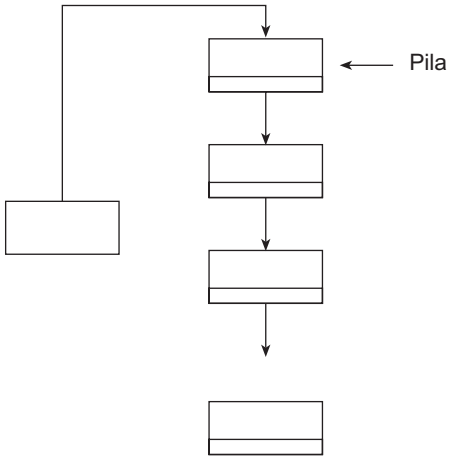


Figura 11.5. Representación de una pila con una lista enlazada.

Nota

Una pila realizada con una lista enlazada crece y decrece dinámicamente . En tiempo de ejecución, se reserva memoria según se ponen elementos en la pila y se libera memoria según se extraen elementos de la pila.

11.3.1. Clase *PilaGenerica* y *NodoPila*

La estructura que tiene la pila implementada con una lista enlaza es muy similar a la expuesta en listas enlazadas. Los elementos de la pila son los nodos de la lista, con un atributo para guardar el elemento y otro de enlace. Las operaciones del tipo pila implementada con listas son, naturalmente, las mismas que si la pila se implementa con arrays, salvo la operación que controla si la pila está llena, *pilaLlena*, que ahora no tiene significado ya que la listas enlazadas crecen indefinidamente, con el único límite de la memoria.

El tipo de dato de elemento se corresponde con el tipo de los elementos de la `Pila`, para que no dependa de un tipo concreto; para que sea genérico, se diseña una *pila genérica* utilizando las *plantillas* (`template`) de C++. La clase `NodoPila` representa un nodo de la lista enlazada, tiene dos atributos: elemento, guarda el elemento de la pila y siguiente, contiene la dirección del siguiente nodo de la lista. En esta implementación, `NodoPila` es una clase interna de `PilaGenerica`.

```
// archivo PilaGenerica.h

template <class T>
class PilaGenerica
{
  class NodoPila
  {

    public:
      NodoPila* siguiente;
      T elemento;
      NodoPila(T x)
      {
        elemento = x;
        siguiente = NULL;
      }
  };
  NodoPila* cima;

  public:
    PilaGenerica ()
    {
      cima = NULL;
    }
    void insertar(T elemento);
    T quitar();
    T cimaPila(); const
    bool pilaVacia(); const
    void limpiarPila();
    ~PilaGenerica()
    {
      limpiarPila();
    }
};
```

11.3.2. Implementación de las operaciones del *TAD Pila* con listas enlazadas

El constructor de `Pila` inicializa a ésta como pila vacía `(cima == NULL)`, realmente, a la condición de *lista vacía*. Las operaciones `insertar`, `quitar` y `cimaPila` acceden a la lista directamente con el puntero cima (apunta al último nodo apilado). Entonces, como no necesitan recorren los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, `O(1)`.

La codificación que a continuación se escribe, es para una pila de elemento de cualquier tipo. Es preciso recordar que al crear una instancia de pila es cuando se informa del tipo concreto de sus elementos, por ejemplo `PilaGenerica<char> pila`.

Verificación del estado de la pila

```
template <class T>
bool PilaGenerica<T>::pilaVacia() const
{
   return cima == NULL;
}
```

Poner un elemento en la pila

Crea un nuevo nodo con el elemento que se pone en la pila y se enlaza por la cima.

```
template <class T>
void PilaGenerica<T>::insertar(T elemento)
{
  NodoPila* nuevo;
  nuevo = new NodoPila(elemento);
  nuevo -> siguiente = cima;
  cima = nuevo;
}
```

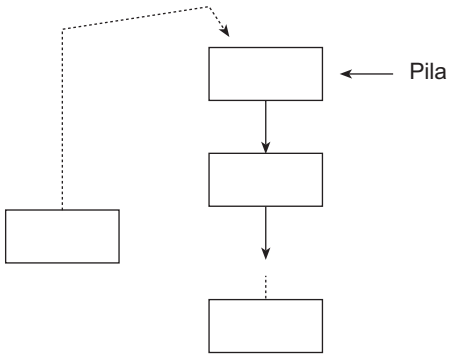


Figura 11.6. Apilar un elemento.

Eliminación del elemento cima

Retorna el elemento cima y lo quita de la pila, disminuye el tamaño de la pila.

```
template <class T>
T PilaGenerica<T>::quitar()
{
  if (pilaVacia())
    throw "Pila vacía, no se puede extraer.";
  T aux = cima -> elemento;
  cima = cima -> siguiente;
  return aux;
}
```

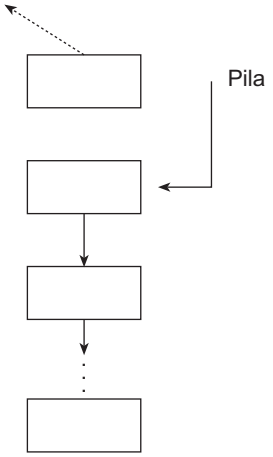


Figura 11.7. Quita la cima de la pila.

Obtención del elemento cima de la pila

```
template <class T>
T PilaGenerica<T>:: cimaPila(); const
{
  if (pilaVacia())
    throw "Pila vacía";
  return cima -> elemento;
}
```

Vaciado de la pila

Libera todos los nodos de que consta la pila. Recorre los `n` nodos de la lista enlazada, es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void PilaGenerica<T>:: limpiarPila()
```

```
{
  NodoPila* n;
  while(!pilaVacia())
  {
    n = cima;
    cima = cima -> siguiente;
    delete n;
  }
}
```

11.4. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS CON PILAS

Una *expresión aritmética* está formada por operandos y operadores. La expresión `x * y - (a + b)` consta de los operadores `*, -, +` y de los operandos `x, y, a, b`. Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. Los operadores son los símbolos conocidos de las operaciones matemáticas.

La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. Si los operandos de la expresión anterior toman los valores: `x = 5, y = 2, a = 3, b = 4` el resultado de la evaluación es:

```
5 * 2 – (3 + 4) = 5 * 2 – 7 = 10 – 7 = 3
```

La forma habitual de escribir expresiones matemáticas sitúa el operador entre sus dos operandos. La expresión anterior está escrita de esa forma, recibe el nombre de *notación infija*. Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para *encerrar* subexpresiones con mayor prioridad.

Los operadores, como es sabido, tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. A continuación, se recuerda estos niveles de prioridad en orden de mayor a menor:

```
Paréntesis              : ( )
Potencia                : ^
Multiplicación/división : *, /
Suma/Resta              : +, -
```

Normalmente, en una expresión hay operadores con la misma prioridad, a igualdad de precedencia, los operadores se evalúan de izquierda a derecha *(asociatividad)*, excepto la potencia que es de derecha a izquierda.

11.4.1. Notación *prefija* y notación *postfija* de una expresiones aritmética

Las operaciones aritméticas escritas en *notación infija* en muchas ocasiones necesitan usar paréntesis para indicar el orden de evaluación. Las expresiones

```
r = a * b / (a + c)          g = a * b / a + c
```

son distintas al no poner paréntesis en la expresión `g`. Igual ocurre con estas otras:

```
r = (a - b) ^ c + d          g = a - b ^ c + d
```

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. La notación en la que el operador se coloca delante de los dos operandos, *notación prefija*, se conoce también como *notación polaca* por el matemático polaco que la propuso. En el Ejemplo 11.4 se escriben expresiones en notación *prefija* o notación polaca.

EJEMPLO 11.4. Dadas las expresiones: `a * b / (a + c); a * b / a + c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación prefija.

Paso a paso, se escribe la transformación de cada expresión algebraica en la expresión equivalente en notación polaca.

```
a * b / (a + c) (infija) → a * b / + ac → * ab / + ac → / *ab + ac (polaca)
a * b / a + c (infija) → * ab / a + c → / * aba + c → + / *abac (polaca)
(a - b) ^ c + d (infija) → -ab ^ c + d → ^ -abc + d → +^-abcd (polaca)
```

Nota

La propiedad fundamental de la notación polaca es que el orden de ejecución de las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión. No son necesarios los paréntesis al escribir la expresión en notación polaca, como se observa en el Ejemplo 11.4.

Notación postfija

Hay más formas de escribir las expresiones. La notación *postfija* o *polaca inversa* coloca el operador a continuación de sus dos operandos.

EJEMPLO 11.5. Dadas las expresiones: `a*b/(a+c); a*b/a+c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación postfija.

Paso a paso se transforma cada subexpresión en notación polaca inversa.

```
a*b/(a+c) (infija) → a*b/ac+ → ab*/ac+ → ab*ac+/    (polaca inversa)
a*b/a+c (infija) → ab*/a+c → ab*a/+c → ab*a/c+      (polaca inversa)
(a-b)^c+d (infija) → ab-^c+d → ab-c^+d → ab-c^d+    (polaca inversa)
```

Recordar

Las diferentes formas de escribir una misma expresión algebraica dependen de la ubicación de los operadores respecto a los operandos. Es importante tener en cuenta que tanto en la notación prefija como en la postfija no son necesarios los paréntesis para cambiar el orden de evaluación.

11.4.2. Evaluación de una expresión aritmética

La evaluación de una expresión aritmética escrita de *manera habitual*, en *notación infija*, se realiza en dos pasos principales:

1.º Transformar la expresión de notación infija a postfija.
2.º Evaluar la expresión en notación postfija.

El *TAD Pila* es fundamental en los algoritmos que se aplican a cada uno de los pasos. El orden que fija la estructura pila asegura que el *último en entrar es el primero en salir*, y de esa forma el algoritmo de transformación a *postfija* sitúa los operadores después de sus operandos, con la prioridad o precedencia que le corresponde. Una vez que se tiene la expresión en notación *postfija*, se utiliza otra pila, de elementos numéricos, para guardar los valores de los operandos, y de las operaciones parciales con el fin de obtener el valor numérico de la expresión.

11.4.3. Transformación de una expresión infija a *postfija*

Se parte de una expresión en *notación infija* que tiene operandos, operadores y puede tener paréntesis. Los operandos se representan con letras, los operadores son éstos:

```
^ (potenciación), *, /, +, - .
```

La transformación se realiza utilizando una pila para guardar operadores y los paréntesis izquierdos. La expresión aritmética se lee del teclado y se procesa carácter a carácter. Los operandos pasan directamente a formar parte de la expresión en *postfija* la cual se guarda en un array. Un operador se mete en la pila si se cumple que:

- La pila esta vacía, o,
- El operador tiene mayor prioridad que el operador cima de la pila, o bien,
- El operador tiene igual prioridad que el operador cima de la pila y se trata de la máxima prioridad.

Si la prioridad es menor o igual que la de *cima pila*, se saca el elemento cima de la pila, se pone en la expresión en *postfija* y se vuelve a hacer la comparación con el nuevo elemento cima.

El paréntesis izquierdo siempre se mete en la pila; ya en la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila. Cuando se lee un paréntesis derecho se sacan todos los operadores de la pila y pasan a la expresión *postfija*, hasta llegar a un paréntesis izquierdo que se elimina ya que los paréntesis no forman parte de la expresión *postfija*. El algoritmo termina cuando no hay más items de la expresión origen y la pila está vacía.

Por ejemplo, dada la expresión `a*(b+c-(d/e^f)-g)-h` escrita en *notación infija*, a continuación, se va a ir formando, paso a paso, la expresión equivalente en postfija.

Expresión en postfija

`a`	Operando `a` pasa a la expresión *postfija*; operador `*` a la pila.
`ab`	Operador `(` pasa a la pila; operando `b` a la expresión.

`abc` Operador + pasa a la pila; operando a la expresión.

En este punto, el estado de la pila:

El siguiente carácter de la expresión, `-`, tiene igual prioridad que el operador de la cima (+), da lugar:

`abc+`
`abc+d` El operador `(`se mete en la pila; el operando `d` a la expresión.
`abc+de` El operador `/` pasa a la pila; el operando e a la expresión.
`abc+def` El operador `^` pasa a la pila; el operando `f` a la expresión.
El siguiente item,`)` (paréntesis derecho), produce que se vacié la pila hasta un `(` .La pila, en este momento, dispone de estos operadores:

`abc+def^/` El algoritmo saca operadores de la pila hasta un '(' y da lugar a la pila:

`abc+def^/-` El operador – pasa a la pila y, a su vez, se extrae `-`; el siguiente carácter, el operando `g` pasa a la expresión.
`abc+def^/-g` El siguiente carácter es `)`, por lo que son extraídos de la pila los operadores hasta un `(`, la pila queda de la siguiente forma:

`abc+def^/-g-` El siguiente carácter es el operador `-`, hace que se saque de la pila el operador * y se meta en la pila el operador –.
`abc+def^/-g-*` Por último, el operando h pasa directamente a la expresión.

`abc+def^/-g-*h` Fin de entrada, se vacía la pila pasando los operadores a la expresión:
`abc+def^/-g-*h-`

El seguimiento realizado pone de manifiesto la importancia de considerar al paréntesis izquierdo un operador de mínima prioridad dentro de la pila, para que los operadores, dentro de un paréntesis, se metan en la pila y después extraerlos cuando se trata el paréntesis derecho. También tiene un comportamiento distinto el operador de potenciación dentro y fuera de la pila, debido a que tienen asociatividad de derecha a izquierda. Las prioridades se fijan en la Tabla 11.1.

Tabla 11.1. Tabla de prioridades de los operadores considerados.

Operador	Prioridad dentro pila	Prioridad fuera pila
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Observe, que el paréntesis derecho no se considera ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo.

Algoritmo de paso de notación *infija* a *postfija*

Los pasos a seguir para transformar una expresión algebraica de *notación infija* a *postfija*:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es un operando, pasarlo a la expresión postfija.
3. Si es operador:
   3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir de 1.
   3.2. Si la pila no está vacía:
   Si prioridad del operador es mayor que prioridad del operador cima, meterlo en la pila y repetir a partir de 1.
   Si prioridad del operador es menor o igual que prioridad del operador cima, sacar operador cima de la pila y ponerlo en la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
   4.1. Sacar operador cima y ponerlo en la expresión postfija.
   4.2. Si el nuevo operador cima es paréntesis izquierdo, suprimir elemento cima.
   4.3. Si cima no es paréntesis izquierdo, volver a 4.1.
   4.4. Volver a partir de 1.
5. Si quedan elementos en la pila pasarlos a la expresión postfija.
6. Fin del algoritmo.

Codificación del algoritmo de transformación a *postfija*

Se necesita crear una pila de caracteres para guardar los operadores. Se utiliza el diseño de Pila genérica del Apartado 11.3. La expresión original se lee del teclado en una cadena de suficiente tamaño. También se declara una estructura para representar un elemento de la ex-

presión, con un campo carácter para el operando o el operador, y el otro campo para indicar si es operador u operando.

```
struct Elemento
{
  char c;
  bool oprdor;
};
```

La función `postFija()` implementa los pasos del algoritmo de transformación, recibe como argumento una cadena con la expesión, crea una pila y en un bucle de tantas iteraciones como caracteres realiza las acciones del algoritmo. La función define el array `Elemento* expresión`, a la que se pasan los elementos que forman la expresión en postfija. Una vez que termina la transformación, la función devuelve la expresión en *notación postfija* y el número de elementos de que consta agrupados en la siguiente estructura:

```
struct Expresion
{
  Elemento* expr;
  int n;
};
```

El archivo `TiposExpresio.h` contiene el tipo `Elemento` y el tipo `Expresion`.

En la función `prdadDentro()`se fija la prioridad de un operador dentro de la pila, y en `prdadFuera()`la prioridad de un operador fuera de la pila.

```
//archivo postFija.cpp

#include <cstdlib>
#include <string.h>
#include <ctype.h>
#include "PilaGenerica.h"
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
int prdadFuera (char op);
int prdadDentro (char op);
bool valido(const char* expresion);
bool operando(char c);

Expresion postFija(const char* expOrg)
{
  PilaGenerica<char> pila;
  Elemento* expsion;
  bool desapila;
  int n = -1;   //contador de expresión en postfija

  if (! valido(expOrg))  // verifica los caracteres de la expresión
     throw "Carácter no válido en una expresión";

  expsion = new Elemento[strlen(expOrg)];
  for (int i = 0; i < strlen(expOrg); i++)
  {
    char ch, opeCima;
    ch = toupper(expOrg[i]);    // operandos en mayúsculas
```

```
    if (operando(ch))           // análisis del elemento
    {
      expsion[++n].c = ch;
      expsion[n].oprdor = false;
    }
    else if (ch != ')')         // es un operador
    {
      desapila = true;
      while (desapila)
      {
        opeCima = ' ';
        if (!pila.pilaVacia())
          opeCima = pila.cimaPila();
          if (pila.pilaVacia() ||
            (prdadFuera(ch) > prdadDentro(opeCima)))
          {
            pila.insertar(ch);
            desapila = false;
          }
          else if (prdadFuera(ch) <= prdadDentro(opeCima))
          {
            expsion[++n].c = pila.quitar();
            expsion[n].oprdor = true;
          }
        }
    }
    else                        // es un ')'
    {
      opeCima = pila.quitar();
      do{
        expsion[++n].c = opeCima;
        expsion[n].oprdor = true;
        opeCima = pila.quitar();
      }while (opeCima != '(');
    }
  }
    /*
       se vuelca los operadores que quedan en la pila y se pasan a la ex-
       presión.
    */
  while (!pila.pilaVacia())
  {
    expsion[++n].c = pila.quitar();
    expsion[n].oprdor = true;
  }
  Expresion post;
  post.expr = expsion;
  post.n = n;
  return post;   // expresión en postfija
}
// prioridad del operador dentro de la pila

int prdadDentro(char op)
{
  int pdp;
  switch (op)
```

```
  {
    case '^':  pdp = 3;
               break;
    case '*': case '/':
               pdp = 2;
               break;
    case '+': case '-':
               pdp = 1;
               break;
    case '(':  pdp = 0;
  }
  return pdp;
}
// prioridad del operador en la expresión infija
int prdadFuera(char op)
{
  int pfp;
  switch (op)
  {
    case '^':  pfp = 4;
               break;
    case '*': case '/':
               pfp = 2;
               break;
    case '+': case '-':
               pfp = 1;
               break;
    case '(':  pfp = 5;
  }
  return pfp;
}

//analiza cada carácter de la expresión
bool valido(const char* expresion)
{
  bool sw = true;
  for (int i = 0; (i < strlen(expresion))&& sw; i++)
  {
    char c;
    c = expresion[i];
    sw = sw && (
                (c >= 'A' && c <= 'Z') ||
                (c >= 'a' && c <= 'z') ||
                (c == '^' || c == '/' || c== '*' ||
                  c == '+' || c == '-' || c== '\n' ||
                  c == '(' || c == ')' )
               );
  }
  return sw;
}

bool operando(char c)
{

//determina si c es un operando
  return (c >= 'A' && c <= 'Z');
}
```

11.4.4. Evaluación de la expresión en notación *postfija*

Una vez que se tiene la expresión en *notación postfija* se evalúa la expresión. Evaluar significa obtener un resultado de la expresión para valores particulares de los operandos. De nuevo, el algoritmo de evaluación utiliza una pila, en esta ocasión de operandos, es decir, de números reales.

Al describir el algoritmo `expsion` es el array con la la expresión postfija. El número de elementos es la longitud, `n`, de la cadena.

1. Examinar `expsion` elemento a elemento: repetir los pasos 2 y 3 para cada elemento.
2. Si el elemento es un operando, meterlo en la pila.
3. Si el elemento es un operador, se simboliza con `&`, entonces:
   - Sacar los dos elementos superiores de la pila, se denominarán `b` y `a` respectivamente.
   - Evaluar `a & b`, el resultado es `z = a & b`.
   - El resultado `z`, meterlo en la pila. Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificación de la evaluación de expresión en postfija

La función `double evalua()` implementa el algoritmo, recibe la expresión en postfija y el array con el valor de cada operando. La pila de números reales, utilizada por el algoritmo, se instancia de la clase genérica `PilaGenerica` para elementos de tipo `double`.

La función `valorOprdos()` da entrada a los valores de los operandos. Estos valores se guardan en un array, cada posición del array se corresponde con una letra, que a su vez es un operando.

```
// archivo evaluaExpresion.h

#include "pilagenerica.h"
#include "TiposExpresio.h"
#include <math.h>

double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

double evalua(Expresion postFija, double v[])
{
  PilaGenerica<double> pila;
  double valor, a, b;

  for (int i = 0; i <= postFija.n; i++)
  {
    char op;

    if (postFija.expr[i].oprdor)     // es un operador
    {
      op = postFija.expr[i].c;
```

```
        b = pila.quitar();
        a = pila.quitar();
        switch (op)
        {
          case '^': valor = pow(a,b);
                    break;
          case '*': valor = a * b;
                    break;
          case '/': if (b != 0.0)
                      valor = a / b;
                    else
                      throw "División por cero.";
                    break;
          case '+': valor = a + b;
                    break;
          case '-': valor = a – b;
        }
        pila.insertar(valor);
      }
      else               // es un operando
      {
        int indice;
        op = postFija.expr[i].c;
        indice = op - 'A';       // posición en array de valores
        pila.insertar(v[indice]);
      }
    }
    return pila.quitar();        // resultado de la expresión
  }

  // asignan valores numéricos a los operandos
  void valorOprdos(Expresion ep, double v[])
  {
    char ch;
    for (int i = 0; i <= ep.n; i++)
    {
      char op;
      op = ep.expr[i].c;
      if (! ep.expr[i].oprdor)   // es un operando
      {
        int indice;
        double d;
        indice = op - 'A';
        cout << op << " = ";
        cin >> v[indice];
      }
    }
  }
```

Programa

La función `main()` controla las etapas principales del algoritmo: petición de la expresión algebraica, llamada a la función que transforma a notación postfija y, por último, evaluar la expresión para unos valores concretos de los operandos.

```
#include <iostream>
using namespace std;
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

int main()
{
  double v[26];
  double resultado;
  char expresion[81];
  Expresion ex;

  cout << "\nExpresión aritmética: ";
  cin.getline(expresion, 80);
  // Conversión de infija a postfija
  expr = postFija(expresion);
  cout << "\nExpresión en postfija: ";
  for (int i = 0; i <= ex.n; i++)
  {
    cout << ex.expr[i];
  }
  // Evaluación de la expresión
  valorOprdos(ex, v);       // valor de operandos
  resultado = evalua(ex, v);
  cout << "Resultado = " << resultado;
  return 0;
}
```

RESUMEN

Una *pila* es una estructura de datos tipo **LIFO** (*last in first out*, último en entrar primero en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila. Se definen las siguientes operaciones básicas sobre pilas: `crear`, `insertar`, `cima`, `quitar`, `pilaVacia`, `pilaLlena` y `limpiarPila`.

`insertar`, añade un elemento en la cima de la pila. Debe de haber espacio en la pila.
`cima`, devuelve el elemento que está en la cima, sin extraerlo.
`quitar`, extrae de la pila el elemento cima de la pila.
`pilaVacia`, determina si el estado de la pila es vacía.
`pilaLlena`, determina si existe espacio en la pila para añadir un nuevo elemento.
`limpiarPila`, el espacio asignado a la pila se libera, queda disponible.

Las aplicaciones de las pilas en la programación son numerosas, entre las que está la evaluación de expresiones aritméticas. Primero, se transforma la expresión a notación postfija, a continuación se evalúa.

Las expresiones en notación polaca, postfija o prefija, tienen la característica de que no necesitan paréntesis.