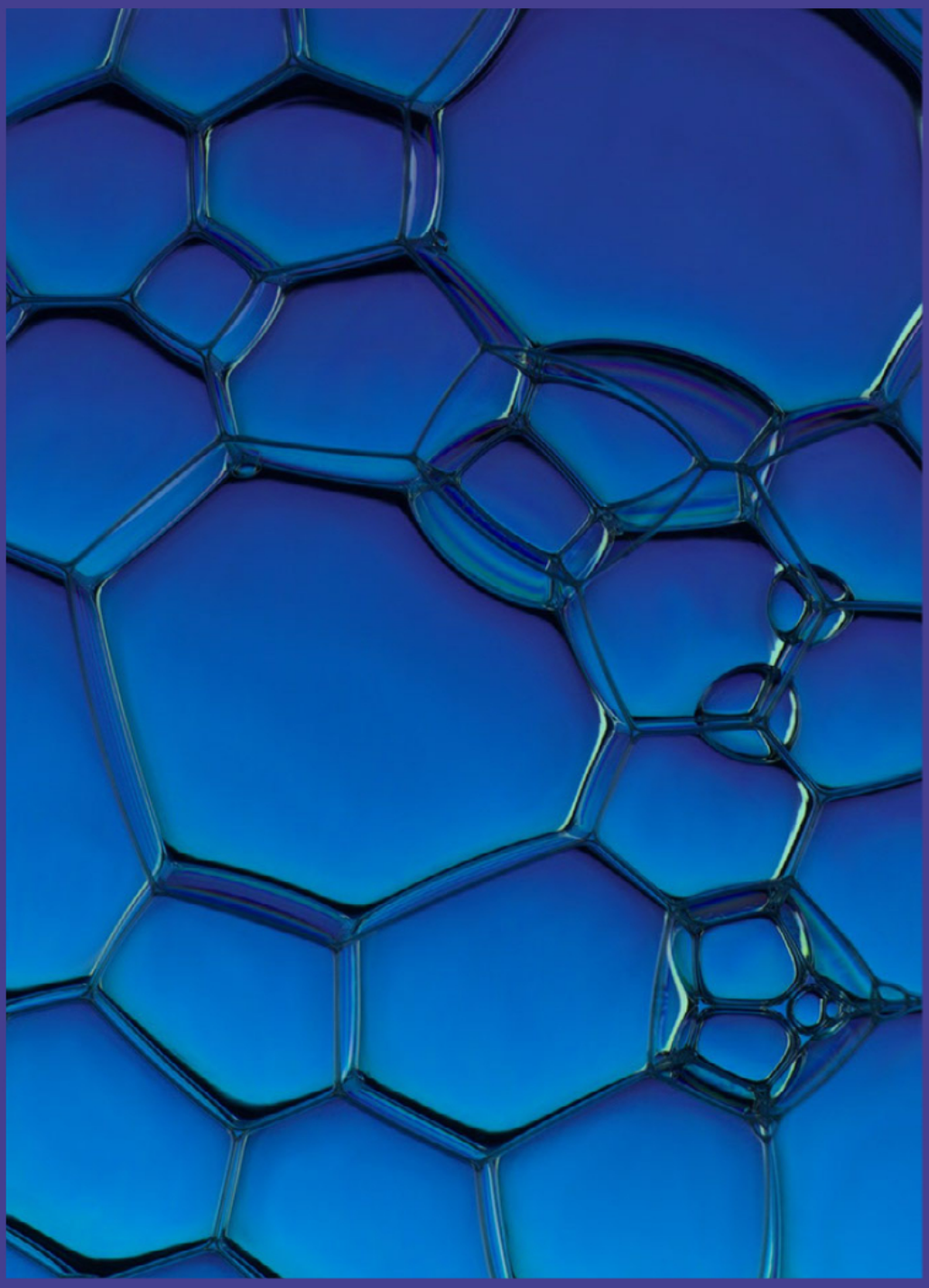
Estructuras de datos

Organización de la información



Salvador Pozo

http://conclase.net

Introducción

Una de las aplicaciones más interesantes y potentes de la memoria dinámica y de los punteros son, sin duda, las estructuras dinámicas de datos. Las estructuras básicas disponibles en C y C++ (structs y *arrays*) tienen una importante limitación: no pueden cambiar de tamaño durante la ejecución. Los *arrays* están compuestos por un determinado número de elementos, número que se decide en la fase de diseño, antes de que el programa ejecutable sea creado.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Por supuesto, podemos crear *arrays* dinámicos, pero una vez creados, tu tamaño también será fijo, y para hacer que crezcan o disminuyan de tamaño, deberemos reconstruirlos desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas. Pero no sólo eso, como veremos, también nos permitirán crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

Las estructuras de datos están compuestas de otras pequeñas estructuras a las que llamaremos nodos o elementos, que agrupan los datos con los que trabajará nuestro programa y además uno o más punteros autoreferenciales, es decir, punteros a objetos del mismo tipo nodo.

Una estructura básica de un nodo para crear listas de datos seria:

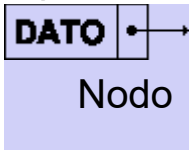
```
struct nodo {
   int dato;
```

```
    struct nodo *otronodo;
};
```

El campo "otronodo" puede apuntar a un objeto del tipo nodo. De este modo, cada nodo puede usarse como un ladrillo para construir listas de datos, y cada uno mantendrá ciertas relaciones con otros nodos.

Para acceder a un nodo de la estructura sólo necesitaremos un puntero a un nodo.

Durante el presente curso usaremos gráficos para mostrar la estructura de las estructuras de datos dinámicas. El nodo anterior se representará asi:



Nodo

Las estructuras dinámicas son una implementación de TDAs o TADs (Tipos Abstractos de Datos). En estos tipos el interés se centra más en la estructura de los datos que en el tipo concreto de información que almacenan.

Dependiendo del número de punteros y de las relaciones entre nodos, podemos distinguir varios tipos de estructuras dinámicas. Enumeraremos ahora sólo de los tipos básicos:

- Listas abiertas: cada elemento sólo dispone de un puntero, que apuntará al siguiente elemento de la lista o valdrá NULL si es el último elemento.
- Pilas: son un tipo especial de lista, conocidas como listas LIFO (Last In, First Out: el último en entrar es el primero en salir). Los elementos se "amontonan" o apilan, de modo que sólo el elemento que está encima de la pila puede ser leído, y sólo pueden añadirse elementos encima de la pila.
- Colas: otro tipo de listas, conocidas como listas FIFO (First In, First Out: El primero en entrar es el primero en salir). Los elementos se almacenan en fila, pero sólo pueden añadirse por un extremo y leerse por el otro.
- Listas circulares: o listas cerradas, son parecidas a las listas abiertas, pero el último elemento apunta al primero. De hecho, en las listas circulares no puede hablarse de "primero" ni de "último". Cualquier nodo puede ser el nodo de entrada y salida.

- Listas doblemente enlazadas: cada elemento dispone de dos punteros, uno a punta al siguiente elemento y el otro al elemento anterior. Al contrario que las listas abiertas anteriores, estas listas pueden recorrerse en los dos sentidos.
- Arboles: cada elemento dispone de dos o más punteros, pero las referencias nunca son a elementos anteriores, de modo que la estructura se ramifica y crece igual que un árbol.
- Arboles binarios: son árboles donde cada nodo sólo puede apuntar a dos nodos.
- Arboles binarios de búsqueda (ABB): son árboles binarios ordenados. Desde cada nodo todos los nodos de una rama serán mayores, según la norma que se haya seguido para ordenar el árbol, y los de la otra rama serán menores.
- Arboles AVL: son también árboles de búsqueda, pero su estructura está más optimizada para reducir los tiempos de búsqueda.
- Arboles B: son estructuras más complejas, aunque también se trata de árboles de búsqueda, están mucho más optimizados que los anteriores.
- Tablas HASH: son estructuras auxiliares para ordenar listas.
- Grafos: es el siguiente nivel de complejidad, podemos considerar estas estructuras como árboles no jerarquizados.
- Diccionarios.

Al final del curso también veremos estructuras dinámicas en las que existen nodos de distintos tipos, en realidad no es obligatorio que las estructuras dinámicas estén compuestas por un único tipo de nodo, la flexibilidad y los tipos de estructuras sólo están limitados por tu imaginación como programador.

Capítulo 1 Listas abiertas

1.1 Definición

La forma más simple de estructura dinámica es la lista abierta. En esta forma los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el puntero del nodo siguiente vale NULL.

En las listas abiertas existe un nodo especial: el primero. Normalmente diremos que nuestra lista es un puntero a ese primer nodo y llamaremos a ese nodo la cabeza de la lista. Eso es porque mediante ese único puntero podemos acceder a toda la lista.

Cuando el puntero que usamos para acceder a la lista vale NULL, diremos que la lista está vacía.

El nodo típico para construir listas tiene esta forma:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

En el ejemplo, cada elemento de la lista sólo contiene un dato de tipo entero, pero en la práctica no hay límite en cuanto a la complejidad de los datos a almacenar.

1.2 Declaraciones de tipos para manejar listas en C

Normalmente se definen varios tipos que facilitan el manejo de las listas, en C, la declaración de tipos puede tener una forma

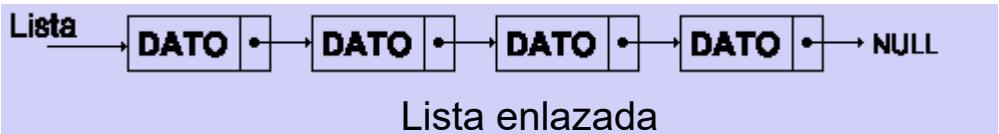
parecida a esta:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

tipoNodo es el tipo para declarar nodos, evidentemente.

pNodo es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas, como puede verse, un puntero a un nodo y una lista son la misma cosa. En realidad, cualquier puntero a un nodo es una lista, cuyo primer elemento es el nodo apuntado.



Lista enlazada

Es muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, ya que si no existe ninguna copia de ese valor, y se pierde, será imposible acceder al nodo y no podremos liberar el espacio de memoria que ocupa.

1.3 Operaciones básicas con listas

Con las listas tendremos un pequeño repertorio de operaciones básicas que se pueden realizar:

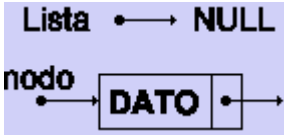
- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través de una lista, anterior, siguiente, primero.

Cada una de estas operaciones tendrá varios casos especiales, por ejemplo, no será lo mismo insertar un nodo en una lista vacía, o al principio de una lista no vacía, o la final, o en una posición intermedia.

1.4 Insertar elementos en una lista abierta

Veremos primero los casos sencillos y finalmente construiremos un algoritmo genérico para la inserción de elementos en una lista.

Insertar un elemento en una lista vacía



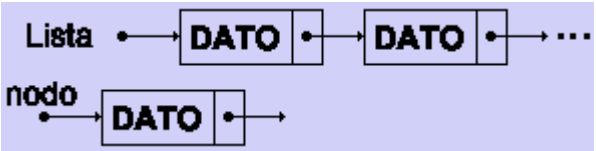
Lista vacía

Este es, evidentemente, el caso más sencillo. Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero a la lista valdrá NULL:

El proceso es muy simple, bastará con que:

1. nodo->siguiente apunte a NULL.
2. Lista apunte a nodo.

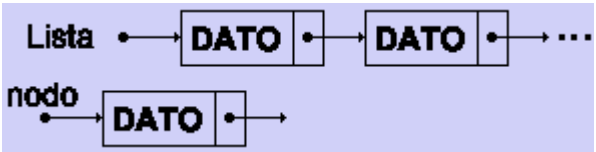
Insertar un elemento en la primera posición de una lista



Insertar al principio

Podemos considerar el caso anterior como un caso particular de éste, la única diferencia es que en el caso anterior la lista es una lista vacía, pero siempre podemos, y debemos considerar una lista vacía como una lista.

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una lista, en este caso no vacía:



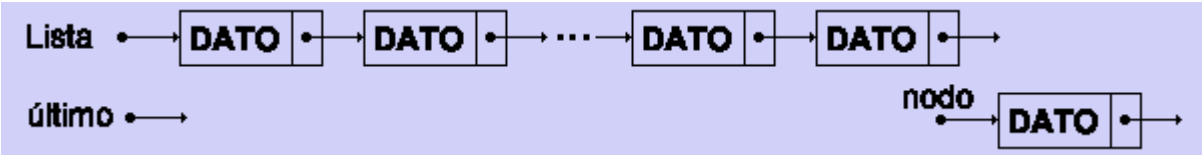
Insertado al principio

El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a Lista.
2. Hacemos que Lista apunte a nodo.

Insertar un elemento en la última posición de una lista

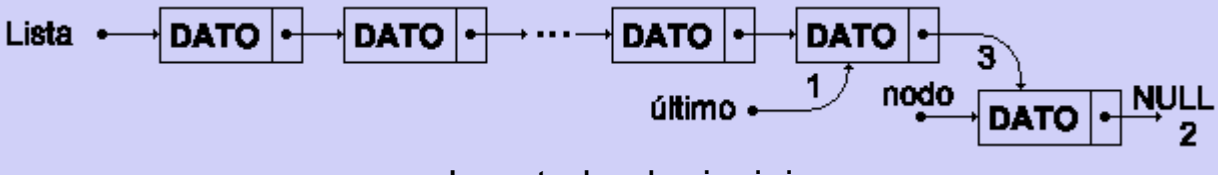
Este es otro caso especial. Para este caso partiremos de una lista no vacía:



Insertar al final

El proceso en este caso tampoco es excesivamente complicado:

1. Necesitamos un puntero que señale al último elemento de la lista. La manera de conseguirlo es empezar por el primero y avanzar hasta que el nodo que tenga como siguiente el valor NULL.
2. Hacer que nodo->siguiente sea NULL.
3. Hacer que ultimo->siguiente sea nodo.

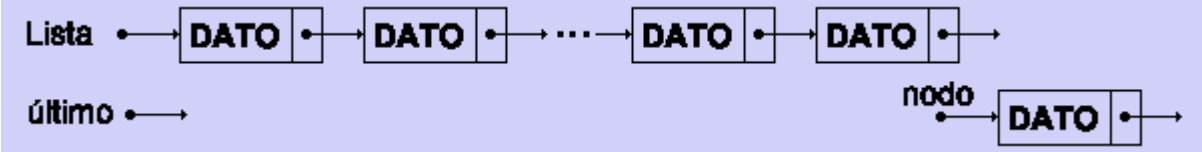


Insertado al principio

Insertar un elemento a continuación de un nodo cualquiera de una lista

De nuevo podemos considerar el caso anterior como un caso particular de este. Ahora el nodo "anterior" será aquel a continuación del cual insertaremos el nuevo nodo:

Suponemos que ya disponemos del nuevo nodo a insertar, apuntado por nodo, y un puntero al nodo a continuación del que lo

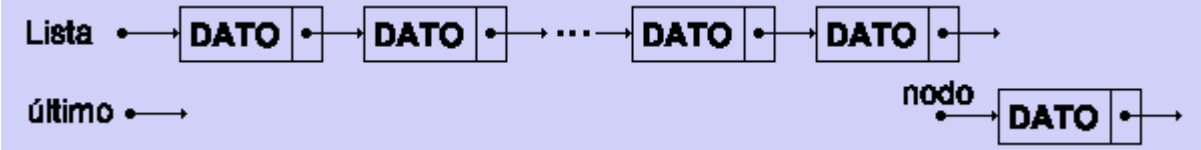


Insertar dentro

insertaremos.

El proceso a seguir será:

1. Hacer que nodo->siguiente señale a anterior->siguiente.
2. Hacer que anterior->siguiente señale a nodo.



Insertado dentro

1.5 Localizar elementos en una lista abierta

Muy a menudo necesitaremos recorrer una lista, ya sea buscando un valor particular o un nodo concreto. Las listas abiertas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente, pero no se puede obtener, por ejemplo, un puntero al nodo anterior desde un nodo cualquiera si no se empieza desde el principio.

Para recorrer una lista procederemos siempre del mismo modo, usaremos un puntero auxiliar como índice:

1. Asignamos al puntero índice el valor de Lista.
2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
3. Dentro del bucle asignaremos al índice el valor del nodo siguiente al índice actual.

Por ejemplo, para mostrar todos los valores de los nodos de una lista, podemos usar el siguente bucle en C:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
...
pNodo indice;
...
indice = Lista;
while(indice) {
   printf("%d\n", indice->dato);
   indice = indice->siguiente;
}
...
```

Supongamos que sólo queremos mostrar los valores hasta que encontremos uno que sea mayor que 100, podemos sustituir el bucle por:

```
...
indice = Lista;
while(indice && indice->dato <= 100) {
   printf("%d\n", indice->dato);
   indice = indice->siguiente;
}
...
```

Si analizamos la condición del bucle, tal vez encontremos un posible error: ¿Qué pasaría si ningún valor es mayor que 100, y alcancemos el final de la lista?. Podría pensarse que cuando indice sea NULL, si intentamos acceder a indice->dato se producirá un error.

En general eso será cierto, no puede accederse a punteros nulos. Pero en este caso, ese acceso está dentro de una condición y forma parte de una expresión "and". Recordemos que cuando se evalúa una expresión "and", se comienza por la izquierda, y la

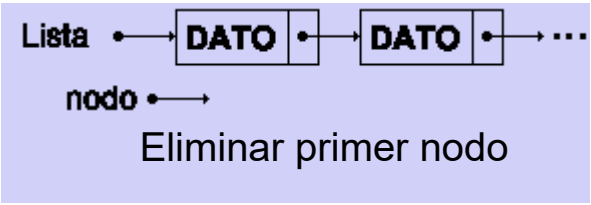
evaluación se abandona cuando una de las expresiones resulta falsa, de modo que la expresión "indice->dato <= 100" nunca se evaluará si indice es NULL.

Si hubiéramos escrito la condición al revés, el programa nunca funcionaría bien. Esto es algo muy importante cuando se trabaja con punteros.

1.6 Eliminar elementos en una lista abierta

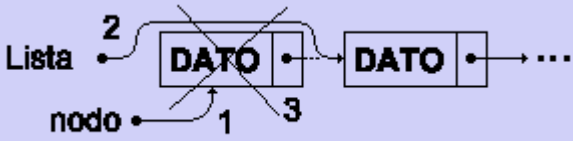
De nuevo podemos encontrarnos con varios casos, según la posición del nodo a eliminar.

Eliminar el primer nodo de una lista abierta



Eliminar primer nodo

Es el caso más simple. Partiremos de una lista con uno o más nodos, y usaremos un puntero auxiliar, nodo:



Primer nodo eliminado

1. Hacemos que nodo apunte al primer elemento de la lista, es decir a Lista.
2. Asignamos a Lista la dirección del segundo nodo de la lista: Lista->siguiente.
3. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

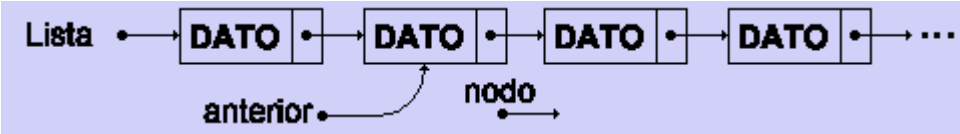
Si no guardamos el puntero al primer nodo antes de actualizar Lista, después nos resultaría imposible liberar la memoria que ocupa. Si liberamos la memoria antes de actualizar Lista, perderemos el puntero al segundo nodo.

Si la lista sólo tiene un nodo, el proceso es también válido, ya que el valor de Lista->siguiente es NULL, y después de eliminar el primer nodo la lista quedará vacía, y el valor de Lista será NULL.

De hecho, el proceso que se suele usar para borrar listas completas es eliminar el primer nodo hasta que la lista esté vacía.

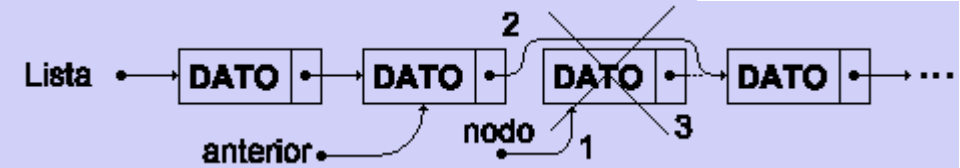
Eliminar un nodo cualquiera de una lista abierta

En todos los demás casos, eliminar un nodo se puede hacer siempre del mismo modo. Supongamos que tenemos una lista con al menos dos elementos, y un puntero al nodo anterior al que queremos eliminar. Y un puntero auxiliar nodo.



Eliminar un nodo

El proceso es parecido al del caso anterior:



Nodo eliminado

1. Hacemos que nodo apunte al nodo que queremos borrar.
2. Ahora, asignamos como nodo siguiente del nodo anterior, el siguiente al que queremos eliminar: anterior->siguiente = nodo->siguiente.
3. Eliminamos la memoria asociada al nodo que queremos eliminar.

Si el nodo a eliminar es el último, es procedimiento es igualmente válido, ya que anterior pasará a ser el último, y anterior->siguiente valdrá NULL.

1.7 Moverse a través de una lista abierta

Sólo hay un modo de moverse a través de una lista abierta, hacia delante.

Aún así, a veces necesitaremos acceder a determinados elementos de una lista abierta. Veremos ahora como acceder a los más corrientes: el primero, el último, el siguiente y el anterior.

Primer elemento de una lista

El primer elemento es el más accesible, ya que es a ese a que apunta el puntero que define la lista. Para obtener un puntero al primer elemento bastará con copiar el puntero Lista.

Elemento siguiente a uno cualquiera

Supongamos que tenemos un puntero nodo que señala a un elemento de una lista. Para obtener un puntero al siguiente bastará con asignarle el campo "siguiente" del nodo, nodo->siguiente.

Elemento anterior a uno cualquiera

Ya hemos dicho que no es posible retroceder en una lista, de modo que para obtener un puntero al nodo anterior a uno dado tendremos que partir del primero, e ir avanzando hasta que el nodo siguiente sea precisamente nuestro nodo.

Último elemento de una lista

Para obtener un puntero al último elemento de una lista partiremos de un nodo cualquiera, por ejemplo el primero, y avanzaremos hasta que su nodo siguiente sea NULL.

Saber si una lista está vacía

Basta con comparar el puntero Lista con NULL, si Lista vale NULL la lista está vacía.

1.8 Borrar una lista completa

El algoritmo genérico para borrar una lista completa consiste simplemente en borrar el primer elemento sucesivamente mientras la lista no esté vacía.

1.9 Ejemplo de lista abierta ordenada en C

Supongamos que queremos construir una lista para almacenar números enteros, pero de modo que siempre esté ordenada de menor a mayor. Para hacer la prueba añadiremos los valores 20, 10, 40, 30. De este modo tendremos todos los casos posibles. Al comenzar, el primer elemento se introducirá en una lista vacía, el segundo se insertará en la primera posición, el tercero en la última, y el último en una posición intermedia.

Insertar un elemento en una lista vacía es equivalente a insertarlo en la primera posición. De modo que no incluiremos una función para asignar un elemento en una lista vacía, y haremos que la función para insertar en la primera posición nos sirva para ese caso también.

Algoritmo de inserción

1. El primer paso es crear un nodo para el dato que vamos a insertar.
2. Si Lista es NULL, o el valor del primer elemento de la lista es mayor que el del nuevo, insertaremos el nuevo nodo en la primera posición de la lista.
3. En caso contrario, buscaremos el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior->siguiente no sea NULL y el dato que contiene anterior->siguiente sea menor o igual que el dato que queremos insertar.
4. Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

```
void Insertar(Lista *lista, int v) {
   pNodo nuevo, anterior;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Si la lista está vacía */
   if(ListaVacia(*lista) || (*lista)->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = *lista;
      /* Ahora, el comienzo de nuestra lista es en nuevo
nodo */
      *lista = nuevo;
   } else {
      /* Buscar el nodo de valor menor a v */
      anterior = *lista;
      /* Avanzamos hasta el último elemento o hasta que el
siguiente tenga
         un valor mayor que v */
      while(anterior->siguiente && anterior->siguiente-
>valor <= v)
         anterior = anterior->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior
*/
      nuevo->siguiente = anterior->siguiente;
      anterior->siguiente = nuevo;
   }
}
```

Algoritmo para borrar un elemento

Después probaremos la función para buscar y borrar, borraremos los elementos 10, 15, 45, 30 y 40, así probaremos los casos de borrar el primero, el último y un caso intermedio o dos nodos que no existan.

Recordemos que para eliminar un nodo necesitamos disponer de un puntero al nodo anterior.

1. Lo primero será localizar el nodo a eliminar, si es que existe. Pero sin perder el puntero al nodo anterior. Partiremos del nodo

primero, y del valor NULL para anterior. Y avanzaremos mientras nodo no sea NULL o mientras que el valor almacenado en nodo sea menor que el que buscamos.
2. Ahora pueden darse tres casos:
3. Que el nodo sea NULL, esto indica que todos los valores almacenados en la lista son menores que el que buscamos y el nodo que buscamos no existe. Retornaremos sin borrar nada.
4. Que el valor almacenado en nodo sea mayor que el que buscamos, en ese caso también retornaremos sin borrar nada, ya que esto indica que el nodo que buscamos no existe.
5. Que el valor almacenado en el nodo sea igual al que buscamos.
6. De nuevo existen dos casos:
7. Que anterior sea NULL. Esto indicaría que el nodo que queremos borrar es el primero, así que modificamos el valor de Lista para que apunte al nodo siguiente al que queremos borrar.
8. Que anterior no sea NULL, el nodo no es el primero, así que asignamos a anterior->siguiente la dirección de nodo->siguiente.
9. Después de 7 u 8, liberamos la memoria de nodo.

```
void Borrar(Lista *lista, int v) {
   pNodo anterior, nodo;

   nodo = *lista;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
   else { /* Borrar el nodo */
      if(!anterior) /* Primer elemento */
         *lista = nodo->siguiente;
      else  /* un elemento cualquiera */
         anterior->siguiente = nodo->siguiente;
      free(nodo);
   }
}
```

Código del ejemplo completo

```
#include <stdlib.h>
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;


typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;

/* Funciones con listas: */
void Insertar(Lista *l, int v);
void Borrar(Lista *l, int v);

int ListaVacia(Lista l);

void BorrarLista(Lista *);
void MostrarLista(Lista l);

int main() {
   Lista lista = NULL;
   pNodo p;

   Insertar(&lista, 20);
   Insertar(&lista, 10);
   Insertar(&lista, 40);
   Insertar(&lista, 30);

   MostrarLista(lista);

   Borrar(&lista, 10);
   Borrar(&lista, 15);
   Borrar(&lista, 45);
   Borrar(&lista, 30);
   Borrar(&lista, 40);

   MostrarLista(lista);

   BorrarLista(&lista);

   return 0;
```

```
}

void Insertar(Lista *lista, int v) {
   pNodo nuevo, anterior;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Si la lista está vacía */
   if(ListaVacia(*lista) || (*lista)->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = *lista;
      /* Ahora, el comienzo de nuestra lista es en nuevo
nodo */
      *lista = nuevo;
   } else {
      /* Buscar el nodo de valor menor a v */
      anterior = *lista;
      /* Avanzamos hasta el último elemento o hasta que el
siguiente tenga
         un valor mayor que v */
      while(anterior->siguiente && anterior->siguiente-
>valor <= v)
         anterior = anterior->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior
*/
      nuevo->siguiente = anterior->siguiente;
      anterior->siguiente = nuevo;
   }
}

void Borrar(Lista *lista, int v) {
   pNodo anterior, nodo;

   nodo = *lista;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
   else { /* Borrar el nodo */
      if(!anterior) /* Primer elemento */
         *lista = nodo->siguiente;
      else  /* un elemento cualquiera */
         anterior->siguiente = nodo->siguiente;
      free(nodo);
```

```
    }
}

int ListaVacia(Lista lista) {
   return (lista == NULL);
}

void BorrarLista(Lista *lista) {
   pNodo nodo;

   while(*lista) {
      nodo = *lista;
      *lista = nodo->siguiente;
      free(nodo);
   }
}

void MostrarLista(Lista lista) {
   pNodo nodo = lista;

   if(ListaVacia(lista)) printf("Lista vacía\n");
   else {
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->siguiente;
     }
     print("\n");
   }
}
```

Fichero con el código fuente.

1.10 Ejemplo de lista abierta en C++ usando clases

Usando clases el programa cambia bastante, aunque los algoritmos son los mismos.

Para empezar, necesitaremos dos clases, una para nodo y otra para lista. Además la clase para nodo debe ser amiga de la clase lista, ya que ésta debe acceder a los miembros privados de nodo.

```
class nodo {
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class lista;
};

typedef nodo *pnodo;

class lista {
   public:
    lista() { primero = actual = NULL; }
    ~lista();

    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return primero == NULL; }
    void Mostrar();
    void Siguiente();
    void Primero();
    void Ultimo();
    bool Actual() { return actual != NULL; }
    int ValorActual() { return actual->valor; }

   private:
    pnodo primero;
    pnodo actual;
};
```

Hemos hecho que la clase para lista sea algo más completa que la equivalente en C, aprovechando las prestaciones de las clases. En concreto, hemos añadido funciones para mantener un puntero a un elemento de la lista y para poder moverse a través de ella.

Los algoritmos para insertar y borrar elementos son los mismos que expusimos para el ejemplo C, tan sólo cambia el modo de crear

y destruir nodos.

Código del ejemplo completo

```
#include <iostream>
using namespace std;

class nodo {
   public:
    nodo(int v, nodo *sig = NULL)
    {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class lista;
};

typedef nodo *pnodo;

class lista {
   public:
    lista() { primero = actual = NULL; }
    ~lista();

    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return primero == NULL; }
    void Mostrar();
    void Siguiente() { if(actual) actual = actual-
>siguiente; }
    void Primero() { actual = primero; }
    void Ultimo() { Primero(); if(!ListaVacia())
while(actual->siguiente) Siguiente(); }
    bool Actual() { return actual != NULL; }
    int ValorActual() { return actual->valor; }

   private:
    pnodo primero;
    pnodo actual;
```

```
};

lista::~lista() {
   pnodo aux;

   while(primero) {
      aux = primero;
      primero = primero->siguiente;
      delete aux;
   }
   actual = NULL;
}

void lista::Insertar(int v) {
   pnodo anterior;

   // Si la lista está vacía
   if(ListaVacia() || primero->valor > v) {
      // Asignamos a lista un nuevo nodo de valor v y
      // cuyo siguiente elemento es la lista actual
      primero = new nodo(v, primero);
   } else {
      // Buscar el nodo de valor menor a v
      anterior = primero;
      // Avanzamos hasta el último elemento o hasta que el
siguiente tenga
      // un valor mayor que v
      while(anterior->siguiente && anterior->siguiente-
>valor <= v)
         anterior = anterior->siguiente;
      // Creamos un nuevo nodo después del nodo anterior, y
cuyo siguiente
      // es el siguiente del anterior
      anterior->siguiente = new nodo(v, anterior-
>siguiente);
   }
}

void lista::Borrar(int v) {
   pnodo anterior, nodo;

   nodo = primero;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
```

```
   else { // Borrar el nodo
      if(!anterior) // Primer elemento
         primero = nodo->siguiente;
      else  // un elemento cualquiera
         anterior->siguiente = nodo->siguiente;
      delete nodo;
   }
}

void lista::Mostrar() {
   nodo *aux;

   aux = primero;
   while(aux) {
      cout << aux->valor << "-> ";
      aux = aux->siguiente;
   }
   cout << endl;
}

int main() {
   lista Lista;

   Lista.Insertar(20);
   Lista.Insertar(10);
   Lista.Insertar(40);
   Lista.Insertar(30);

   Lista.Mostrar();

   cout << "Lista de elementos:" << endl;
   Lista.Primero();
   while(Lista.Actual()) {
      cout << Lista.ValorActual() << endl;
      Lista.Siguiente();
   }
   Lista.Primero();
   cout << "Primero: " << Lista.ValorActual() << endl;

   Lista.Ultimo();
   cout << "Ultimo: " << Lista.ValorActual() << endl;

   Lista.Borrar(10);
   Lista.Borrar(15);
   Lista.Borrar(45);
   Lista.Borrar(30);
   Lista.Borrar(40);
```

```
    Lista.Mostrar();

    return 0;
}
```

Fichero con el código fuente.

1.11 Ejemplo de lista abierta en C++ usando plantillas

El siguiente paso es usar plantillas (templates) para definir listas genéricas, cuyos nodos pueden ser objetos de cualquier tipo definido en nuestro programa.

El código es algo más complicado, al menos a primera vista. Pero comprobaremos que es mucho más flexible y versatil.

Seguimos necesitando dos clases, una para nodo y otra para lista. Pero ahora podremos usar esas clases para construir listas de cualquier tipo de datos. Además, definiremos la clase para nodo como local dentro de la clase para lista, de ese modo nos evitamos definir amistades entre clases. A fin de cuentas, la clase nodo sólo la usaremos en esta lista.

Código del un ejemplo completo

Es preferible declarar cada plantilla clase en un fichero independiente, de este modo podremos usarlas en otros programas fácilmente. Empezamos con una plantilla para lista:

```
// ListaAb.h: Plantilla para lista abierta ordenada
// C con Clase. (C) Marzo de 2002
// Plantilla para lista abierta
// Posibles mejoras:
// * Implementar constructor copia.

#ifndef _LISTAABIERTA_
```

```
#define _LISTAABIERTA_

template<class DATO>
class Lista {
  private:
   //// Clase local de Lista para Nodo de Lista:
   template<class DATON>
   class Nodo {
     public:
      // Constructor:
      Nodo(const DATON dat, Nodo<DATON> *sig) : dato(dat),
siguiente(sig) {}
      // Miembros:
      DATON dato;
      Nodo<DATON> *siguiente;
   };

   // Punteros de la lista, para cabeza y nodo actual:
   Nodo<DATO> *primero;
   Nodo<DATO> *actual;

  public:
   // Constructor y destructor básicos:
   Lista() : primero(NULL), actual(NULL) {}
   ~Lista();
   // Funciones de inserción:
   void InsertarFinal(const DATO dat);
   void InsertarPrincipio(const DATO dat);
   bool InsertarActual(const DATO dat);
   void Insertar(const DATO dat);
   // Funciones de borrado:
   void BorrarActual();
   bool BorrarPrimerValor(const DATO dat);
   // Función de búsqueda:
   bool BuscarPrimerValor(const DATO dat);
   // Comprobar si la lista está vacía:
   bool Vacia() { return primero==NULL; }
   // Devolver referencia al dato del nodo actual:
   DATO &ValorActual() { return actual->dato; }
   // Hacer que el nodo actual sea el primero:
   void Primero() { actual = primero; }
   // Comprobar si el nodo actual es válido:
   bool Actual() { return actual != NULL; }
   // Moverse al siguiente nodo de la lista:
   void Siguiente() { if(actual) actual = actual->siguiente;
}
   // Sobrecargar operator++ en forma sufija para los mismo:
   void operator++(int) { Siguiente(); }
```

```
   // Aplicar una función a cada elemento de la lista:
   void ParaCada(void (*func)(DATO&));
};

//////// Implementación:

// Destructor
template<class DATO>
Lista<DATO>::~Lista() {
   while(!Vacia()) {
      actual = primero;
      primero = primero->siguiente;
      delete actual;
   }
}

template<class DATO>
void Lista<DATO>::InsertarFinal(const DATO dat) {
   Nodo<DATO> *ultimo;

   // Si la lista está vacía, insertar al principio:
   if(Vacia()) InsertarPrincipio(dat);
   else { // Si no lo está:
      // Buscar el último nodo:
      ultimo = primero;
      while(ultimo->siguiente) ultimo = ultimo->siguiente;
      // Insertar a continuación:
      ultimo->siguiente = new Nodo<DATO>(dat, NULL);
   }
}

template<class DATO>
void Lista<DATO>::InsertarPrincipio(const DATO dat) {
   primero = new Nodo<DATO>(dat, primero);
}

template<class DATO>
bool Lista<DATO>::InsertarActual(const DATO dat) {
   // Sólo si la lista no está vacía y actual es válido:
   if(!Vacia() && actual) {
      actual->siguiente = new Nodo<DATO>(dat, actual-
>siguiente);
      return true;
   }
   // Si no se puede insertar, retornar con false:
   return false;
}
```

```
// Insertar ordenadamente:
template<class DATO>
void Lista<DATO>::Insertar(const DATO dat) {
   Nodo<DATO> *temp = primero;
   Nodo<DATO> *anterior = NULL;

   // Si la lista está vacía, insertar al principio:
   if(Vacia()) InsertarPrincipio(dat);
   else {
      // Buscar el nodo anterior al primer nodo con un dato
mayor qur 'dat'
      while(temp && temp->dato < dat) {
         anterior = temp;
         temp = temp->siguiente;
      }
      // Si no hay anterior, insertar al principio,
      // nuestro valor es el menor de la lista:
      if(!anterior)
         InsertarPrincipio(dat);
      else // Insertar:
         anterior->siguiente = new Nodo<DATO>(dat, temp);
   }
}

template<class DATO>
void Lista<DATO>::BorrarActual() {
   Nodo<DATO> *anterior;

   // Si el nodo actual es el primero:
   if(actual && actual == primero) {
      // El primer nodo será ahora el segundo:
      // Sacar el nodo actual de la lista:
      primero = actual->siguiente;
      // Borrarlo:
      delete actual;
      actual = NULL;
   } else
   if(actual && !Vacia()) {
      // Buscar el nodo anterior al actual:
      anterior = primero;
      while(anterior && anterior->siguiente != actual)
         anterior = anterior->siguiente;
      // Sacar el nodo actual de la lista:
      anterior->siguiente = actual->siguiente;
      // Borrarlo:
      delete actual;
      actual = NULL;
   }
```

```
}

// Borrar el primer nodo cuyo dato sea igual a 'dat':
template<class DATO>
bool Lista<DATO>::BorrarPrimerValor(const DATO dat) {
   Nodo<DATO> *anterior = NULL;
   Nodo<DATO> *temp = primero;

   if(!Vacia()) {
      // Si la lista no está vacía, buscar el nodo a borrar
(temp)
      // y el nodo anterior a ese (anterior):
      while(temp && temp->dato != dat) {
         anterior = temp;
         temp = temp->siguiente;
      }
      // Si el valor está en la lista:
      if(temp) {
         // Si anterior es válido, no es el primer valor de
la lista
         if(anterior) // Sacar nodo temp de la lista
            anterior->siguiente = temp->siguiente;
         else         // Ahora el primero es el segundo:
           primero = temp->siguiente;  // Borrar primer
elemento
         // Borrar nodo:
         delete temp;
         return true; // Se ha encontrado y borrado dat
      }
   }
   return false; // valor no encontrado
}

// Busca el primer nodo con valor 'dat':
template<class DATO>
bool Lista<DATO>::BuscarPrimerValor(const DATO dat) {
   actual = primero;

   // Si la lista no está vacía:
   if(!Vacia()) {
      while(actual && actual->dato != dat) {
         actual = actual->siguiente;
      }
   }
   // Si el nodo es válido, se ha encontrado el valor:
   return actual != NULL;
}
```

```
// Aplicar una función a cada nodo de la lista:
template<class DATO>
void Lista<DATO>::ParaCada(void (*func)(DATO&)) {
   Nodo<DATO> *temp = primero;

   // Recorrer la lista:
   while(temp) {
      // Aplicar la función:
      func(temp->dato);
      temp = temp->siguiente;
   }
}

// La función "func" debe ser una plantilla de una función
// que no retorne valor y que admita un parámetro del mismo
// tipo que la lista:
// template <class DATO>
// void <funcion>(DATO d);

#endif
```

Hemos introducido algunos refinamientos en nuestra clase que la harán más fácil de usar. Por ejemplo:

- Cuatro versiones distintas para insertar nodos, una para insertar al principio, otra al final, otra a continuación del nodo actual y una cuarta para insertar por orden. Ésta última nos permite crear listas ordenadas.
- Dos funciones para borrar valores, una borra el elemento actual y la otra el primer nodo que contenga el valor especificado.
- Sobrecarga del operador de postincremento, que nos permite movernos a lo largo de la lista de un modo más intuitivo.
- Función "ParaCada", que aplica una función a cada elemento de la lista. Veremos cómo podemos usar esta función para hacer cosas como mostrar la lista completa o incrementar todos los elementos.

En el tema de plantillas del curso de C++ ya hemos visto que existen algunas limitaciones para los tipos que se pueden emplear en plantillas. Por ejemplo, no podemos crear una lista de cadenas

usando Lista<**char** *>, ya que de ese modo sólo creamos una lista de punteros.

Para poder crear listas de cadenas hemos implementado una clase especial para cadenas, en la que además de encapsular las cadenas hemos definido los operadadores =, ==, !=, >, <, >=, <=, algunos de los cuales son necesarios para poder crear listas con esta clase.

Clase para manejar cadenas:

```
// CCadena.h: Fichero de cabecera de definición de cadenas
// C con Clase: Marzo de 2002

#ifndef CCADENA
#define CCADENA
#include <cstring>
using namespace std;

class Cadena {
  public:
   Cadena(char *cad) {
      cadena = new char[strlen(cad)+1];
      strcpy(cadena, cad);
   }
   Cadena() : cadena(NULL) {}
   Cadena(const Cadena &c) : cadena(NULL) {*this = c;}
   ~Cadena() { if(cadena) delete[] cadena; }
   Cadena &operator=(const Cadena &c) {
      if(this != &c) {
         if(cadena) delete[] cadena;
         if(c.cadena) {
            cadena = new char[strlen(c.cadena)+1];
            strcpy(cadena, c.cadena);
         }
         else cadena = NULL;
      }
      return *this;
   }
   bool operator==(const Cadena &c) const {
      return !strcmp(cadena, c.cadena);
   }
   bool operator!=(const Cadena &c) const {
      return strcmp(cadena, c.cadena);
   }
```

```
    bool operator<(const Cadena &c) const {
       return strcmp(cadena, c.cadena) < 0;
    }
    bool operator>(const Cadena &c) const {
       return strcmp(cadena, c.cadena) > 0;
    }
    bool operator<=(const Cadena &c) const {
       return strcmp(cadena, c.cadena) <= 0;
    }
    bool operator>=(const Cadena &c) const {
       return strcmp(cadena, c.cadena) >= 0;
    }

    const char* Lee() const {return cadena;}

  private:
    char *cadena;
};

ostream& operator<<(ostream &os, const Cadena& cad) {
   os << cad.Lee();
   return os;
}
#endif
```

Ahora ya podemos poner algunos ejemplos de listas creadas con plantillas:

```
// ListaAb.cpp: Ejemplo de uso de plantilla para listas
abiertas
// C con Clase: Marzo de 2002

#include <iostream>
#include "CCadena.h"
#include "ListaAb.h"
using namespace std;

// Plantilla de función que incrementa el valor del objeto
// dado como parámetro aplicando el operador ++
template<class DATO>
void Incrementar(DATO &d) {
   d++;
}
```

```
// Plantilla de función que mustra el valor del objeto
// dado como parámetro formando una lista separada con comas
template<class DATO>
void Mostrar(DATO &d) {
   cout << d << ", ";
}

int main() {
   // Declaración de una lista de enteros:
   Lista<int> ListaInt;

   // Inserción de algunos valores:
   ListaInt.InsertarFinal(43);
   ListaInt.InsertarFinal(65);
   ListaInt.InsertarFinal(33);
   ListaInt.InsertarFinal(64);
   ListaInt.InsertarFinal(22);
   ListaInt.InsertarFinal(11);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaInt.ParaCada(Mostrar); // Aplicamos la función
Mostrar a cada elemento
   cout << endl << "-------------" << endl;

   // Incrementamos los valores de todos los elementos de la
lista
   // aplicando a cada uno la función "Incrementar":
   cout << "---Incrementar todos---" << endl;
   ListaInt.ParaCada(Incrementar);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaInt.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;
   cin.get();

   // Borrar el primer elemento de valor 34:
   cout << "borrar 34" << endl;
   ListaInt.BorrarPrimerValor(34);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaInt.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;

   // Declaración de una lista de floats:
   Lista<float> ListaFloat;
```

```
   // Inserción de algunos valores:
   ListaFloat.InsertarFinal(43.2);
   ListaFloat.InsertarFinal(65.3);
   ListaFloat.InsertarFinal(33.1);
   ListaFloat.InsertarFinal(64.8);
   ListaFloat.InsertarFinal(22.32);
   ListaFloat.InsertarFinal(11.003);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaFloat.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;

   // Incrementamos todos:
   cout << "---Incrementar todos---" << endl;
   ListaFloat.ParaCada(Incrementar);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaFloat.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;

   cin.get();

   // Declaración de una lista de cadenas:
   Lista<Cadena> ListaCad;

   // Inserción de algunos valores, creando una lista
ordenada:
   ListaCad.Insertar("alfa");
   ListaCad.Insertar("delta");
   ListaCad.Insertar("beta");
   ListaCad.Insertar("gamma");
   ListaCad.Insertar("delta");
   ListaCad.Insertar("epsilon");
   ListaCad.Insertar("sigma");
   ListaCad.Insertar("delta");

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaCad.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;
   cin.get();

   // Borramos todos los elementos de valor "delta":
   while(ListaCad.BorrarPrimerValor("delta")) {
      cout << "borrar 'delta'" << endl;
```

```
      // Mostrar lista:
      cout << "---listado---" << endl;
      ListaCad.ParaCada(Mostrar);
      cout << endl << "-------------" << endl;
      cin.get();
   }

   // Buscar el primer elemento de valor "gamma":
   cout << "buscar 'gamma'" << endl;
   if(ListaCad.BuscarPrimerValor("gamma"))
      cout << ListaCad.ValorActual() << endl;
   else
      cout << "No encontrado" << endl;

   // Declaración de una lista de enteros:
   Lista<int> ListaOrden;

   // Inserción de algunos valores, creando una lista
ordenada:
   cout << "Lista ordenada de enteros" << endl;
   ListaOrden.Insertar(43);
   ListaOrden.Insertar(65);
   ListaOrden.Insertar(33);
   ListaOrden.Insertar(64);
   ListaOrden.Insertar(4);
   ListaOrden.Insertar(22);
   ListaOrden.Insertar(1);
   ListaOrden.Insertar(11);
   ListaOrden.Insertar(164);

   // Mostrar lista:
   cout << "---listado---" << endl;
   ListaOrden.ParaCada(Mostrar);
   cout << endl << "-------------" << endl;

   cin.get();
   return 0;
}
```

Hemos creado dos plantillas de funciones para demostrar el uso de la función "ParaCada", una de ellas incrementa cada valor de la lista, la otra lo muestra por pantalla. La primera no puede aplicarse a listas de cadenas, porque el operador ++ no está definido en la clase Cadena.

El resto creo que no necesita mucha explicación.

Fichero con el código fuente.

Capítulo 2 Pilas

2.1 Definición

Una pila es un tipo especial de lista abierta en la que sólo se pueden insertar y eliminar nodos en uno de los extremos de la lista. Estas operaciones se conocen como "push" y "pop", respectivamente "empujar" y "tirar". Además, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo leído.

Estas características implican un comportamiento de lista LIFO (Last In First Out), el último en entrar es el primero en salir.

El símil del que deriva el nombre de la estructura es una pila de platos. Sólo es posible añadir platos en la parte superior de la pila, y sólo pueden tomarse del mismo extremo.

El nodo típico para construir pilas es el mismo que vimos en el capítulo anterior para la construcción de listas:

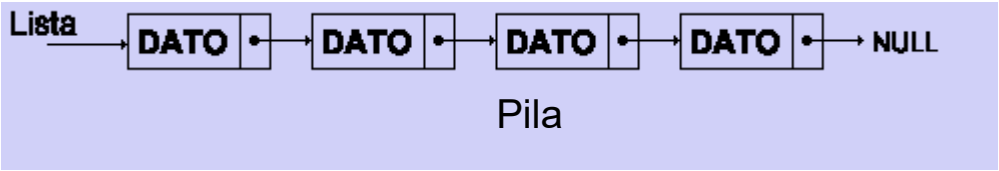
```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

2.2 Declaraciones de tipos para manejar pilas en C

Los tipos que definiremos normalmente para manejar pilas serán casi los mismos que para manejar listas, tan sólo cambiaremos algunos nombres:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Pila;
```

tipoNodo es el tipo para declarar nodos, evidentemente.
pNodo es el tipo para declarar punteros a un nodo.
Pila es el tipo para declarar pilas.



Pila

Es evidente, a la vista del gráfico, que una pila es una lista abierta. Así que sigue siendo muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, igual que pasa con las listas abiertas.

Teniendo en cuenta que las inserciones y borrados en una pila se hacen siempre en un extremo, lo que consideramos como el primer elemento de la lista es en realidad el último elemento de la pila.

2.3 Operaciones básicas con pilas

Las pilas tienen un conjunto de operaciones muy limitado, sólo permiten las operaciones de "push" y "pop":

- Push: Añadir un elemento al final de la pila.
- Pop: Leer y eliminar un elemento del final de la pila.

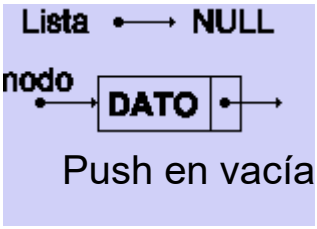
2.4 Push, insertar elemento

Las operaciones con pilas son muy simples, no hay casos especiales, salvo que la pila esté vacía.

Push en una pila vacía

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero a la pila valdrá NULL:

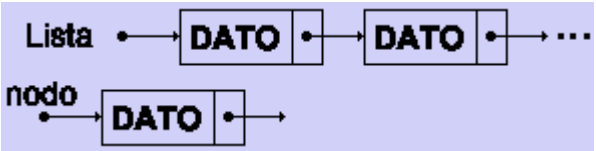
El proceso es muy simple, bastará con que:



Push en vacía

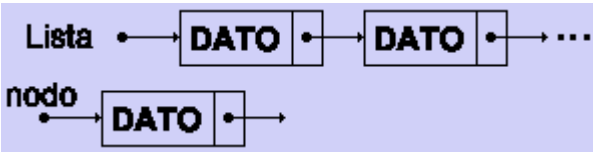
1. nodo->siguiente apunte a NULL.
2. Pilaa apunte a nodo.

Push en una pila no vacía



Push en pila no vacía

Podemos considerar el caso anterior como un caso particular de éste, la única diferencia es que podemos y debemos trabajar con una pila vacía como con una pila normal.



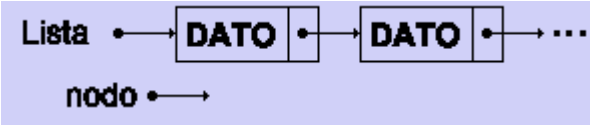
Resultado

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una pila, en este caso no vacía:

El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a Pila.
2. Hacemos que Pila apunte a nodo.

2.5 Pop, leer y eliminar un elemento

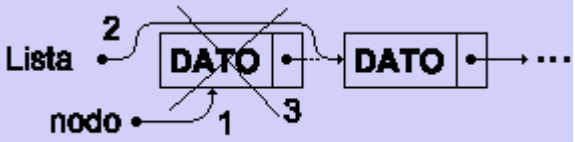


Ahora sólo existe un caso posible, ya que sólo podemos

Pop

leer desde un extremo de la pila.

Partiremos de una pila con uno o más nodos, y usaremos un puntero auxiliar, nodo:



Resultado

1. Hacemos que nodo apunte al primer elemento de la pila, es decir a Pila.
2. Asignamos a Pila la dirección del segundo nodo de la pila: Pila->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación pop equivale a leer y borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

Si la pila sólo tiene un nodo, el proceso sigue siendo válido, ya que el valor de Pila->siguiente es NULL, y después de eliminar el último nodo la pila quedará vacía, y el valor de Pila será NULL.

2.6 Ejemplo de pila en C

Supongamos que queremos construir una pila para almacenar números enteros. Haremos pruebas intercalando varios "push" y "pop", y comprobando el resultado.

Algoritmo de la función "push"

1. Creamos un nodo para el valor que colocaremos en la pila.
2. Hacemos que nodo->siguiente apunte a Pila.
3. Hacemos que Pila apunte a nodo.

```
void Push(Pila *pila, int v) {
   pNodo nuevo;

   /* Crear un nodo nuevo */
```

```
    nuevo = (pNodo)malloc(sizeof(tipoNodo));
    nuevo->valor = v;

    /* Añadimos la pila a continuación del nuevo nodo */
    nuevo->siguiente = *pila;
    /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
    *pila = nuevo;
}
```

Algoritmo de la función "pop"

1. Hacemos que nodo apunte al primer elemento de la pila, es decir a Pila.
2. Asignamos a Pila la dirección del segundo nodo de la pila: Pila->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación pop equivale a leer y borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

```
int Pop(Pila *pila) {
    pNodo nodo; /* variable auxiliar para manipular nodo */
    int v;      /* variable auxiliar para retorno */

    /* Nodo apunta al primer elemento de la pila */
    nodo = *pila;
    if(!nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
    /* Asignamos a pila toda la pila menos el primer elemento
*/
    *pila = nodo->siguiente;
    /* Guardamos el valor de retorno */
    v = nodo->valor;
    /* Borrar el nodo */
    free(nodo);
    return v;
}
```

Código del ejemplo completo

```c
#include <stdlib.h>
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Pila;

/* Funciones con pilas: */
void Push(Pila *l, int v);
int Pop(Pila *l);

int main() {
   Pila pila = NULL;

   Push(&pila, 20);
   Push(&pila, 10);
   printf("%d, ", Pop(&pila));
   Push(&pila, 40);
   Push(&pila, 30);

   printf("%d, ", Pop(&pila));
   printf("%d, ", Pop(&pila));
   Push(&pila, 90);
   printf("%d, ", Pop(&pila));
   printf("%d\n", Pop(&pila));

   getchar();
   return 0;
}

void Push(Pila *pila, int v) {
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Añadimos la pila a continuación del nuevo nodo */
   nuevo->siguiente = *pila;
   /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
   *pila = nuevo;
}
```

```
int Pop(Pila *pila) {
   pNodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = *pila;
   if(!nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Asignamos a pila toda la pila menos el primer elemento
*/
   *pila = nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = nodo->valor;
   /* Borrar el nodo */
   free(nodo);
   return v;
}
```

Fichero con el código fuente

2.7 Ejemplo de pila en C++ usando clases

Al igual que pasaba con las listas, usando clases el programa cambia bastante. Las clases para pilas son versiones simplificadas de las mismas clases que usamos para listas.

Para empezar, necesitaremos dos clases, una para nodo y otra para pila. Además la clase para nodo debe ser amiga de la clase pila, ya que ésta debe acceder a los miembros privados de nodo.

```
class nodo {
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;
```

```
   friend class pila;
};

typedef nodo *pnodo;

class pila {
   public:
    pila() : ultimo(NULL) {}
    ~pila();

    void Push(int v);
    int Pop();

   private:
    pnodo ultimo;
};
```

Los algoritmos para Push y Pop son los mismos que expusimos para el ejemplo C, tan sólo cambia el modo de crear y destruir nodos.

Código del ejemplo completo

```
#include <iostream>
using namespace std;

class nodo {
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class pila;
};

typedef nodo *pnodo;
```

```
class pila {
   public:
    pila() : ultimo(NULL) {}
    ~pila();

    void Push(int v);
    int Pop();

   private:
    pnodo ultimo;
};

pila::~pila() {
   while(ultimo) Pop();
}

void pila::Push(int v) {
   pnodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = new nodo(v, ultimo);
   /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
   ultimo = nuevo;
}

int pila::Pop() {

   pnodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   if(!ultimo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Nodo apunta al primer elemento de la pila */
   nodo = ultimo;
   /* Asignamos a pila toda la pila menos el primer elemento
*/
   ultimo = nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = nodo->valor;
   /* Borrar el nodo */
   delete nodo;
   return v;
}

int main() {
   pila Pila;

   Pila.Push(20);
```

```
    cout << "Push(20)" << endl;
    Pila.Push(10);
    cout << "Push(10)" << endl;
    cout << "Pop() = " << Pila.Pop() << endl;
    Pila.Push(40);
    cout << "Push(40)" << endl;
    Pila.Push(30);
    cout << "Push(30)" << endl;
    cout << "Pop() = " << Pila.Pop() << endl;
    cout << "Pop() = " << Pila.Pop() << endl;
    Pila.Push(90);
    cout << "Push(90)" << endl;
    cout << "Pop() = " << Pila.Pop() << endl;
    cout << "Pop() = " << Pila.Pop() << endl;

    return 0;
}
```

Fichero con el código fuente

2.8 Ejemplo de pila en C++ usando plantillas

Veremos ahora un ejemplo sencillo usando plantillas. Ya que la estructura para pilas es más sencilla que para listas abiertas, nuestro ejemplo también será más simple.

Seguimos necesitando dos clases, una para nodo y otra para pila. Pero ahora podremos usar esas clases para construir listas de cualquier tipo de datos.

Código del un ejemplo completo

Veremos primero las declaraciones de las dos clases que necesitamos:

```
template<class TIPO> class pila;

template<class TIPO>
class nodo {
   public:
```

```
      nodo(TIPO v, nodo<TIPO> *sig = NULL) {
         valor = v;
         siguiente = sig;
      }

   private:
      TIPO valor;
      nodo<TIPO> *siguiente;

      friend class pila<TIPO>;
};

template<class TIPO>
class pila {
   public:
      pila() : ultimo(NULL) {}
      ~pila();

      void Push(TIPO v);
      TIPO Pop();

   private:
      nodo<TIPO> *ultimo;
};
```

La implementación de las funciones es la misma que para el ejemplo de la página anterior.

```
template<class TIPO>
pila<TIPO>::~pila() {
   while(ultimo) Pop();
}

template<class TIPO>
void pila<TIPO>::Push(TIPO v) {
   nodo<TIPO> *nuevo;

   /* Crear un nodo nuevo */
   nuevo = new nodo<TIPO>(v, ultimo);
   /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
   ultimo = nuevo;
}

template<class TIPO>
```

```
TIPO pila<TIPO>::Pop() {
    nodo<TIPO> *Nodo; /* variable auxiliar para manipular
nodo */
    TIPO v;      /* variable auxiliar para retorno */

    if(!ultimo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
    /* Nodo apunta al primer elemento de la pila */
    Nodo = ultimo;
    /* Asignamos a pila toda la pila menos el primer
elemento */
    ultimo = Nodo->siguiente;
    /* Guardamos el valor de retorno */
    v = Nodo->valor;
    /* Borrar el nodo */
    delete Nodo;
    return v;
}
```

Eso es todo, ya sólo falta usar nuestras clases para un ejemplo práctico:

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

template<class TIPO> class pila;

template<class TIPO>
class nodo {
   public:
      nodo(TIPO v, nodo<TIPO> *sig = NULL) {
         valor = v;
         siguiente = sig;
      }

   private:
      TIPO valor;
      nodo<TIPO> *siguiente;

      friend class pila<TIPO>;
};

template<class TIPO>
```

```
class pila {
   public:
      pila() : ultimo(NULL) {}
      ~pila();

      void Push(TIPO v);
      TIPO Pop();

   private:
      nodo<TIPO> *ultimo;
};

template<class TIPO>
pila<TIPO>::~pila() {
   while(ultimo) Pop();
}

template<class TIPO>
void pila<TIPO>::Push(TIPO v) {
   nodo<TIPO> *nuevo;

   /* Crear un nodo nuevo */
   nuevo = new nodo<TIPO>(v, ultimo);
   /* Ahora, el comienzo de nuestra pila es en nuevo nodo */
   ultimo = nuevo;
}

template<class TIPO>
TIPO pila<TIPO>::Pop() {
    nodo<TIPO> *Nodo; /* variable auxiliar para manipular
nodo */
    TIPO v;      /* variable auxiliar para retorno */

    if(!ultimo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
    /* Nodo apunta al primer elemento de la pila */
    Nodo = ultimo;
    /* Asignamos a pila toda la pila menos el primer
elemento */
    ultimo = Nodo->siguiente;
    /* Guardamos el valor de retorno */
    v = Nodo->valor;
    /* Borrar el nodo */
    delete Nodo;
    return v;
}

int main() {
```

```
    pila<int> iPila;
    pila<float> fPila;
    pila<double> dPila;
    pila<char> cPila;
    pila<Cadena> sPila;

    // Prueba con <int>
    iPila.Push(20);
    iPila.Push(10);
    cout << iPila.Pop() << ",";
    iPila.Push(40);
    iPila.Push(30);

    cout << iPila.Pop() << ",";
    cout << iPila.Pop() << ",";
    iPila.Push(90);
    cout << iPila.Pop() << ",";
    cout << iPila.Pop() << endl;


    // Prueba con <float>
    fPila.Push(20.01);
    fPila.Push(10.02);
    cout << fPila.Pop() << ",";
    fPila.Push(40.03);
    fPila.Push(30.04);

    cout << fPila.Pop() << ",";
    cout << fPila.Pop() << ",";
    fPila.Push(90.05);
    cout << fPila.Pop() << ",";
    cout << fPila.Pop() << endl;

    // Prueba con <double>
    dPila.Push(0.0020);
    dPila.Push(0.0010);
    cout << dPila.Pop() << ",";
    dPila.Push(0.0040);
    dPila.Push(0.0030);

    cout << dPila.Pop() << ",";
    cout << dPila.Pop() << ",";
    dPila.Push(0.0090);
    cout << dPila.Pop() << ",";
    cout << dPila.Pop() << endl;

    // Prueba con <Cadena>
    cPila.Push('x');
```

```
    cPila.Push('y');
    cout << cPila.Pop() << ",";
    cPila.Push('a');
    cPila.Push('b');

    cout << cPila.Pop() << ",";
    cout << cPila.Pop() << ",";
    cPila.Push('m');
    cout << cPila.Pop() << ",";
    cout << cPila.Pop() << endl;

    // Prueba con <char *>
    sPila.Push("Hola");
    sPila.Push("somos");
    cout << sPila.Pop() << ",";
    sPila.Push("programadores");
    sPila.Push("buenos");

    cout << sPila.Pop() << ",";
    cout << sPila.Pop() << ",";
    sPila.Push("!!!!");
    cout << sPila.Pop() << ",";
    cout << sPila.Pop() << endl;

    cin.get();
    return 0;
}
```

Fichero con el código fuente

Capítulo 3 Colas

3.1 Definición

Una cola es un tipo especial de lista abierta en la que sólo se pueden insertar nodos en uno de los extremos de la lista y sólo se pueden eliminar nodos en el otro. Además, como sucede con las pilas, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo leído.

Este tipo de lista es conocido como lista FIFO (First In First Out), el primero en entrar es el primero en salir.

El símil cotidiano es una cola para comprar, por ejemplo, las entradas del cine. Los nuevos compradores sólo pueden colocarse al final de la cola, y sólo el primero de la cola puede comprar la entrada.

El nodo típico para construir pilas es el mismo que vimos en los capítulos anteriores para la construcción de listas y pilas:

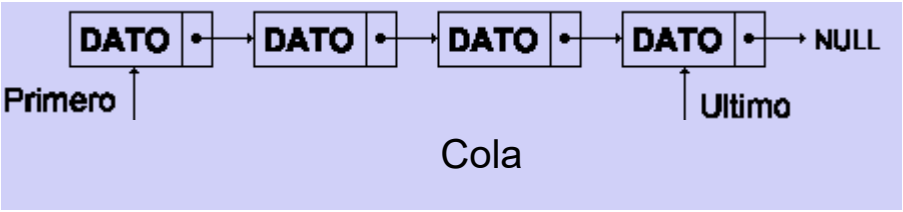
```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

3.2 Declaraciones de tipos para manejar colas en C

Los tipos que definiremos normalmente para manejar colas serán casi los mismos que para manejar listas y pilas, tan sólo cambiaremos algunos nombres:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Cola;
```

tipoNodo es el tipo para declarar nodos, evidentemente.
pNodo es el tipo para declarar punteros a un nodo.
Cola es el tipo para declarar colas.



Cola

Es evidente, a la vista del gráfico, que una cola es una lista abierta. Así que sigue siendo muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, igual que pasa con las listas abiertas. Además, debido al funcionamiento de las colas, también deberemos mantener un puntero para el último elemento de la cola, que será el punto donde insertemos nuevos nodos.

Teniendo en cuenta que las lecturas y escrituras en una cola se hacen siempre en extremos distintos, lo más fácil será insertar nodos por el final, a continuación del nodo que no tiene nodo siguiente, y leerlos desde el principio, hay que recordar que leer un nodo implica eliminarlo de la cola.

3.3 Operaciones básicas con colas

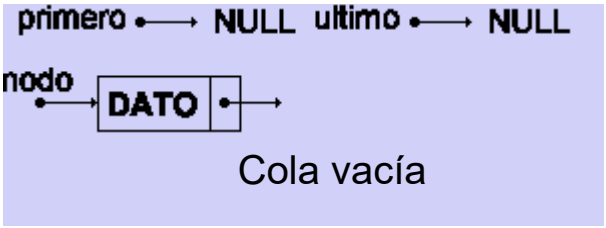
De nuevo nos encontramos ante una estructura con muy pocas operaciones disponibles. Las colas sólo permiten añadir y leer elementos:

- Añadir: Inserta un elemento al final de la cola.
- Leer: Lee y elimina un elemento del principio de la cola.

3.4 Añadir un elemento

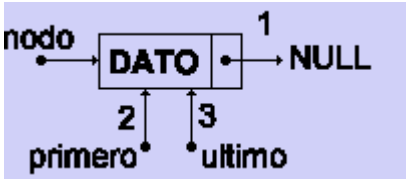
Las operaciones con colas son muy sencillas, prácticamente no hay casos especiales, salvo que la cola esté vacía.

Añadir elemento en una cola vacía



Cola vacía

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además los punteros que definen la cola, primero y ultimo que valdrán NULL:

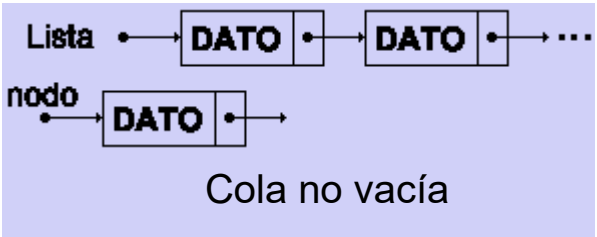


Elemento encolado

El proceso es muy simple, bastará con que:

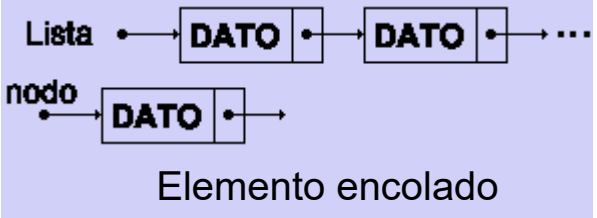
1. Hacer que nodo->siguiente apunte a NULL.
2. Que el puntero primero apunte a nodo.
3. Y que el puntero último también apunte a nodo.

Añadir elemento en una cola no vacía



Cola no vacía

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una cola, en este caso, al no estar vacía, los punteros primero y ultimo no serán nulos:



Elemento encolado

El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a NULL.
2. Después que ultimo->siguiente apunte a nodo.
3. Y actualizamos ultimo, haciendo que apunte a nodo.

Añadir elemento en una cola, caso general

Para generalizar el caso anterior, sólo necesitamos añadir una operación:

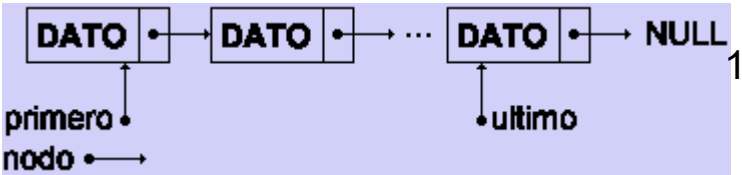
1. Hacemos que nodo->siguiente apunte a NULL.
2. Si ultimo no es NULL, hacemos que ultimo->siguiente apunte a nodo.
3. Y actualizamos ultimo, haciendo que apunte a nodo.
4. Si primero es NULL, significa que la cola estaba vacía, así que haremos que primero apunte también a nodo.

3.5 Leer un elemento de una cola, implica eliminarlo

Ahora también existen dos casos, que la cola tenga un solo elemento o que tenga más de uno.

Leer un elemento en una cola con más de un elemento

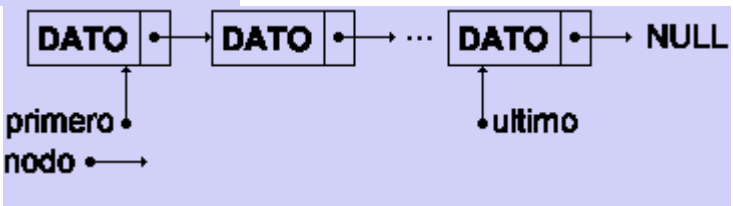
Usaremos un puntero a un nodo auxiliar:



Cola con más de un elemento

1. Hacemos que nodo apunte al primer elemento de la cola, es decir a primero.

2. Asignamos a primero la dirección del segundo nodo

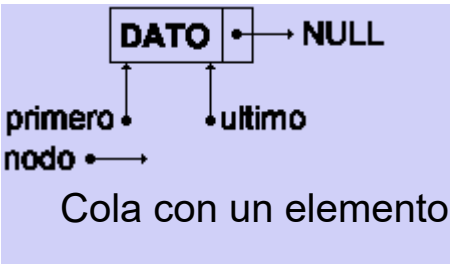


de la pila: primero->siguiente.

Elemento desencolado

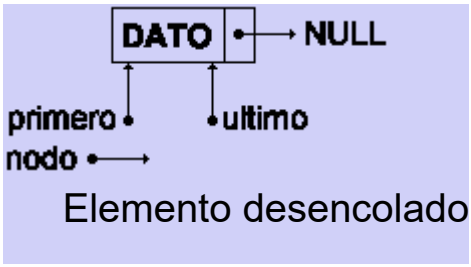
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación de lectura en colas implican también borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

Leer un elemento en una cola con un solo elemento



Cola con un elemento

También necesitamos un puntero a un nodo auxiliar:



Elemento desencolado

1. Hacemos que nodo apunte al primer elemento de la pila, es decir a primero.
2. Asignamos NULL a primero, que es la dirección del segundo nodo teórico de la cola: primero->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación de lectura en colas implican también borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.
5. Hacemos que ultimo apunte a NULL, ya que la lectura ha dejado la cola vacía.

Leer un elemento en una cola caso general

1. Hacemos que nodo apunte al primer elemento de la pila, es decir a primero.
2. Asignamos a primero la dirección del segundo nodo de la pila: primero->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación de lectura en colas implican también borrar.

4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.
5. Si primero es NULL, hacemos que ultimo también apunte a NULL, ya que la lectura ha dejado la cola vacía.

3.6 Ejemplo de cola en C

Construiremos una cola para almacenar números enteros. Haremos pruebas insertando varios valores y leyéndolos alternativamente para comprobar el resultado.

Algoritmo de la función "Anadir"

1. Creamos un nodo para el valor que colocaremos en la cola.
2. Hacemos que nodo->siguiente apunte a NULL.
3. Si "ultimo" no es NULL, hacemos que ultimo->siguiente apunte a nodo.
4. Actualizamos "ultimo" haciendo que apunte a nodo.
5. Si "primero" es NULL, hacemos que apunte a nodo.

```
void Anadir(pNodo *primero, pNodo *ultimo, int v) {
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;
   /* Este será el último nodo, no debe tener siguiente */
   nuevo->siguiente = NULL;
   /* Si la cola no estaba vacía, añadimos el nuevo a
continuación de ultimo */
   if(*ultimo) (*ultimo)->siguiente = nuevo;
   /* Ahora, el último elemento de la cola es el nuevo nodo
*/
   *ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora
primero apuntará también al nuevo nodo */
   if(!*primero) *primero = nuevo;
}
```

Algoritmo de la función "leer"

1. Hacemos que nodo apunte al primer elemento de la cola, es decir a primero.
2. Asignamos a primero la dirección del segundo nodo de la cola: primero->siguiente.
3. Guardamos el contenido del nodo para devolverlo como retorno, recuerda que la operación de lectura equivale a leer y borrar.
4. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.
5. Si primero es NULL, haremos que último también apunte a NULL, ya que la cola habrá quedado vacía.

```
int Leer(pNodo *primero, pNodo *ultimo) {
   pNodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = *primero;
   if(!nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Asignamos a primero la dirección del segundo nodo */
   *primero = nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = nodo->valor;
   /* Borrar el nodo */
   free(nodo);
   /* Si la cola quedó vacía, ultimo debe ser NULL también*/
   if(!*primero) *ultimo = NULL;
   return v;
}
```

Código del ejemplo completo

Tan sólo nos queda escribir una pequeña prueba para verificar el funcionamiento de las colas:

```
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;

/* Funciones con colas: */
void Anadir(pNodo *primero, pNodo *ultimo, int v);
int Leer(pNodo *primero, pNodo *ultimo);

int main() {
   pNodo primero = NULL, ultimo = NULL;

   Anadir(&primero, &ultimo, 20);
   printf("Añadir(20)\n");
   Anadir(&primero, &ultimo, 10);
   printf("Añadir(10)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   Anadir(&primero, &ultimo, 40);
   printf("Añadir(40)\n");
   Anadir(&primero, &ultimo, 30);
   printf("Añadir(30)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   Anadir(&primero, &ultimo, 90);
   printf("Añadir(90)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   printf("Leer: %d\n", Leer(&primero, &ultimo));

   return 0;
}

void Anadir(pNodo *primero, pNodo *ultimo, int v) {
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;
   /* Este será el último nodo, no debe tener siguiente */
   nuevo->siguiente = NULL;
   /* Si la cola no estaba vacía, añadimos el nuevo a
continuación de ultimo */
   if(*ultimo) (*ultimo)->siguiente = nuevo;
```

```
   /* Ahora, el último elemento de la cola es el nuevo nodo
*/
   *ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora
primero apuntará también al nuevo nodo */
   if(!*primero) *primero = nuevo;
}

int Leer(pNodo *primero, pNodo *ultimo) {
   pNodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = *primero;
   if(!nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Asignamos a primero la dirección del segundo nodo */
   *primero = nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = nodo->valor;
   /* Borrar el nodo */
   free(nodo);
   /* Si la cola quedó vacía, ultimo debe ser NULL también*/
   if(!*primero) *ultimo = NULL;
   return v;
}
```

Fichero con el código fuente

3.7 Ejemplo de cola en C++ usando clases

Ya hemos visto que las colas son casos particulares de listas abiertas, pero más simples. Como en los casos anteriores, veremos ahora un ejemplo de cola usando clases.

Para empezar, y como siempre, necesitaremos dos clases, una para nodo y otra para cola. Además la clase para nodo debe ser amiga de la clase cola, ya que ésta debe acceder a los miembros privados de nodo.

```
class nodo {
```

```
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class cola;
};

typedef nodo *pnodo;

class cola {
   public:
    cola() : ultimo(NULL), primero(NULL) {}
    ~cola();

    void Anadir(int v);
    int Leer();

   private:
    pnodo primero, ultimo;
};
```

Los algoritmos para Anadir y Leer son los mismos que expusimos para el ejemplo C, tan sólo cambia el modo de crear y destruir nodos.

Código del ejemplo completo

```
#include <iostream>
using namespace std;

class nodo {
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }
```

```
   private:
    int valor;
    nodo *siguiente;

   friend class cola;
};

typedef nodo *pnodo;

class cola {
   public:
    cola() : ultimo(NULL), primero(NULL) {}
    ~cola();

    void Push(int v);
    int Pop();

   private:
    pnodo ultimo;
};

cola::~cola() {
   while(primero) Leer();
}

void cola::Anadir(int v) {
   pnodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = new nodo(v);
   /* Si la cola no estaba vacía, añadimos el nuevo a
continuación de ultimo */
   if(ultimo) ultimo->siguiente = nuevo;
   /* Ahora, el último elemento de la cola es el nuevo nodo
*/
   ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora
primero apuntará también al nuevo nodo */
   if(!primero) primero = nuevo;
}

int cola::Leer() {

   pnodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = primero;
```

```
    if(!nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
    /* Asignamos a primero la dirección del segundo nodo */
    primero = nodo->siguiente;
    /* Guardamos el valor de retorno */
    v = nodo->valor;
    /* Borrar el nodo */
    delete nodo;
    /* Si la cola quedó vacía, ultimo debe ser NULL también*/
    if(!primero) ultimo = NULL;
    return v;
}

int main() {
    cola Cola;

    Cola.Anadir(20);
    cout << "Añadir(20)" << endl;
    Cola.Anadir(10);
    cout << "Añadir(10)" << endl;
    cout << "Leer: " << Cola.Leer() << endl;
    Cola.Anadir(40);
    cout << "Añadir(40)" << endl;
    Cola.Anadir(30);
    cout << "Añadir(30)" << endl;
    cout << "Leer: " << Cola.Leer() << endl;
    cout << "Leer: " << Cola.Leer() << endl;
    Cola.Anadir(90);
    cout << "Añadir(90)" << endl;
    cout << "Leer: " << Cola.Leer() << endl;
    cout << "Leer: " << Cola.Leer() << endl;

    return 0;
}
```

Fichero con el código fuente

3.8 Ejemplo de cola en C++ usando plantillas

Veremos ahora un ejemplo sencillo usando plantillas. Ya que la estructura para colas es más sencilla que para listas abiertas, nuestro ejemplo también será más simple.

Seguimos necesitando dos clases, una para nodo y otra para cola. Pero ahora podremos usar esas clases para construir listas de cualquier tipo de datos.

Código del un ejemplo completo

Veremos primero las declaraciones de las dos clases que necesitamos:

```
template<class TIPO> class cola;

template<class TIPO>
class nodo {
   public:
      nodo(TIPO v, nodo<TIPO> *sig = NULL) {
         valor = v;
         siguiente = sig;
      }

   private:
      TIPO valor;
      nodo<TIPO> *siguiente;

      friend class cola<TIPO>;
};

template<class TIPO>
class cola {
   public:
      cola() : primero(NULL), ultimo(NULL) {}
      ~cola();

      void Anadir(TIPO v);
      TIPO Leer();

   private:
      nodo<TIPO> *primero, *ultimo;
};
```

La implementación de las funciones es la misma que para el ejemplo de la página anterior.

```
template<class TIPO>
cola<TIPO>::~cola() {
   while(primero) Leer();
}

template<class TIPO>
void cola<TIPO>::Anadir(TIPO v) {
   nodo<TIPO> *nuevo;

   /* Crear un nodo nuevo */
   /* Este será el último nodo, no debe tener siguiente */
   nuevo = new nodo<tipo>(v);
   /* Si la cola no estaba vacía, añadimos el nuevo a
continuación de ultimo */
   if(ultimo) ultimo->siguiente = nuevo;
   /* Ahora, el último elemento de la cola es el nuevo nodo
*/
   ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora
primero apuntará también al nuevo nodo */
   if(!primero) primero = nuevo;
}

template<class TIPO>
TIPO cola<TIPO>::Leer() {
    nodo<TIPO> *Nodo; /* variable auxiliar para manipular
nodo */
    TIPO v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   Nodo = primero;
   if(!Nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Asignamos a primero la dirección del segundo nodo */
   primero = Nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = Nodo->valor;
   /* Borrar el nodo */
   delete Nodo;
   /* Si la cola quedó vacía, ultimo debe ser NULL también*/
   if(!primero) ultimo = NULL;
   return v;
}
```

Eso es todo, ya sólo falta usar nuestras clases para un ejemplo práctico:

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

template<class TIPO> class cola;

template<class TIPO>
class nodo {
   public:
      nodo(TIPO v, nodo<TIPO> *sig = NULL) {
         valor = v;
         siguiente = sig;
      }

   private:
      TIPO valor;
      nodo<TIPO> *siguiente;

      friend class cola<TIPO>;
};

template<class TIPO>
class cola {
   public:
      cola() : primero(NULL), ultimo(NULL) {}
      ~cola();

      void Anadir(TIPO v);
      TIPO Leer();

   private:
      nodo<TIPO> *primero, *ultimo;
};

template<class TIPO>
cola<TIPO>::~cola() {
   while(primero) Leer();
}

template<class TIPO>
void cola<TIPO>::Anadir(TIPO v) {
   nodo<TIPO> *nuevo;
```

```
   /* Crear un nodo nuevo */
   /* Este será el último nodo, no debe tener siguiente */
   nuevo = new nodo<tipo>(v);
   /* Si la cola no estaba vacía, añadimos el nuevo a
continuación de ultimo */
   if(ultimo) ultimo->siguiente = nuevo;
   /* Ahora, el último elemento de la cola es el nuevo nodo
*/
   ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora
primero apuntará también al nuevo nodo */
   if(!primero) primero = nuevo;
}

template<class TIPO>
TIPO cola<TIPO>::Leer() {
    nodo<TIPO> *Nodo; /* variable auxiliar para manipular
nodo */
    TIPO v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   Nodo = primero;
   if(!Nodo) return 0; /* Si no hay nodos en la pila
retornamos 0 */
   /* Asignamos a primero la dirección del segundo nodo */
   primero = Nodo->siguiente;
   /* Guardamos el valor de retorno */
   v = Nodo->valor;
   /* Borrar el nodo */
   delete Nodo;
   /* Si la cola quedó vacía, ultimo debe ser NULL también*/
   if(!primero) ultimo = NULL;
   return v;
}

int main() {
   cola <int> iCola;
   cola <float> fCola;
   cola <double> dCola;
   cola <char> cCola;
   cola <Cadena> sCola;

   // Prueba con <int>
   iCola.Anadir(20);
   cout << "Añadir(20)" << endl;
   iCola.Anadir(10);
   cout << "Añadir(10)" << endl;
```

```
    cout << "Leer: " << iCola.Leer() << endl;
    iCola.Anadir(40);
    cout << "Añadir(40)" << endl;
    iCola.Anadir(30);
    cout << "Añadir(30)" << endl;
    cout << "Leer: " << iCola.Leer() << endl;
    cout << "Leer: " << iCola.Leer() << endl;
    iCola.Anadir(90);
    cout << "Añadir(90)" << endl;
    cout << "Leer: " << iCola.Leer() << endl;
    cout << "Leer: " << iCola.Leer() << endl;

    // Prueba con <float>
    fCola.Anadir(20.01);
    cout << "Añadir(20.01)" << endl;
    fCola.Anadir(10.02);
    cout << "Añadir(10.02)" << endl;
    cout << "Leer: " << fCola.Leer() << endl;
    fCola.Anadir(40.03);
    cout << "Añadir(40.03)" << endl;
    fCola.Anadir(30.04);
    cout << "Añadir(30.04)" << endl;
    cout << "Leer: " << fCola.Leer() << endl;
    cout << "Leer: " << fCola.Leer() << endl;
    fCola.Anadir(90.05);
    cout << "Añadir(90.05)" << endl;
    cout << "Leer: " << fCola.Leer() << endl;
    cout << "Leer: " << fCola.Leer() << endl;

    // Prueba con <double>
    dCola.Anadir(0.0020);
    cout << "Añadir(0.0020)" << endl;
    dCola.Anadir(0.0010);
    cout << "Añadir(0.0010)" << endl;
    cout << "Leer: " << dCola.Leer() << endl;
    dCola.Anadir(0.0040);
    cout << "Añadir(0.0040)" << endl;
    dCola.Anadir(0.0030);
    cout << "Añadir(0.0030)" << endl;
    cout << "Leer: " << dCola.Leer() << endl;
    cout << "Leer: " << dCola.Leer() << endl;
    dCola.Anadir(0.0090);
    cout << "Añadir(0.0090)" << endl;
    cout << "Leer: " << dCola.Leer() << endl;
    cout << "Leer: " << dCola.Leer() << endl;

    // Prueba con <char>
    cCola.Anadir('x');
```

```
    cout << "Añadir(\'x\')" << endl;
    cCola.Anadir('y');
    cout << "Añadir(\'y\')" << endl;
    cout << "Leer: " << cCola.Leer() << endl;
    cCola.Anadir('a');
    cout << "Añadir(\'a\')" << endl;
    cCola.Anadir('b');
    cout << "Añadir(\'b\')" << endl;
    cout << "Leer: " << cCola.Leer() << endl;
    cout << "Leer: " << cCola.Leer() << endl;
    cCola.Anadir('m');
    cout << "Añadir(\'m\')" << endl;
    cout << "Leer: " << cCola.Leer() << endl;
    cout << "Leer: " << cCola.Leer() << endl;

    // Prueba con <Cadena>
    sCola.Anadir("Hola");
    cout << "Añadir(\"Hola\")" << endl;
    sCola.Anadir("somos");
    cout << "Añadir(\"somos\")" << endl;
    cout << "Leer: " << sCola.Leer() << endl;
    sCola.Anadir("programadores");
    cout << "Añadir(\"programadores\")" << endl;
    sCola.Anadir("buenos");
    cout << "Añadir(\"buenos\")" << endl;
    cout << "Leer: " << sCola.Leer() << endl;
    cout << "Leer: " << sCola.Leer() << endl;
    sCola.Anadir("!!!!");
    cout << "Añadir(\"!!!!\")" << endl;
    cout << "Leer: " << sCola.Leer() << endl;
    cout << "Leer: " << sCola.Leer() << endl;

    return 0;
}
```

Fichero con el código fuente

Capítulo 4 Listas circulares

4.1 Definición

Una lista circular es una lista lineal en la que el último nodo a punta al primero.

Las listas circulares evitan excepciones en la operaciones que se realicen sobre ellas. No existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente.

En algunas listas circulares se añade un nodo especial de cabecera, de ese modo se evita la única excepción posible, la de que la lista esté vacía.

El nodo típico es el mismo que para construir listas abiertas:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

4.2 Declaraciones de tipos para manejar listas circulares en C

Los tipos que definiremos normalmente para manejar listas cerradas son los mismos que para para manejar listas abiertas:

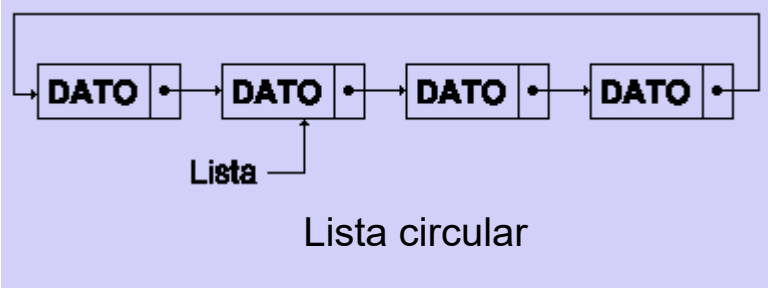
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;
```

```
typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

tipoNodo es el tipo para declarar nodos, evidentemente.

pNodo es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas, tanto abiertas como circulares. En el caso de las circulares, apuntará a un nodo cualquiera de la lista.



Lista circular

A pesar de que las listas circulares simplifiquen las operaciones sobre ellas, también introducen algunas complicaciones. Por ejemplo, en un proceso de búsqueda, no es tan sencillo dar por terminada la búsqueda cuando el elemento buscado no existe.

Por ese motivo se suele resaltar un nodo en particular, que no tiene por qué ser siempre el mismo. Cualquier nodo puede cumplir ese propósito, y puede variar durante la ejecución del programa.

Otra alternativa que se usa a menudo, y que simplifica en cierto modo el uso de listas circulares es crear un nodo especial de hará la función de nodo cabecera. De este modo, la lista nunca estará vacía, y se eliminan casi todos los casos especiales.

4.3 Operaciones básicas con listas circulares

A todos los efectos, las listas circulares son como las listas abiertas en cuanto a las operaciones que se pueden realizar sobre ellas:

- Añadir o insertar elementos.
- Buscar o localizar elementos.

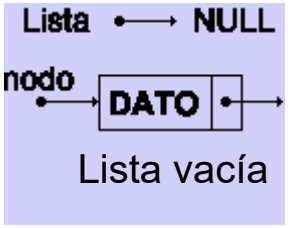
- Borrar elementos.
- Moverse a través de la lista, siguiente.

Cada una de éstas operaciones podrá tener varios casos especiales, por ejemplo, tendremos que tener en cuenta cuando se inserte un nodo en una lista vacía, o cuando se elimina el único nodo de una lista.

4.4 Añadir un elemento

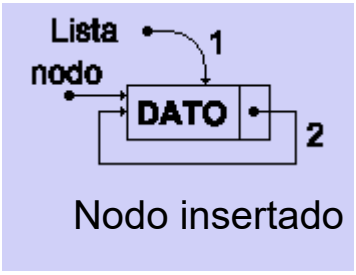
El único caso especial a la hora de insertar nodos en listas circulares es cuando la lista esté vacía.

Añadir elemento en una lista circular vacía



Lista vacía

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero que define la lista, que valdrá NULL:

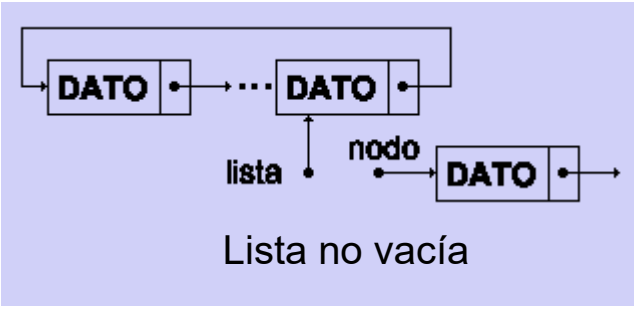


Nodo insertado

El proceso es muy simple, bastará con hacer que:

1. lista apunte a **nodo**.
2. y lista->siguiente apunte a **nodo**.

Añadir elemento en una lista circular no vacía

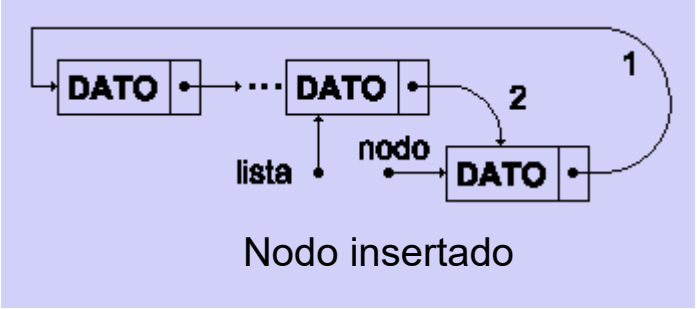


Lista no vacía

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una lista, en este caso, el puntero no será nulo:

El proceso sigue siendo muy sencillo:

1. Hacemos que **nodo**->siguiente apunte a lista->siguiente.
2. Después que lista->siguiente apunte a **nodo**.



Nodo insertado

Añadir elemento en una lista circular, caso general

Para generalizar los dos casos anteriores, sólo necesitamos añadir una operación:

1. Si lista está vacía hacemos que lista apunte a **nodo**.
2. Si lista no está vacía, hacemos que **nodo**->siguiente apunte a lista->siguiente.
3. Después que lista->siguiente apunte a **nodo**.

4.5 Buscar o localizar un elemento de una lista circular

A la hora de buscar elementos en una lista circular sólo hay que tener una precaución, es necesario almacenar el puntero del nodo en que se empezó la búsqueda, para poder detectar el caso en que no exista el valor que se busca. Por lo demás, la búsqueda es igual que en el caso de las listas abiertas, salvo que podemos empezar en cualquier punto de la lista.

4.6 Eliminar un elemento de una lista circular

Para ésta operación podemos encontrar tres casos diferentes:

1. Eliminar un nodo cualquiera, que no sea el apuntado por lista.
2. Eliminar el nodo apuntado por lista, y que no sea el único nodo.
3. Eliminar el único nodo de la lista.

En el primer caso necesitamos localizar el nodo anterior al que queremos borrar. Como el principio de la lista puede ser cualquier nodo, haremos que sea precisamente lista quien apunte al nodo anterior al que queremos eliminar.

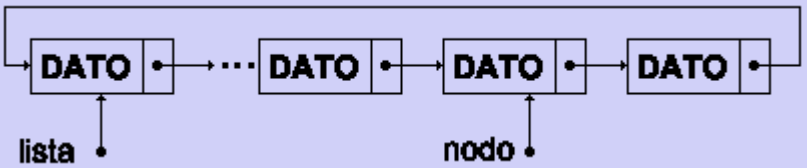
Esto elimina la excepción del segundo caso, ya que lista nunca será el nodo a eliminar, salvo que sea el único nodo de la lista.

Una vez localizado el nodo anterior y apuntado por lista, hacemos que lista->siguiente apunte a nodo->siguiente. Y a continuación borramos nodo.

En el caso de que sólo exista un nodo, será imposible localizar el nodo anterior, así que simplemente eliminaremos el nodo, y haremos que lista valga NULL.

Eliminar un nodo en una lista circular con más de un elemento

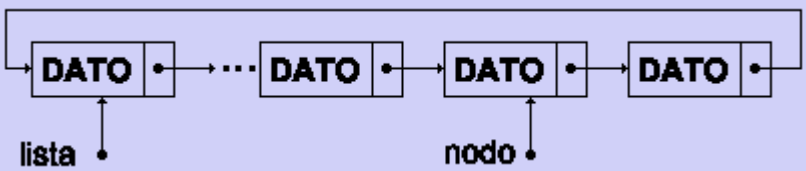
Consideraremos los dos primeros casos como uno sólo.



Lista no vacía

1. El primer paso es conseguir que lista apunte al nodo anterior al que queremos eliminar.

   Esto se consigue haciendo que lista valga lista->siguiente mientras lista->siguiente sea distinto de **nodo**.



Nodo eliminado

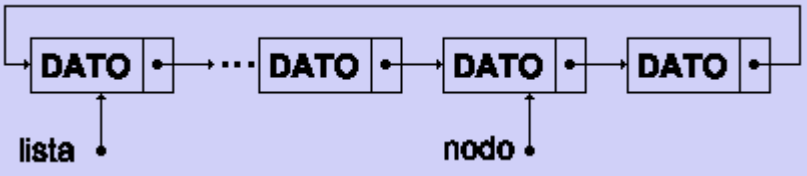
2. Hacemos que lista->siguiente apunte a **nodo**->siguiente.
3. Eliminamos el **nodo**.

Eliminar el único nodo en una lista circular

Este caso es mucho más sencillo. Si lista es el único nodo de una lista circular:

1. Borramos el nodo apuntado por lista.
2. Hacemos que lista valga NULL.

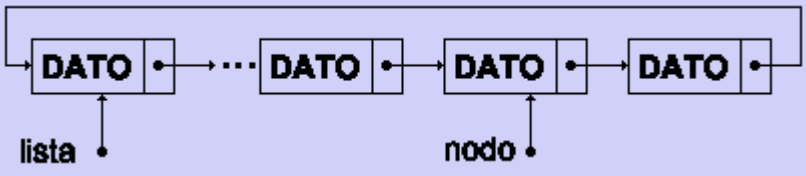
Otro algoritmo para borrar nodos



Lista no vacía

Existe un modo alternativo de eliminar un nodo en una lista circular con más de un nodo.

Supongamos que queremos eliminar un nodo apuntado por **nodo**:



Nodo eliminado

1. Copiamos el contenido del **nodo**->siguiente sobre el contenido de **nodo**.
2. Hacemos que **nodo**->siguiente apunte a **nodo**->siguiente->siguiente.
3. Eliminamos **nodo**->siguiente.
4. Si lista es el **nodo**->siguiente, hacemos lista = **nodo**.

Este método también funciona con listas circulares de un sólo elemento, salvo que el nodo a borrar es el único nodo que existe, y hay que hacer que lista apunte a NULL.

4.7 Ejemplo de lista circular en C

Construiremos una lista cerrada para almacenar números enteros. Haremos pruebas insertando varios valores, buscándolos y

eliminándolos alternativamente para comprobar el resultado.

Algoritmo de la función "Insertar"

1. Si lista está vacía hacemos que lista apunte a nodo.
2. Si lista no está vacía, hacemos que nodo->siguiente apunte a lista->siguiente.
3. Después que lista->siguiente apunte a nodo.

```
void Insertar(Lista *lista, int v) {
   pNodo nodo;

   // Creamos un nodo para el nuvo valor a insertar
   nodo = (pNodo)malloc(sizeof(tipoNodo));
   nodo->valor = v;

   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
del apuntado
   // por lista
   if(*lista == NULL) *lista = nodo;
   else nodo->siguiente = (*lista)->siguiente;
   // En cualquier caso, cerramos la lista circular
   (*lista)->siguiente = nodo;
}
```

Algoritmo de la función "Borrar"

1. ¿Tiene la lista un único nodo?
2. **SI:**
    1. Borrar el nodo lista.
    2. Hacer lista = NULL.
3. **NO:**
    1. Hacemos lista->siguiente = nodo->siguiente.
    2. Borramos nodo.

```
void Borrar(Lista *lista, int v) {
```

```
   pNodo nodo;

   nodo = *lista;

   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if((*lista)->siguiente->valor != v) *lista = (*lista)-
>siguiente;
   } while((*lista)->siguiente->valor != v && *lista !=
nodo);
   // Si existe un nodo con el valor v:
   if((*lista)->siguiente->valor == v) {
      // Y si la lista sólo tiene un nodo
      if(*lista == (*lista)->siguiente) {
         // Borrar toda la lista
         free(*lista);
         *lista = NULL;
      }
      else {
         // Si la lista tiene más de un nodo, borrar el nodo
de valor v
         nodo = (*lista)->siguiente;
         (*lista)->siguiente = nodo->siguiente;
         free(nodo);
      }
   }
}
```

Código del ejemplo completo

Tan sólo nos queda escribir una pequeña prueba para verificar el funcionamiento:

```
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

```
// Funciones con listas:
void Insertar(Lista *l, int v);
void Borrar(Lista *l, int v);
void BorrarLista(Lista *);
void MostrarLista(Lista l);

int main() {
   Lista lista = NULL;
   pNodo p;

   Insertar(&lista, 10);
   Insertar(&lista, 40);
   Insertar(&lista, 30);
   Insertar(&lista, 20);
   Insertar(&lista, 50);

   MostrarLista(lista);

   Borrar(&lista, 30);
   Borrar(&lista, 50);

   MostrarLista(lista);

   BorrarLista(&lista);
   return 0;
}

void Insertar(Lista *lista, int v) {
   pNodo nodo;

   // Creamos un nodo para el nuvo valor a insertar
   nodo = (pNodo)malloc(sizeof(tipoNodo));
   nodo->valor = v;

   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
del apuntado
   // por lista
   if(*lista == NULL) *lista = nodo;
   else nodo->siguiente = (*lista)->siguiente;
   // En cualquier caso, cerramos la lista circular
   (*lista)->siguiente = nodo;
}

void Borrar(Lista *lista, int v) {
   pNodo nodo;

   nodo = *lista;
```

```
   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if((*lista)->siguiente->valor != v) *lista = (*lista)-
>siguiente;
   } while((*lista)->siguiente->valor != v && *lista !=
nodo);
   // Si existe un nodo con el valor v:
   if((*lista)->siguiente->valor == v) {
      // Y si la lista sólo tiene un nodo
      if(*lista == (*lista)->siguiente) {
         // Borrar toda la lista
         free(*lista);
         *lista = NULL;
      }
      else {
         // Si la lista tiene más de un nodo, borrar el nodo
de valor v
         nodo = (*lista)->siguiente;
         (*lista)->siguiente = nodo->siguiente;
         free(nodo);
      }
   }
}

void BorrarLista(Lista *lista) {
   pNodo nodo;

   // Mientras la lista tenga más de un nodo
   while((*lista)->siguiente != *lista) {
      // Borrar el nodo siguiente al apuntado por lista
      nodo = (*lista)->siguiente;
      (*lista)->siguiente = nodo->siguiente;
      free(nodo);
   }
   // Y borrar el último nodo
   free(*lista);
   *lista = NULL;
}

void MostrarLista(Lista lista) {
   pNodo nodo = lista;

   do {
      printf("%d -> ", nodo->valor);
      nodo = nodo->siguiente;
   } while(nodo != lista);
```

```
   printf("\n");
}
```

Fichero con el código fuente

4.8 Ejemplo de lista circular en C++ usando clases

Para empezar, y como siempre, necesitaremos dos clases, una para nodo y otra para lista. Además la clase para nodo debe ser amiga de la clase lista, ya que ésta debe acceder a los miembros privados de nodo.

```
class nodo {
   public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    int valor;
    nodo *siguiente;

   friend class lista;
};

typedef nodo *pnodo;

class lista {
   public:
    lista() { actual = NULL; }
    ~lista();

    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return actual == NULL; }
    void Mostrar();
    void Siguiente();
    bool Actual() { return actual != NULL; }
```

```
    int ValorActual() { return actual->valor; }

  private:
    pnodo actual;
};
```

Hemos hecho que la clase para lista sea algo más completa que la equivalente en C, aprovechando las prestaciones de las clases.

Los algoritmos para insertar y borrar elementos son los mismos que expusimos para el ejemplo C, tan sólo cambia el modo de crear y destruir nodos.

Código del ejemplo completo

```
#include <iostream>
using namespace std;

class nodo {
  public:
    nodo(int v, nodo *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

  private:
    int valor;
    nodo *siguiente;

  friend class lista;
};

typedef nodo *pnodo;

class lista {
  public:
    lista() { actual = NULL; }
    ~lista();

    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return actual == NULL; }
    void Mostrar();
```

```
     void Siguiente();
     bool Actual() { return actual != NULL; }
     int ValorActual() { return actual->valor; }

   private:
     pnodo actual;
};

lista::~lista() {
   pnodo nodo;

   // Mientras la lista tenga más de un nodo
   while(actual->siguiente != actual) {
      // Borrar el nodo siguiente al apuntado por lista
      nodo = actual->siguiente;
      actual->siguiente = nodo->siguiente;
      delete nodo;
   }
   // Y borrar el último nodo
   delete actual;
   actual = NULL;
}

void lista::Insertar(int v) {
   pnodo Nodo;

   // Creamos un nodo para el nuevo valor a insertar
   Nodo = new nodo(v);

   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
del apuntado
   // por lista
   if(actual == NULL) actual = Nodo;
   else Nodo->siguiente = actual->siguiente;
   // En cualquier caso, cerramos la lista circular
   actual->siguiente = Nodo;
}

void lista::Borrar(int v) {
   pnodo nodo;

   nodo = actual;

   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if(actual->siguiente->valor != v) actual = actual-
>siguiente;
```

```
    } while(actual->siguiente->valor != v && actual != nodo);
    // Si existe un nodo con el valor v:
    if(actual->siguiente->valor == v) {
       // Y si la lista sólo tiene un nodo
       if(actual == actual->siguiente) {
          // Borrar toda la lista
          delete actual;
          actual = NULL;
       }
       else {
          // Si la lista tiene más de un nodo, borrar el nodo
de valor v
          nodo = actual->siguiente;
          actual->siguiente = nodo->siguiente;
          delete nodo;
       }
    }
}

void lista::Mostrar() {
    pnodo nodo = actual;

    do {
       cout << nodo->valor << "-> ";
       nodo = nodo->siguiente;
    } while(nodo != actual);

    cout << endl;
}

void lista::Siguiente() {
    if(actual) actual = actual->siguiente;
}

int main() {
    lista Lista;

    Lista.Insertar(20);
    Lista.Insertar(10);
    Lista.Insertar(40);
    Lista.Insertar(30);
    Lista.Insertar(60);

    Lista.Mostrar();

    cout << "Lista de elementos:" << endl;
    Lista.Borrar(10);
    Lista.Borrar(30);
```

```
    Lista.Mostrar();

    cin.get();
    return 0;
```

Fichero con el código fuente

4.9 Ejemplo de lista circular en C++ usando plantillas

Veremos ahora un ejemplo sencillo usando plantillas.

Seguimos necesitando dos clases, una para nodo y otra para la lista circular. Pero ahora podremos aprovechar las características de las plantillas para crear listas circulares de cualquier tipo de objeto.

Código del un ejemplo completo

Veremos primero las declaraciones de las dos clases que necesitamos:

```
template<class TIPO> class lista;

template<class TIPO>
class nodo {
   public:
    nodo(TIPO v, nodo<TIPO> *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    TIPO valor;
    nodo<TIPO> *siguiente;

   friend class lista<TIPO>;
};
```

```
template<class TIPO>
class lista {
   public:
    lista() { actual = NULL; }
    ~lista();

    void Insertar(TIPO v);
    void Borrar(TIPO v);
    bool ListaVacia() { return actual == NULL; }
    void Mostrar();
    void Siguiente();
    bool Actual() { return actual != NULL; }
    TIPO ValorActual() { return actual->valor; }

   private:
    nodo<TIPO> *actual;
};
```

Ahora veremos la implementación de estas clases. No difiere demasiado de otros ejemplos.

```
template<class TIPO>
lista<TIPO>::~lista() {
   nodo<TIPO> *Nodo;

   // Mientras la lista tenga más de un nodo
   while(actual->siguiente != actual) {
      // Borrar el nodo siguiente al apuntado por lista
      Nodo = actual->siguiente;
      actual->siguiente = Nodo->siguiente;
      delete Nodo;
   }
   // Y borrar el último nodo
   delete actual;
   actual = NULL;
}

template<class TIPO>
void lista<TIPO>::Insertar(TIPO v) {
   nodo<TIPO> *Nodo;

   // Creamos un nodo para el nuevo valor a insertar
   Nodo = new nodo<TIPO>(v);
```

```
   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
del apuntado
   // por lista
   if(actual == NULL) actual = Nodo;
   else Nodo->siguiente = actual->siguiente;
   // En cualquier caso, cerramos la lista circular
   actual->siguiente = Nodo;
}

template<class TIPO>
void lista<TIPO>::Borrar(TIPO v) {
   nodo<TIPO> *Nodo;

   Nodo = actual;

   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if(actual->siguiente->valor != v) actual = actual-
>siguiente;
   } while(actual->siguiente->valor != v && actual != Nodo);
   // Si existe un nodo con el valor v:
   if(actual->siguiente->valor == v) {
      // Y si la lista sólo tiene un nodo
      if(actual == actual->siguiente) {
         // Borrar toda la lista
         delete actual;
         actual = NULL;
      }
      else {
         // Si la lista tiene más de un nodo, borrar el nodo
de valor v
         Nodo = actual->siguiente;
         actual->siguiente = Nodo->siguiente;
         delete Nodo;
      }
   }
}

template<class TIPO>
void lista<TIPO>::Mostrar() {
   nodo<TIPO> *Nodo = actual;

   do {
      cout << Nodo->valor << "-> ";
      Nodo = Nodo->siguiente;
   } while(Nodo != actual);
```

```
   cout << endl;
}

template<class TIPO>
void lista<TIPO>::Siguiente() {
   if(actual) actual = actual->siguiente;
}
```

Eso es todo, ya sólo falta usar nuestras clases para un ejemplo práctico:

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

template<class TIPO> class lista;

template<class TIPO>
class nodo {
   public:
    nodo(TIPO v, nodo<TIPO> *sig = NULL) {
       valor = v;
       siguiente = sig;
    }

   private:
    TIPO valor;
    nodo<TIPO> *siguiente;

   friend class lista<TIPO>;
};

template<class TIPO>
class lista {
   public:
    lista() { actual = NULL; }
    ~lista();

    void Insertar(TIPO v);
    void Borrar(TIPO v);
    bool ListaVacia() { return actual == NULL; }
    void Mostrar();
    void Siguiente();
    bool Actual() { return actual != NULL; }
```

```
    TIPO ValorActual() { return actual->valor; }

  private:
    nodo<TIPO> *actual;
};

template<class TIPO>
lista<TIPO>::~lista() {
   nodo<TIPO> *Nodo;

   // Mientras la lista tenga más de un nodo
   while(actual->siguiente != actual) {
      // Borrar el nodo siguiente al apuntado por lista
      Nodo = actual->siguiente;
      actual->siguiente = Nodo->siguiente;
      delete Nodo;
   }
   // Y borrar el último nodo
   delete actual;
   actual = NULL;
}

template<class TIPO>
void lista<TIPO>::Insertar(TIPO v) {
   nodo<TIPO> *Nodo;

   // Creamos un nodo para el nuevo valor a insertar
   Nodo = new nodo<TIPO>(v);

   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
del apuntado
   // por lista
   if(actual == NULL) actual = Nodo;
   else Nodo->siguiente = actual->siguiente;
   // En cualquier caso, cerramos la lista circular
   actual->siguiente = Nodo;
}

template<class TIPO>
void lista<TIPO>::Borrar(TIPO v) {
   nodo<TIPO> *Nodo;

   Nodo = actual;

   // Hacer que lista apunte al nodo anterior al de valor v
   do {
      if(actual->siguiente->valor != v) actual = actual-
```

```
>siguiente;
   } while(actual->siguiente->valor != v && actual != Nodo);
   // Si existe un nodo con el valor v:
   if(actual->siguiente->valor == v) {
      // Y si la lista sólo tiene un nodo
      if(actual == actual->siguiente) {
         // Borrar toda la lista
         delete actual;
         actual = NULL;
      }
      else {
         // Si la lista tiene más de un nodo, borrar el nodo
de valor v
         Nodo = actual->siguiente;
         actual->siguiente = Nodo->siguiente;
         delete Nodo;
      }
   }
}

template<class TIPO>
void lista<TIPO>::Mostrar() {
   nodo<TIPO> *Nodo = actual;

   do {
      cout << Nodo->valor << "-> ";
      Nodo = Nodo->siguiente;
   } while(Nodo != actual);

   cout << endl;
}

template<class TIPO>
void lista<TIPO>::Siguiente() {
   if(actual) actual = actual->siguiente;
}

int main() {
   lista<int> iLista;
   lista<float> fLista;
   lista<double> dLista;
   lista<char> cLista;
   lista<Cadena> sLista;

   // Prueba con <int>
   iLista.Insertar(20);
   iLista.Insertar(10);
   iLista.Insertar(40);
```

```
    iLista.Insertar(30);
    iLista.Insertar(60);

    iLista.Mostrar();

    cout << "Lista de elementos:" << endl;
    iLista.Borrar(10);
    iLista.Borrar(30);

    iLista.Mostrar();

    // Prueba con <float>
    fLista.Insertar(20.01);
    fLista.Insertar(10.02);
    fLista.Insertar(40.03);
    fLista.Insertar(30.04);
    fLista.Insertar(60.05);

    fLista.Mostrar();

    cout << "Lista de elementos:" << endl;
    fLista.Borrar(10.02);
    fLista.Borrar(30.04);

    fLista.Mostrar();

    // Prueba con <double>
    dLista.Insertar(0.0020);
    dLista.Insertar(0.0010);
    dLista.Insertar(0.0040);
    dLista.Insertar(0.0030);
    dLista.Insertar(0.0060);

    dLista.Mostrar();

    cout << "Lista de elementos:" << endl;
    dLista.Borrar(0.0010);
    dLista.Borrar(0.0030);

    dLista.Mostrar();

    // Prueba con <char>
    cLista.Insertar('x');
    cLista.Insertar('y');
    cLista.Insertar('a');
    cLista.Insertar('b');
    cLista.Insertar('m');
```

```
    cLista.Mostrar();

    cout << "Lista de elementos:" << endl;
    cLista.Borrar('y');
    cLista.Borrar('b');

    cLista.Mostrar();

    // Prueba con <Cadena>
    sLista.Insertar("Hola");
    sLista.Insertar("somos");
    sLista.Insertar("programadores");
    sLista.Insertar("buenos");
    sLista.Insertar("!!!!");

    sLista.Mostrar();

    cout << "Lista de elementos:" << endl;
    sLista.Borrar("somos");
    sLista.Borrar("buenos");

    sLista.Mostrar();

    cin.get();
    return 0;
}
```

Fichero con el código fuente

Capítulo 5 Listas doblemente enlazadas

5.1 Definición

Una lista doblemente enlazada es una lista lineal en la que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior.

Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos.

El nodo típico es el mismo que para construir las listas que hemos visto, salvo que tienen otro puntero al nodo anterior:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
   struct nodo *anterior;
};
```

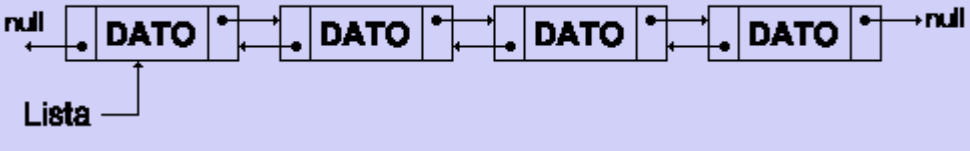
5.2 Declaraciones de tipos para manejar listas doblemente enlazadas en C

Para C, y basándonos en la declaración de nodo que hemos visto más arriba, trabajaremos con los siguientes tipos:

```
typedef struct _nodo {
   int dato;
```

```
    struct _nodo *siguiente;
    struct _nodo *anterior;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

tipoNodo es el tipo para declarar nodos, evidentemente.
pNodo es el tipo para declarar punteros a un nodo.



Lista doblemente enlazada

Lista es el tipo para declarar listas abiertas doblemente enlazadas. También es posible, y potencialmente útil, crear listas doblemente enlazadas y circulares.

El movimiento a través de listas doblemente enlazadas es más sencillo, y como veremos las operaciones de búsqueda, inserción y borrado, también tienen más ventajas.

5.3 Operaciones básicas con listas doblemente enlazadas

De nuevo tenemos el mismo repertorio de operaciones sobre este tipo listas:

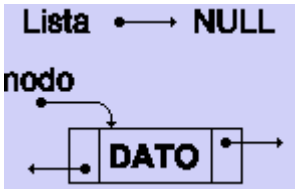
- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través de la lista, siguiente y anterior.

5.4 Añadir un elemento

Nos encontramos ahora ante un tipo de estructura algo diferente de las que hemos estado viendo, así que entraremos en más detalles.

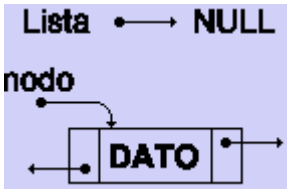
Vamos a intentar ver todos los casos posibles de inserción de elementos en listas doblemente enlazadas.

Añadir elemento en una lista doblemente enlazada vacía



Lista vacía

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero que define la lista, que valdrá NULL:

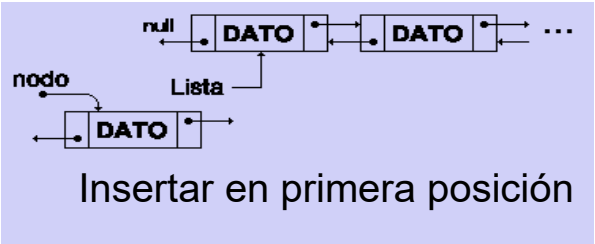


Lista de un elemento

El proceso es muy simple, bastará con que:

1. lista apunta a **nodo**.
2. lista->siguiente y lista->anterior apunten a null.

Insertar un elemento en la primera posición de la lista

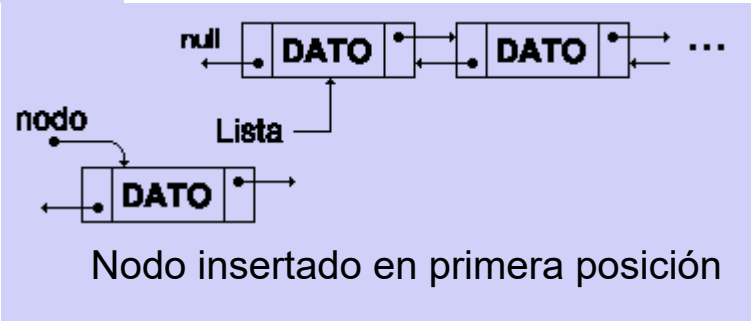


Insertar en primera posición

Partimos de una lista no vacía. Para simplificar, consideraremos que lista apunta al primer elemento de la lista doblemente enlazada:

El proceso es el siguiente:

1. **nodo**->siguiente debe apuntar a Lista.
2. **nodo**->anterior apuntará a Lista->anterior.

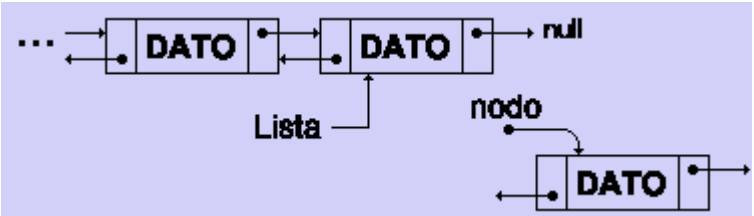


Nodo insertado en primera posición

3. Lista->anterior debe apuntar a **nodo**.

Recuerda que Lista no tiene por qué apuntar a ningún miembro concreto de una lista doblemente enlazada, cualquier miembro es igualmente válido como referencia.

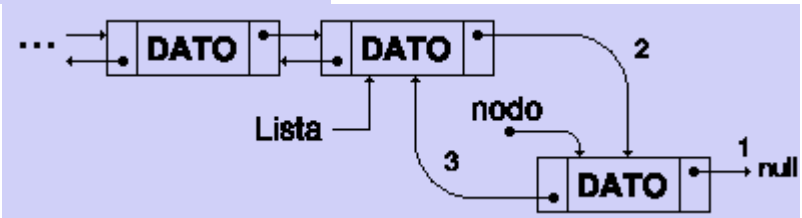
Insertar un elemento en la última posición de la lista



Insertar nodo al final

Igual que en el caso anterior, partiremos de una lista no vacía, y de nuevo para simplificar, que Lista está apuntando al último elemento de la lista:

El proceso es el siguiente:



Insertado al final

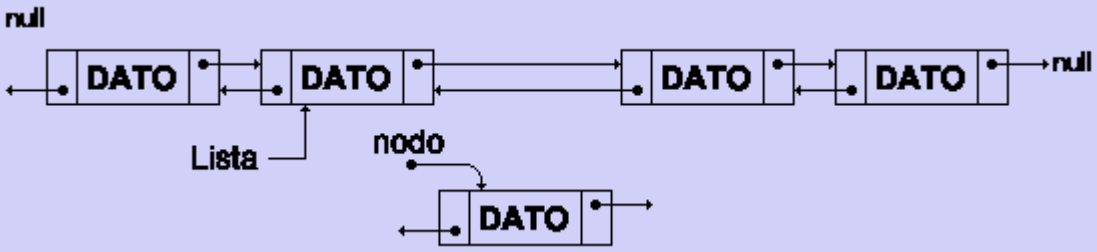
1. **nodo**->siguiente debe apuntar a Lista->siguiente (NULL).
2. Lista->siguiente debe apuntar a **nodo**.
3. **nodo**->anterior apuntará a Lista.

Insertar un elemento a continuación de un nodo cualquiera de una lista

Bien, este caso es más genérico, ahora partimos de una lista no vacía, e insertaremos un nodo a continuación de uno nodo cualquiera que no sea el último de la lista:

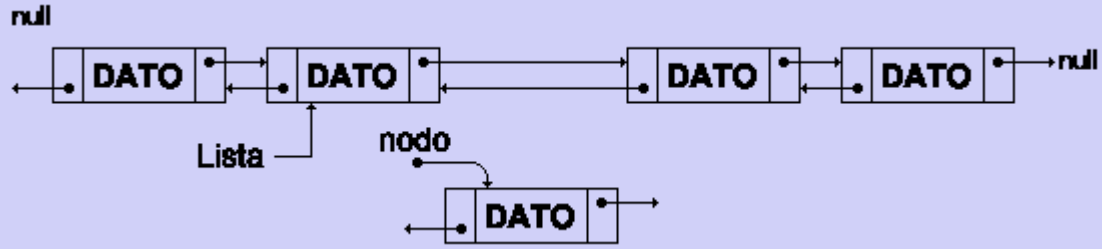
El proceso sigue siendo muy sencillo:

1. Hacemos que **nodo**->siguiente apunte a lista->siguiente.
2. Hacemos que Lista->siguiente apunte a **nodo**.



Insertar a continuación de un nodo

3. Hacemos que **nodo**->anterior apunte a lista.
4. Hacemos que **nodo**->siguiente->anterior apunte a **nodo**.



Nodo insertado después de un nodo dado

Lo que hemos hecho es trabajar como si tuviéramos dos listas enlazadas, los dos primeros pasos equivalen a lo que hacíamos para insertar elementos en una lista abierta corriente.

Los dos siguientes pasos hacen lo mismo con la lista que enlaza los nodos en sentido contrario.

El paso 4 es el más oscuro, quizás requiera alguna explicación.

Supongamos que disponemos de un puntero auxiliar, "p" y que antes de empezar a insertar nodo, hacemos que apunte al nodo que quedará a continuación de nodo después de insertarlo, es decir p = Lista->siguiente.

Ahora empezamos el proceso de inserción, ejecutamos los pasos 1, 2 y 3. El cuarto sería sólo hacer que p->anterior apunte a nodo. Pero nodo->siguiente ya apunta a p, así que en realidad no necesitamos el puntero auxiliar, bastará con hacer que nodo->siguiente->anterior apunte a nodo.

Añadir elemento en una lista doblemente enlazada, caso general

Para generalizar todos los casos anteriores, sólo necesitamos añadir una operación:

1. Si lista está vacía hacemos que Lista apunte a nodo. Y nodo->anterior y nodo->siguiente a NULL.
2. Si lista no está vacía, hacemos que nodo->siguiente apunte a Lista->siguiente.
3. Después que Lista->siguiente apunte a nodo.
4. Hacemos que nodo->anterior apunte a Lista.
5. Si nodo->siguiente no es NULL, entonces hacemos que nodo->siguiente->anterior apunte a nodo.

El paso 1 es equivalente a insertar un nodo en una lista vacía.

Los pasos 2 y 3 equivalen a la inserción en una lista enlazada corriente.

Los pasos 4, 5 equivalen a insertar en una lista que recorre los nodos en sentido contrario.

Existen más casos, las listas doblemente enlazadas son mucho más versátiles, pero todos los casos pueden reducirse a uno de los que hemos explicado aquí.

5.5 Buscar o localizar un elemento de una lista doblemente enlazada

En muchos aspectos, una lista doblemente enlazada se comporta como dos listas abiertas que comparten los datos. En ese sentido, todo lo dicho en el capítulo sobre la localización de elementos en listas abiertas se puede aplicar a listas doblemente enlazadas.

Pero además tenemos la ventaja de que podemos avanzar y retroceder desde cualquier nodo, sin necesidad de volver a uno de los extremos de la lista.

Por supuesto, se pueden hacer listas doblemente enlazadas no ordenadas, existen cientos de problemas que pueden requerir de este tipo de estructuras. Pero parece que la aplicación más sencilla de listas doblemente enlazadas es hacer arrays dinámicos ordenados, donde buscar un elemento concreto a partir de cualquier otro es más sencillo que en una lista abierta corriente.

Pero de todos modos veamos algún ejemplo sencillo.

Para recorrer una lista procederemos de un modo parecido al que usábamos con las listas abiertas, ahora no necesitamos un puntero auxiliar, pero tenemos que tener en cuenta que Lista no tiene por qué estar en uno de los extremos:

1. Retrocedemos hasta el comienzo de la lista, asignamos a lista el valor de lista->anterior mientras lista->anterior no sea NULL.
2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
3. Dentro del bucle asignaremos a lista el valor del nodo siguiente al actual.

Por ejemplo, para mostrar todos los valores de los nodos de una lista, podemos usar el siguiente bucle en C:

```
typedef struct _nodo {
  int dato;
  struct _nodo *siguiente;
  struct _nodo *anterior;
  } tipoNodo;

  typedef tipoNodo *pNodo;
  typedef tipoNodo *Lista;
  ...
  pNodo = indice;
  ...
  indice = Lista;
  while(indice->anterior) indice = indice->anterior;
  while(indice) {
     printf("%d\n", indice->dato);
     indice = indice->siguiente;
```

```
    }
    ...
```

Es importante que no perdamos el nodo Lista, si por error le asignáramos un valor de un puntero a un nodo que no esté en la lista, no podríamos acceder de nuevo a ella.

Es por eso que tendremos especial cuidado en no asignar el valor NULL a Lista.

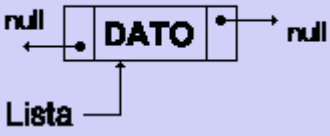
5.6 Eliminar un elemento de una lista doblemente enlazada

Analizaremos tres casos diferentes:

1. Eliminar el único nodo de una lista doblemente enlazada.
2. Eliminar el primer nodo.
3. Eliminar el último nodo.
4. Eliminar un nodo intermedio.

Para los casos que lo permitan consideraremos dos casos: que el nodo a eliminar es el actualmente apuntado por Lista o que no.

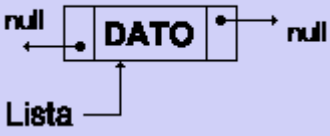
Eliminar el único nodo en una lista doblemente enlazada



Lista con un elemento

En este caso, ese nodo será el apuntado por Lista.

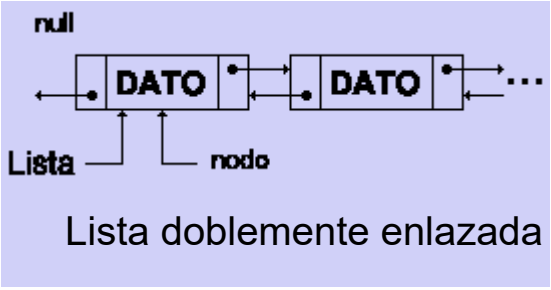
El proceso es simple:



Lista vacía

1. Eliminamos el **nodo**.
2. Hacemos que Lista apunte a NULL.

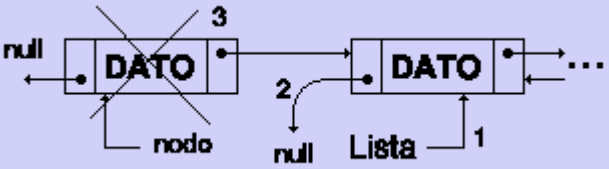
Eliminar el primer nodo de una lista doblemente enlazada



Lista doblemente enlazada

Tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->siguiente.

1. Si **nodo** apunta a Lista, hacemos que Lista apunte a Lista->siguiente.
2. Hacemos que **nodo**->siguiente->anterior apunte a NULL
3. Borramos el nodo apuntado por **nodo**.

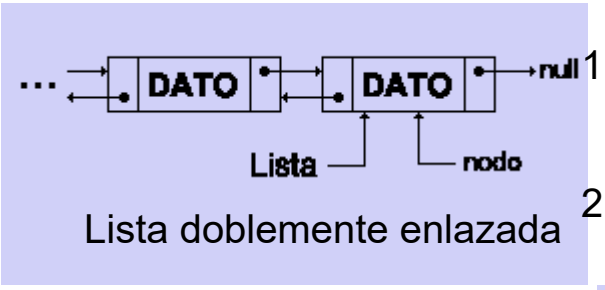


Elemento eliminado

El paso 2 separa el nodo a borrar del resto de la lista, independientemente del nodo al que apunte Lista.

Eliminar el último nodo de una lista doblemente enlazada

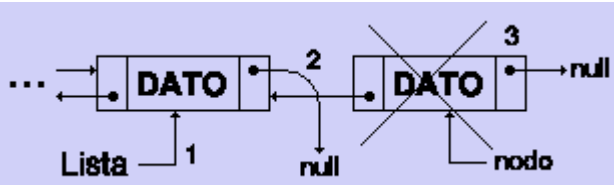
De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior.



Lista doblemente enlazada

1. Si **nodo** apunta a Lista, hacemos que Lista apunte a Lista->anterior.
2. Hacemos que **nodo**->anterior->siguiente apunte a NULL
3. Borramos el nodo apuntado por **nodo**.

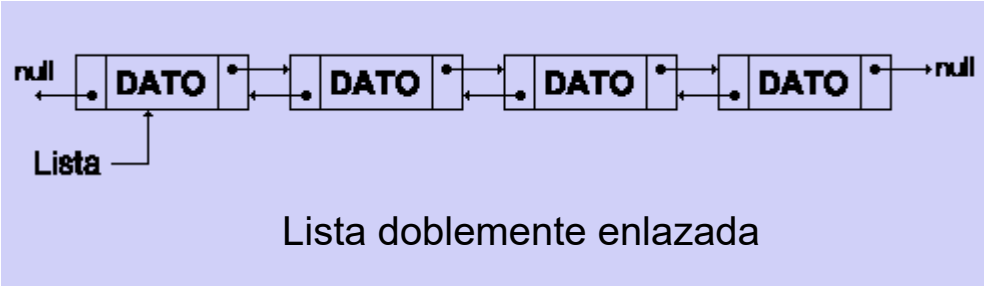
El paso 2 depara el nodo a borrar del resto de la lista,



Elemento eliminado

independientemente del nodo al que apunte Lista.

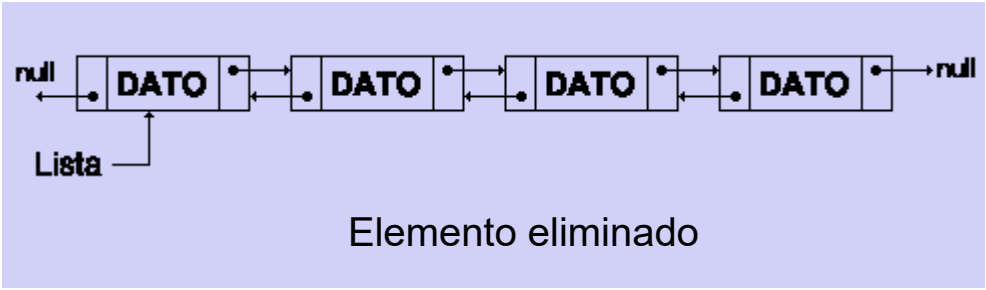
Eliminar un nodo intermedio de una lista doblemente enlazada



Lista doblemente enlazada

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior o Lista->siguiente

Se trata de un caso más general de los dos casos anteriores.



Elemento eliminado

1. Si **nodo** apunta a Lista, hacemos que Lista apunte a Lista->anterior (o Lista->siguiente).
2. Hacemos que **nodo**->anterior->siguiente apunte a **nodo**->siguiente.
3. Hacemos que **nodo**->siguiente->anterior apunte a **nodo**->anterior.
4. Borramos el nodo apuntado por nodo.

Eliminar un nodo de una lista doblemente enlazada, caso general

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos

que Lista sea Lista->anterior, si no es NULL o Lista->siguiente en caso contrario.

1. Si **nodo** apunta a Lista,
    - Si Lista->anterior no es NULL hacemos que Lista apunte a Lista->anterior.
    - Si Lista->siguiente no es NULL hacemos que Lista apunte a Lista->siguiente.
    - Si ambos son NULL, hacemos que Lista sea NULL.
2. Si **nodo**->anterior no es NULL, hacemos que **nodo**->anterior->siguiente apunte a **nodo**->siguiente.
3. Si **nodo**->siguiente no es NULL, hacemos que **nodo**->siguiente->anterior apunte a **nodo**->anterior.
4. Borramos el nodo apuntado por **nodo**.

5.7 Ejemplo de lista doblemente enlazada en C

Como en el caso de los ejemplos anteriores, construiremos una lista doblemente enlazada para almacenar números enteros. Para aprovechar mejor las posibilidades de estas listas, haremos que la lista esté ordenada. Haremos pruebas insertando varios valores, buscándolos y eliminándolos alternativamente para comprobar el resultado.

Algoritmo de inserción

1. El primer paso es crear un nodo para el dato que vamos a insertar.
2. Si Lista está vacía, o el valor del primer elemento de la lista es mayor que el del nuevo, insertaremos el nuevo nodo en la primera posición de la lista.
3. En caso contrario, buscaremos el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior->siguiente no

sea NULL y el dato que contiene anterior->siguiente sea menor o igual que el dato que queremos insertar.

4. Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

```
void Insertar(Lista *lista, int v) {
   pNodo nuevo, actual;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Colocamos actual en la primera posición de la lista */
   actual = *lista;
   if(actual) while(actual->anterior) actual = actual-
>anterior;

   /* Si la lista está vacía o el primer miembro es mayor
que el nuevo */
   if(!actual || actual->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = actual;
      nuevo->anterior = NULL;
      if(actual) actual->anterior = nuevo;
      if(!*lista) *lista = nuevo;
   }
   else {
      /* Avanzamos hasta el último elemento o hasta que el
siguiente tenga
         un valor mayor que v */
      while(actual->siguiente && actual->siguiente->valor <=
v)
         actual = actual->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior
*/
      nuevo->siguiente = actual->siguiente;
      actual->siguiente = nuevo;
      nuevo->anterior = actual;
      if(nuevo->siguiente) nuevo->siguiente->anterior =
nuevo;
   }
}
```

Algoritmo de la función "Borrar"

1. Localizamos el nodo de valor v
2. ¿Existe?
    - **SI**:
        - ¿Es el nodo apuntado por lista?
            - **SI**: Hacer que lista apunte a otro sitio.
        - ¿Es el primer nodo de la lista?
            - **NO**: nodo->anterior->siguiente = nodo->siguiente
        - ¿Es el último nodo de la lista?
            - **NO**: nodo->siguiente->anterior = nodo->anterior
        - Borrar nodo

```
void Borrar(Lista *lista, int v) {
   pNodo nodo;

   /* Buscar el nodo de valor v */
   nodo = *lista;
   while(nodo && nodo->valor < v) nodo = nodo->siguiente;
   while(nodo && nodo->valor > v) nodo = nodo->anterior;

   /* El valor v no está en la lista */
   if(!nodo || nodo->valor != v) return;

   /* Borrar el nodo */
   /* Si lista apunta al nodo que queremos borrar, apuntar a
otro */
   if(nodo == *lista)
     if(nodo->anterior) *lista = nodo->anterior;
     else *lista = nodo->siguiente;

   if(nodo->anterior) /* no es el primer elemento */
      nodo->anterior->siguiente = nodo->siguiente;
   if(nodo->siguiente) /* no es el último nodo */
      nodo->siguiente->anterior = nodo->anterior;
   free(nodo);
}
```

Código del ejemplo completo

Tan sólo nos queda escribir una pequeña prueba para verificar el funcionamiento:

```
#include <stdio.h>

#define ASCENDENTE 1
#define DESCENDENTE 0

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
   struct _nodo *anterior;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;

/* Funciones con listas: */
void Insertar(Lista *l, int v);
void Borrar(Lista *l, int v);

void BorrarLista(Lista *);
void MostrarLista(Lista l, int orden);

int main() {
   Lista lista = NULL;
   pNodo p;

   Insertar(&lista, 20);
   Insertar(&lista, 10);
   Insertar(&lista, 40);
   Insertar(&lista, 30);

   MostrarLista(lista, ASCENDENTE);
   MostrarLista(lista, DESCENDENTE);

   Borrar(&lista, 10);
   Borrar(&lista, 15);
   Borrar(&lista, 45);
   Borrar(&lista, 30);

   MostrarLista(lista, ASCENDENTE);
   MostrarLista(lista, DESCENDENTE);

   BorrarLista(&lista);
```

```
   return 0;
}

void Insertar(Lista *lista, int v) {
   pNodo nuevo, actual;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Colocamos actual en la primera posición de la lista */
   actual = *lista;
   if(actual) while(actual->anterior) actual = actual-
>anterior;
   /* Si la lista está vacía o el primer miembro es mayor
que el nuevo */
   if(!actual || actual->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = actual;
      nuevo->anterior = NULL;
      if(actual) actual->anterior = nuevo;
      if(!*lista) *lista = nuevo;
   }
   else {
      /* Avanzamos hasta el último elemento o hasta que el
siguiente tenga
         un valor mayor que v */
      while(actual->siguiente &&actual->siguiente->valor <=
v)
         actual = actual->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior
*/
      nuevo->siguiente = actual->siguiente;
      actual->siguiente = nuevo;
      nuevo->anterior = actual;
      if(nuevo->siguiente) nuevo->siguiente->anterior =
nuevo;
   }
}

void Borrar(Lista *lista, int v) {
   pNodo nodo;

   /* Buscar el nodo de valor v */
   nodo = *lista;
   while(nodo && nodo->valor < v) nodo = nodo->siguiente;
   while(nodo && nodo->valor > v) nodo = nodo->anterior;
```

```
   /* El valor v no está en la lista */
   if(!nodo || nodo->valor != v) return;

   /* Borrar el nodo */
   /* Si lista apunta al nodo que queremos borrar, apuntar a
otro */
   if(nodo == *lista)
     if(nodo->anterior) *lista = nodo->anterior;
     else *lista = nodo->siguiente;

   if(nodo->anterior) /* no es el primer elemento */
      nodo->anterior->siguiente = nodo->siguiente;
   if(nodo->siguiente) /* no es el último nodo */
      nodo->siguiente->anterior = nodo->anterior;
   free(nodo);
}

void BorrarLista(Lista *lista) {
   pNodo nodo, actual;

   actual = *lista;
   while(actual->anterior) actual = actual->anterior;

   while(actual) {
      nodo = actual;
      actual = actual->siguiente;
      free(nodo);
   }
   *lista = NULL;
}

void MostrarLista(Lista lista, int orden) {
   pNodo nodo = lista;

   if(!lista) printf("Lista vacía");

   nodo = lista;
   if(orden == ASCENDENTE) {
      while(nodo->anterior) nodo = nodo->anterior;
      printf("Orden ascendente: ");
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->siguiente;
      }
   }
   else {
      while(nodo->siguiente) nodo = nodo->siguiente;
```

```
      printf("Orden descendente: ");
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->anterior;
      }
   }

   printf("\n");
}
```

Fichero con el código fuente

5.8 Ejemplo de lista doblemente enlazada en C++ usando clases

Veamos ahora el mismo ejemplo usando clases.

Para empezar, y como siempre, necesitaremos dos clases, una para nodo y otra para lista. Además la clase para nodo debe ser amiga de la clase lista, ya que ésta debe acceder a los miembros privados de nodo.

```
class nodo {
   public:
    nodo(int v, nodo *sig = NULL, nodo *ant = NULL) :
        valor(v), siguiente(sig), anterior(ant) {}

   private:
    int valor;
    nodo *siguiente;
    nodo *anterior;

   friend class lista;
};

typedef nodo *pnodo;

class lista {
   public:
    lista() : lista(NULL) {}
    ~lista();
```

```
    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return lista == NULL; }
    void Mostrar();
    void Siguiente();
    void Anterior();
    void Primero();
    void Ultimo();
    pnodo Actual() { return lista; }
    int ValorActual() { return lista->valor; }

  private:
    pnodo lista;
};
```

Ahora sólo necesitamos un puntero para referenciar la lista y movernos a través de ella. Hemos añadido funciones para avanzar y retroceder, moverse al primero o último nodo, obtener un puntero al nodo actual o su valor.

Los algoritmos para insertar y borrar elementos son los mismos que expusimos para el ejemplo C, tan sólo cambia el modo de crear y destruir nodos.

Código del ejemplo completo

```
#include <iostream>
using namespace std;

#define ASCENDENTE 1
#define DESCENDENTE 0

class nodo {
  public:
    nodo(int v, nodo *sig = NULL, nodo *ant = NULL) :
       valor(v), siguiente(sig), anterior(ant) {}

  private:
    int valor;
    nodo *siguiente;
    nodo *anterior;
```

```
   friend class lista;
};

typedef nodo *pnodo;

class lista {
   public:
    lista() : plista(NULL) {}
    ~lista();

    void Insertar(int v);
    void Borrar(int v);
    bool ListaVacia() { return plista == NULL; }
    void Mostrar(int);
    void Siguiente();
    void Anterior();
    void Primero();
    void Ultimo();
    bool Actual() { return plista != NULL; }
    int ValorActual() { return plista-&gt;valor; }

   private:
    pnodo plista;
};

lista::~lista() {
   pnodo aux;

   Primero();
   while(plista) {
      aux = plista;
      plista = plista->siguiente;
      delete aux;
   }
}

void lista::Insertar(int v) {
   pnodo nuevo;

   Primero();
   // Si la lista está vacía
   if(ListaVacia() || plista->valor > v) {
      // Asignamos a lista un nuevo nodo de valor v y
      // cuyo siguiente elemento es la lista actual
      nuevo = new nodo(v, plista);
      if(!plista) plista = nuevo;
      else plista->anterior = nuevo;
```

```
    }
    else {
       // Buscar el nodo de valor menor a v
       // Avanzamos hasta el último elemento o hasta que el
siguiente tenga
       // un valor mayor que v
       while(plista->siguiente && plista->siguiente->valor <=
v) Siguiente();
       // Creamos un nuevo nodo después del nodo actual
       nuevo = new nodo(v, plista->siguiente, plista);
       plista->siguiente = nuevo;
       if(nuevo->siguiente) nuevo->siguiente->anterior =
nuevo;
    }
}

void lista::Borrar(int v) {
   pnodo nodo;

   nodo = plista;
   while(nodo && nodo->valor < v) nodo = nodo->siguiente;
   while(nodo && nodo->valor > v) nodo = nodo->anterior;

   if(!nodo || nodo->valor != v) return;
   // Borrar el nodo

   if(nodo->anterior) // no es el primer elemento
      nodo->anterior->siguiente = nodo->siguiente;
   if(nodo->siguiente) // no el el último nodo
      nodo->siguiente->anterior = nodo->anterior;
   delete nodo;
}

void lista::Mostrar(int orden) {
   pnodo nodo;
   if(orden == ASCENDENTE) {
      Primero();
      nodo = plista;
      while(nodo) {
         cout << nodo->valor << "-> ";
         nodo = nodo->siguiente;
      }
   }
   else {
      Ultimo();
      nodo = plista;
      while(nodo) {
         cout << nodo->valor << "-> ";
```

```
         nodo = nodo->anterior;
      }
   }
   cout << endl;
}

void lista::Siguiente() {
   if(plista) plista = plista->siguiente;
}

void lista::Anterior() {
   if(plista) plista = plista->anterior;
}

void lista::Primero() {
   while(plista && plista->anterior) plista = plista-
>anterior;
}

void lista::Ultimo() {
   while(plista && plista->siguiente) plista = plista-
>siguiente;
}

int main() {
   lista Lista;

   Lista.Insertar(20);
   Lista.Insertar(10);
   Lista.Insertar(40);
   Lista.Insertar(30);

   Lista.Mostrar(ASCENDENTE);
   Lista.Mostrar(DESCENDENTE);

   Lista.Primero();
   cout << "Primero: " << Lista.ValorActual() << endl;

   Lista.Ultimo();
   cout << "Ultimo: " << Lista.ValorActual() << endl;

   Lista.Borrar(10);
   Lista.Borrar(15);
   Lista.Borrar(45);
   Lista.Borrar(40);

   Lista.Mostrar(ASCENDENTE);
   Lista.Mostrar(DESCENDENTE);
```

```
   return 0;
}
```

Fichero con el código fuente

5.9 Ejemplo de lista doblemente enlazada en C++ usando plantillas

Veremos ahora un ejemplo sencillo usando plantillas.

Seguimos necesitando dos clases, una para nodo y otra para la lista circular. Pero ahora podremos aprovechar las características de las plantillas para crear listas circulares de cualquier tipo de objeto.

Código del un ejemplo completo

Veremos primero las declaraciones de las dos clases que necesitamos:

```
#define ASCENDENTE 1
#define DESCENDENTE 0

template<class TIPO> class lista;

template<class TIPO>
class nodo {
   public:
    nodo(TIPO v, nodo<TIPO> *sig = NULL, nodo<TIPO> *ant =
NULL) :
       valor(v), siguiente(sig), anterior(ant) {}

   private:
    TIPO valor;
    nodo<TIPO> *siguiente;
    nodo<TIPO> *anterior;

   friend class lista<TIPO>;
};
```

```
template<class TIPO>
class lista {
   public:
    lista() : plista(NULL) {}
    ~lista();

    void Insertar(TIPO v);
    void Borrar(TIPO v);
    bool ListaVacia() { return plista == NULL; }
    void Mostrar(int);
    void Siguiente();
    void Anterior();
    void Primero();
    void Ultimo();
    bool Actual() { return plista != NULL; }
    TIPO ValorActual() { return plista->valor; }

   private:
    nodo<TIPO> *plista;
};
```

Para resumir, veremos la implementación de estas clases junto con el código de un ejemplo de uso:

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

#define ASCENDENTE 1
#define DESCENDENTE 0

template<class TIPO> class lista;

template<class TIPO>
class nodo {
   public:
    nodo(TIPO v, nodo<TIPO> *sig = NULL, nodo<TIPO> *ant =
NULL) :
       valor(v), siguiente(sig), anterior(ant) {}

   private:
    TIPO valor;
    nodo<TIPO> *siguiente;
```

```
    nodo<TIPO> *anterior;

   friend class lista<TIPO>;
};

template<class TIPO>
class lista {
   public:
    lista() : plista(NULL) {}
    ~lista();

    void Insertar(TIPO v);
    void Borrar(TIPO v);
    bool ListaVacia() { return plista == NULL; }
    void Mostrar(int);
    void Siguiente();
    void Anterior();
    void Primero();
    void Ultimo();
    bool Actual() { return plista != NULL; }
    TIPO ValorActual() { return plista->valor; }

   private:
    nodo<TIPO> *plista;
};

template<class TIPO>
lista<TIPO>::~lista() {
   nodo<TIPO> *aux;

   Primero();
   while(plista) {
      aux = plista;
      plista = plista->siguiente;
      delete aux;
   }
}

template<class TIPO>
void lista<TIPO>::Insertar(TIPO v) {
   nodo<TIPO> *nuevo;

   Primero();
   // Si la lista está vacía
   if(ListaVacia() || plista->valor > v) {
      // Asignamos a lista un nuevo nodo de valor v y
      // cuyo siguiente elemento es la lista actual
      nuevo = new nodo<TIPO>(v, plista);
```

```
      if(!plista) plista = nuevo;
      else plista->anterior = nuevo;
   }
   else {
      // Buscar el nodo de valor menor a v
      // Avanzamos hasta el último elemento o hasta que el
siguiente tenga
      // un valor mayor que v
      while(plista->siguiente && plista->siguiente->valor <=
v) Siguiente();
      // Creamos un nuevo nodo después del nodo actual
      nuevo = new nodo<TIPO>(v, plista->siguiente, plista);
      plista->siguiente = nuevo;
      if(nuevo->siguiente) nuevo->siguiente->anterior =
nuevo;
   }
}

template<class TIPO>
void lista<TIPO>::Borrar(TIPO v) {
   nodo<TIPO> *pnodo;

   pnodo = plista;
   while(pnodo && pnodo->valor < v) pnodo = pnodo-
>siguiente;
   while(pnodo && pnodo->valor > v) pnodo = pnodo->anterior;

   if(!pnodo || pnodo->valor != v) return;
   // Borrar el nodo

   if(pnodo->anterior) // no es el primer elemento
      pnodo->anterior->siguiente = pnodo->siguiente;
   if(pnodo->siguiente) // no el el último nodo
      pnodo->siguiente->anterior = pnodo->anterior;
}

template<class TIPO>
void lista<TIPO>::Mostrar(int orden) {
   nodo<TIPO> *pnodo;
   if(orden == ASCENDENTE) {
      Primero();
      pnodo = plista;
      while(pnodo) {
         cout << pnodo->valor << "-> ";
         pnodo = pnodo->siguiente;
      }
   }
   else {
```

```
      Ultimo();
      pnodo = plista;
      while(pnodo) {
         cout << pnodo->valor << "-> ";
         pnodo = pnodo->anterior;
      }
   }
   cout << endl;
}

template<class TIPO>
void lista<TIPO>::Siguiente() {
   if(plista) plista = plista->siguiente;
}

template<class TIPO>
void lista<TIPO>::Anterior() {
   if(plista) plista = plista->anterior;
}

template<class TIPO>
void lista<TIPO>::Primero() {
   while(plista && plista->anterior) plista = plista-
>anterior;
}

template<class TIPO>
void lista<TIPO>::Ultimo() {
   while(plista && plista->siguiente) plista = plista-
>siguiente;
}

int main() {
   lista<int> iLista;
   lista<float> fLista;
   lista<double> dLista;
   lista<char> cLista;
   lista<Cadena> cadLista;

   // Prueba con <int>
   iLista.Insertar(20);
   iLista.Insertar(10);
   iLista.Insertar(40);
   iLista.Insertar(30);

   iLista.Mostrar(ASCENDENTE);
   iLista.Mostrar(DESCENDENTE);
```

```
    iLista.Primero();
    cout << "Primero: " << iLista.ValorActual() << endl;

    iLista.Ultimo();
    cout << "Ultimo: " << iLista.ValorActual() << endl;

    iLista.Borrar(10);
    iLista.Borrar(15);
    iLista.Borrar(45);
    iLista.Borrar(40);

    iLista.Mostrar(ASCENDENTE);
    iLista.Mostrar(DESCENDENTE);

    // Prueba con <float>
    fLista.Insertar(20.01);
    fLista.Insertar(10.02);
    fLista.Insertar(40.03);
    fLista.Insertar(30.04);

    fLista.Mostrar(ASCENDENTE);
    fLista.Mostrar(DESCENDENTE);

    fLista.Primero();
    cout << "Primero: " << fLista.ValorActual() << endl;

    fLista.Ultimo();
    cout << "Ultimo: " << fLista.ValorActual() << endl;

    fLista.Borrar(10.02);
    fLista.Borrar(15.05);
    fLista.Borrar(45.06);
    fLista.Borrar(40.03);

    fLista.Mostrar(ASCENDENTE);
    fLista.Mostrar(DESCENDENTE);

    // Prueba con <double>
    dLista.Insertar(0.0020);
    dLista.Insertar(0.0010);
    dLista.Insertar(0.0040);
    dLista.Insertar(0.0030);

    dLista.Mostrar(ASCENDENTE);
    dLista.Mostrar(DESCENDENTE);

    dLista.Primero();
    cout << "Primero: " << dLista.ValorActual() << endl;
```

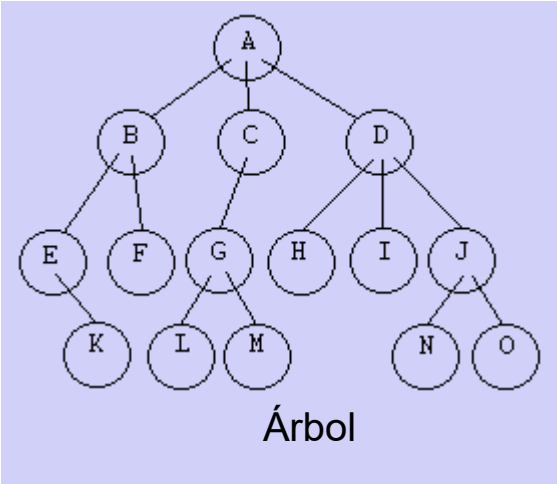
```
    dLista.Ultimo();
    cout << "Ultimo: " << dLista.ValorActual() << endl;

    dLista.Borrar(0.0010);
    dLista.Borrar(0.0015);
    dLista.Borrar(0.0045);
    dLista.Borrar(0.0040);

    dLista.Mostrar(ASCENDENTE);
    dLista.Mostrar(DESCENDENTE);

    // Prueba con <char>
    cLista.Insertar('x');
    cLista.Insertar('y');
    cLista.Insertar('a');
    cLista.Insertar('b');

    cLista.Mostrar(ASCENDENTE);
    cLista.Mostrar(DESCENDENTE);

    cLista.Primero();
    cout << "Primero: " << cLista.ValorActual() << endl;

    cLista.Ultimo();
    cout << "Ultimo: " << cLista.ValorActual() << endl;

    cLista.Borrar('y');
    cLista.Borrar('m');
    cLista.Borrar('n');
    cLista.Borrar('a');

    cLista.Mostrar(ASCENDENTE);
    cLista.Mostrar(DESCENDENTE);

    // Prueba con <Cadena>
    cadLista.Insertar("Hola");
    cadLista.Insertar("seguimos");
    cadLista.Insertar("estando");
    cadLista.Insertar("aquí");

    cadLista.Mostrar(ASCENDENTE);
    cadLista.Mostrar(DESCENDENTE);

    cadLista.Primero();
    cout << "Primero: " << cadLista.ValorActual() << endl;

    cadLista.Ultimo();
```

```
    cout << "Ultimo: " << cadLista.ValorActual() << endl;

    cadLista.Borrar("seguimos");
    cadLista.Borrar("adios");
    cadLista.Borrar("feos");
    cadLista.Borrar("estando");

    cadLista.Mostrar(ASCENDENTE);
    cadLista.Mostrar(DESCENDENTE);

    cin.get();
    return 0;
}
```

Fichero con el código fuente

Capítulo 6 Árboles

6.1 Definición



Árbol

Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos.

También se suele dar una definición recursiva: un árbol es una estructura en compuesta por un dato y varios árboles.

Esto son definiciones simples. Pero las características que implican no lo son tanto.

Definiremos varios conceptos. En relación con otros nodos:

- **Nodo hijo:** cualquiera de los nodos apuntados por uno de los nodos del árbol. En el ejemplo, 'L' y 'M' son hijos de 'G'.
- **Nodo padre:** nodo que contiene un puntero al nodo actual. En el ejemplo, el nodo 'A' es padre de 'B', 'C' y 'D'.

Los árboles con los que trabajaremos tienen otra característica importante: cada nodo sólo puede ser apuntado por otro nodo, es decir, cada nodo sólo tendrá un padre. Esto hace que estos árboles estén fuertemente jerarquizados, y es lo que en realidad les da la apariencia de árboles.

En cuanto a la posición dentro del árbol:

- **Nodo raíz:** nodo que no tiene padre. Este es el nodo que usaremos para referirnos al árbol. En el ejemplo, ese nodo es el 'A'.

- **Nodo hoja:** nodo que no tiene hijos. En el ejemplo hay varios: 'F', 'H', 'I', 'K', 'L', 'M', 'N' y 'O'.
- **Nodo rama:** aunque esta definición apenas la usaremos, estos son los nodos que no pertenecen a ninguna de las dos categorías anteriores. En el ejemplo: 'B', 'C', 'D', 'E', 'G' y 'J'.

Otra característica que normalmente tendrán nuestros árboles es que todos los nodos contengan el mismo número de punteros, es decir, usaremos la misma estructura para todos los nodos del árbol. Esto hace que la estructura sea más sencilla, y por lo tanto también los programas para trabajar con ellos.

Tampoco es necesario que todos los nodos hijos de un nodo concreto existan. Es decir, que pueden usarse todos, algunos o ninguno de los punteros de cada nodo.

Un árbol en el que en cada nodo o bien todos o ninguno de los hijos existe, se llama **árbol completo**.

En una cosa, los árboles se parecen al resto de las estructuras que hemos visto: dado un nodo cualquiera de la estructura, podemos considerarlo como una estructura independiente. Es decir, un nodo cualquiera puede ser considerado como la raíz de un árbol completo.

Existen otros conceptos que definen las características del árbol, en relación a su tamaño:

- **Orden**: es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos es de *orden dos*, si puede apuntar a tres será de *orden tres*, etc.
- **Grado**: el número de hijos que tiene el elemento con más hijos dentro del árbol. En el árbol del ejemplo, el grado es tres, ya que tanto 'A' como 'D' tienen tres hijos, y no existen elementos con más de tres hijos.
- **Nivel**: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero y el de sus hijos uno. Así sucesivamente. En el ejemplo, el nodo 'D' tiene nivel 1, el nodo 'G' tiene nivel 2, y el nodo 'N', nivel 3.

- **Altura**: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas. El árbol del ejemplo tiene altura 3, la rama 'B' tiene altura 2, la rama 'G' tiene altura 1, la 'H' cero, etc.

Los árboles de orden dos son bastante especiales, de hecho les dedicaremos varios capítulos. Estos árboles se conocen también como **árboles binarios**.

Frecuentemente, aunque tampoco es estrictamente necesario, para hacer más fácil moverse a través del árbol, añadiremos un puntero a cada nodo que apunte al nodo padre. De este modo podremos avanzar en dirección a la raíz, y no sólo hacia las hojas.

Es importante conservar siempre el nodo *raíz* ya que es el nodo a partir del cual se desarrolla el árbol, si perdemos este nodo, perderemos el acceso a todo el árbol.

El nodo típico de un árbol difiere de los nodos que hemos visto hasta ahora para listas, aunque sólo en el número de nodos. Veamos un ejemplo de nodo para crear árboles de orden tres:

```
struct nodo {
   int dato;
   struct nodo *rama1;
   struct nodo *rama2;
   struct nodo *rama3;
};
```

O generalizando más:

```
#define ORDEN 5

struct nodo {
   int dato;
   struct nodo *rama[ORDEN];
};
```

6.2 Declaraciones de tipos para manejar árboles en C

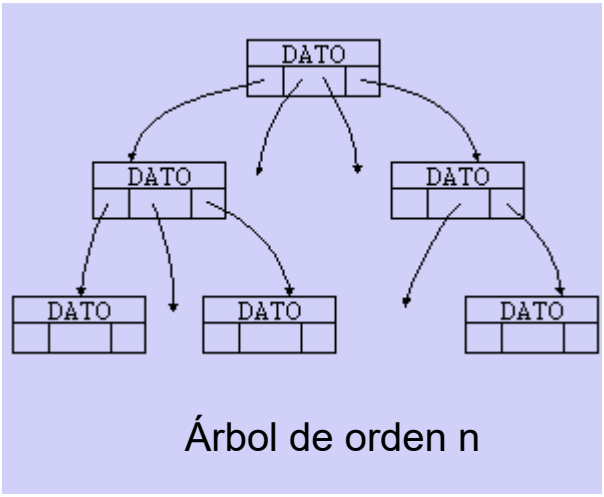
Para C, y basándonos en la declaración de nodo que hemos visto más arriba, trabajaremos con los siguientes tipos:

```
typedef struct _nodo {
   int dato;
   struct _nodo *rama[ORDEN];
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

Al igual que hicimos con las listas que hemos visto hasta ahora, declaramos un tipo *tipoNodo* para declarar nodos, y un tipo *pNodo* para es el tipo para declarar punteros a un nodo.

Arbol es el tipo para declarar árboles de orden *ORDEN*.

El movimiento a través de árboles, salvo que implementemos punteros al nodo padre, será siempre partiendo del nodo raíz hacia un nodo hoja. Cada vez que lleguemos a un nuevo nodo podremos optar por cualquiera de los nodos a los que apunta para avanzar al siguiente nodo.



Árbol de orden n

En general, intentaremos que exista algún significado asociado a cada uno de los punteros dentro de cada nodo, los árboles que estamos viendo son abstractos, pero las aplicaciones no tienen por qué serlo. Un ejemplo de estructura en árbol es el sistema de directorios y ficheros de un sistema operativo. Aunque en este caso se trata de árboles con nodos de dos tipos, nodos directotio y nodos

fichero, podríamos considerar que los nodos hoja son ficheros y los nodos rama son directorios.

Otro ejemplo podría ser la tabla de contenido de un libro, por ejemplo de este mismo curso, dividido en capítulos, y cada uno de ellos en subcapítulos. Aunque el libro sea algo lineal, como una lista, en el que cada capítulo sigue al anterior, también es posible acceder a cualquier punto de él a través de la tabla de contenido.

También se suelen organizar en forma de árbol los organigramas de mando en empresas o en el ejército, y los árboles genealógicos.

6.3 Operaciones básicas con árboles

Salvo que trabajemos con árboles especiales, como los que veremos más adelante, las inserciones serán siempre en punteros de nodos hoja o en punteros libres de nodos rama. Con estas estructuras no es tan fácil generalizar, ya que existen muchas variedades de árboles.

De nuevo tenemos casi el mismo repertorio de operaciones de las que disponíamos con las listas:

- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través del árbol.
- Recorrer el árbol completo.

Los algoritmos de inserción y borrado dependen en gran medida del tipo de árbol que estemos implementando, de modo que por ahora los pasaremos por alto y nos centraremos más en el modo de recorrer árboles.

6.4 Recorridos por árboles

El modo evidente de moverse a través de las ramas de un árbol es siguiendo los punteros, del mismo modo en que nos movíamos a través de las listas.

Esos recorridos dependen en gran medida del tipo y propósito del árbol, pero hay ciertos recorridos que usaremos frecuentemente. Se trata de aquellos recorridos que incluyen todo el árbol.

Hay tres formas de recorrer un árbol completo, y las tres se suelen implementar mediante recursividad. En los tres casos se sigue siempre a partir de cada nodo todas las ramas una por una.

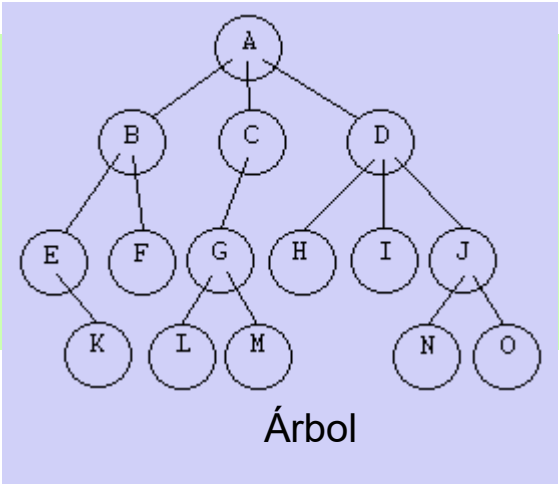
Supongamos que tenemos un árbol de orden tres, y queremos recorrerlo por completo.

Partiremos del nodo raíz:

```
RecorrerArbol(raiz);
```

La función *RecorrerArbol*, aplicando recursividad, será tan sencilla como invocar de nuevo a la función *RecorrerArbol* para cada una de las ramas:

```
void RecorrerArbol(Arbol a) {
   if(a == NULL) return;
   RecorrerArbol(a->rama[0]);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
}
```



Árbol

Lo que diferencia los distintos métodos de recorrer el árbol no es el sistema de hacerlo, sino el momento que elegimos para procesar el valor de cada nodo con relación a los recorridos de cada una de las ramas.

Los tres tipos son:

Pre-orden

En este tipo de recorrido, el valor del nodo se procesa antes de recorrer las ramas:

```
void PreOrden(Arbol a) {
   if(a == NULL) return;
   Procesar(dato);
   RecorrerArbol(a->rama[0]);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
}
```

Si seguimos el árbol del ejemplo en pre-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

```
A B E K F C G L M D H I J N O
```

In-orden

En este tipo de recorrido, el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última. Esto tiene más sentido en el caso de árboles binarios, y también cuando existen ORDEN-1 datos, en cuyo caso procesaremos cada dato entre el recorrido de cada dos ramas (este es el caso de los árboles-b):

```
void InOrden(Arbol a) {
   if(a == NULL) return;
   RecorrerArbol(a->rama[0]);
   Procesar(dato);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
}
```

Si seguimos el árbol del ejemplo en in-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

```
K E B F A L G M C H D I N J O
```

Post-orden

En este tipo de recorrido, el valor del nodo se procesa después de recorrer todas las ramas:

```
void PostOrden(Arbol a) {
   if(a == NULL) return;
   RecorrerArbol(a->rama[0]);
   RecorrerArbol(a->rama[1]);
   RecorrerArbol(a->rama[2]);
   Procesar(dato);
}
```

Si seguimos el árbol del ejemplo en post-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

```
K E F B L M G C H I N O J D A
```

6.5 Eliminar nodos en un árbol

El proceso general es muy sencillo en este caso, pero con una importante limitación, sólo podemos borrar nodos hoja:

El proceso sería el siguiente:

1. Buscar el nodo padre del que queremos eliminar.
2. Buscar el puntero del nodo padre que apunta al nodo que queremos borrar.

3. Liberar el nodo.
4. padre->nodo[i] = NULL;.

Cuando el nodo a borrar no sea un nodo hoja, diremos que hacemos una "poda", y en ese caso eliminaremos el árbol cuya raíz es el nodo a borrar. Se trata de un procedimiento recursivo, aplicamos el recorrido PostOrden, y el proceso será borrar el nodo.

El procedimiento es similar al de borrado de un nodo:

1. Buscar el nodo padre del que queremos eliminar.
2. Buscar el puntero del nodo padre que apunta al nodo que queremos borrar.
3. Podar el árbol cuyo padre es nodo.
4. padre->nodo[i] = NULL;.

En el árbol del ejemplo, para podar la rama 'B', recorreremos el subárbol 'B' en postorden, eliminando cada nodo cuando se procese, de este modo no perdemos los punteros a las ramas apuntadas por cada nodo, ya que esas ramas se borrarán antes de eliminar el nodo.

De modo que el orden en que se borrarán los nodos será:

```
K E F y B
```

6.6 árboles ordenados

A partir del siguiente capítulo sólo hablaremos de árboles ordenados, ya que son los que tienen más interés desde el punto de vista de TAD, y los que tienen más aplicaciones genéricas.

Un árbol ordenado, en general, es aquel a partir del cual se puede obtener una secuencia ordenada siguiendo uno de los recorridos posibles del árbol: inorden, preorden o postorden.

En estos árboles es importante que la secuencia se mantenga ordenada aunque se añadan o se eliminen nodos.

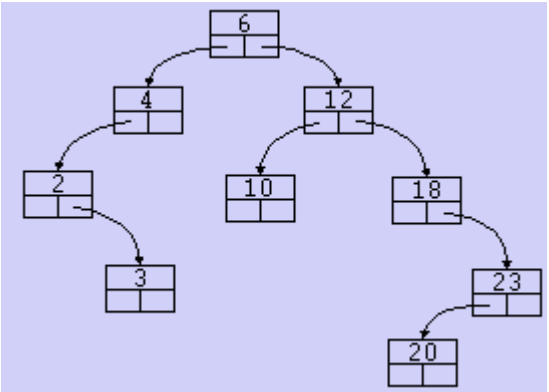
Existen varios tipos de árboles ordenados, que veremos a continuación:

- árboles binarios de búsqueda (ABB): son árboles de orden 2 que mantienen una secuencia ordenada si se recorren en inorden.
- árboles AVL: son árboles binarios de búsqueda equilibrados, es decir, los niveles de cada rama para cualquier nodo no difieren en más de 1.
- árboles perfectamente equilibrados: son árboles binarios de búsqueda en los que el número de nodos de cada rama para cualquier nodo no difieren en más de 1. Son por lo tanto árboles AVL también.
- árboles 2-3: son árboles de orden 3, que contienen dos claves en cada nodo y que están también equilibrados. También generan secuencias ordenadas al recorrerlos en inorden.
- árboles-B: caso general de árboles 2-3, que para un orden M, contienen M-1 claves.

Capítulo 7 Árboles binarios de búsqueda (ABB)

7.1 Definición

Se trata de árboles de orden 2 en los que se cumple que para cada nodo, el valor de la clave de la raíz del subárbol izquierdo es menor que el valor de la clave del nodo y que el valor de la clave raíz del subárbol derecho es mayor que el valor de la clave del nodo.



Árbol binario de búsqueda

7.2 Operaciones en ABB

El repertorio de operaciones que se pueden realizar sobre un ABB es parecido al que realizábamos sobre otras estructuras de datos, más alguna otra propia de árboles:

- Buscar un elemento.
- Insertar un elemento.
- Borrar un elemento.
- Movimientos a través del árbol:
  - Izquierda.
  - Derecha.
  - Raiz.
- Información:
  - Comprobar si un árbol está vacío.

- Calcular el número de nodos.
- Comprobar si el nodo es hoja.
- Calcular la altura de un nodo.
- Calcular la altura de un árbol.

7.3 Buscar un elemento

Partiendo siempre del nodo raíz, el modo de buscar un elemento se define de forma recursiva.

- Si el árbol está vacío, terminamos la búsqueda: el elemento no está en el árbol.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

El valor de retorno de una función de búsqueda en un ABB puede ser un puntero al nodo encontrado, o NULL, si no se ha encontrado.

7.4 Insertar un elemento

Para insertar un elemento nos basamos en el algoritmo de búsqueda. Si el elemento está en el árbol no lo insertaremos. Si no lo está, lo insertaremos a continuación del último nodo visitado.

Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

- Padre = NULL
- nodo = Raiz
- Bucle: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.

    - Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo: Padre=nodo, nodo=nodo->izquierdo.
    - Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho: Padre=nodo, nodo=nodo->derecho.
- Si nodo no es NULL, el elemento está en el árbol, por lo tanto salimos.
- Si Padre es NULL, el árbol estaba vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
- Si el elemento es menor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol izquierdo de Padre.
- Si el elemento es mayor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol derecho de Padre.

Este modo de actuar asegura que el árbol sigue siendo ABB.

7.5 Borrar un elemento

Para borrar un elemento también nos basamos en el algoritmo de búsqueda. Si el elemento no está en el árbol no lo podremos borrar. Si está, hay dos casos posibles:

1. Se trata de un nodo hoja: en ese caso lo borraremos directamente.
2. Se trata de un nodo rama: en ese caso no podemos eliminarlo, puesto que perderíamos todos los elementos del árbol de que el nodo actual es padre. En su lugar buscamos el nodo más a la izquierda del subárbol derecho, o el más a la derecha del subárbol izquierdo e intercambiamos sus valores. A continuación eliminamos el nodo hoja.

Necesitamos un puntero auxiliar para conservar una referencia al padre del nodo raíz actual. El valor inicial para ese puntero es NULL.

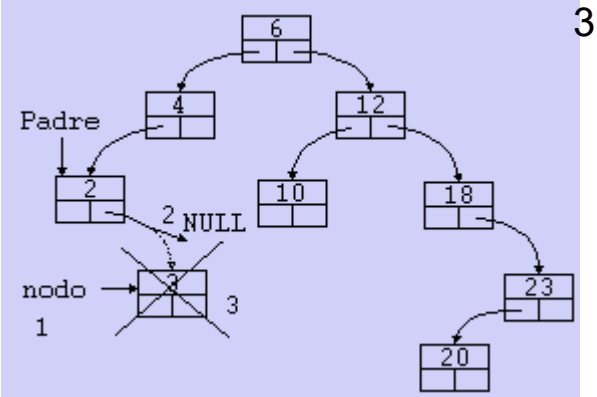
- Padre = NULL

- Si el árbol está vacío: el elemento no está en el árbol, por lo tanto salimos sin eliminar ningún elemento.
- (1) Si el valor del nodo raíz es igual que el del elemento que buscamos, estamos ante uno de los siguientes casos:
    - El nodo raíz es un nodo hoja:
        - Si 'Padre' es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
        - Si raíz es la rama derecha de 'Padre', hacemos que esa rama apunte a NULL.
        - Si raíz es la rama izquierda de 'Padre', hacemos que esa rama apunte a NULL.
        - Eliminamos el nodo, y salimos.
    - El nodo no es un nodo hoja:
        - Buscamos el 'nodo' más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, actualizamos 'Padre' para que apunte al padre de 'nodo'.
        - Intercambiamos los elementos de los nodos raíz y 'nodo'.
        - Borramos el nodo 'nodo'. Esto significa volver a (1), ya que puede suceder que 'nodo' no sea un nodo hoja. (Ver ejemplo 3)
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Ejemplo 1: Borrar un nodo hoja

En el árbol de ejemplo, borrar el nodo 3.

1. Localizamos el nodo a borrar, al tiempo que mantenemos un puntero a 'Padre'.
2. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.



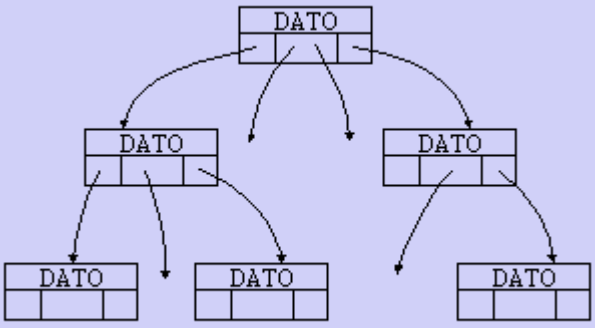
Borrar un nodo hoja

3. Borramos el 'nodo'.

Ejemplo 2: Borrar un nodo rama con intercambio de un nodo hoja

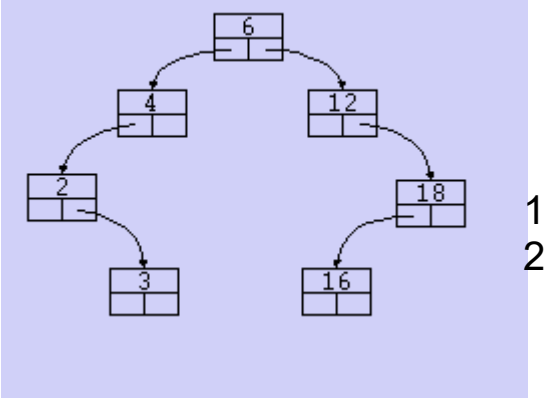
En el árbol de ejemplo, borrar el nodo 4.

1. Localizamos el nodo a borrar ('raíz').
2. Buscamos el nodo más a la derecha del árbol izquierdo de 'raíz', en este caso el 3, al tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
3. Intercambiamos los elementos 3 y 4.
4. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
5. Borramos el 'nodo'.



Borrar con intercambio de nodo hoja

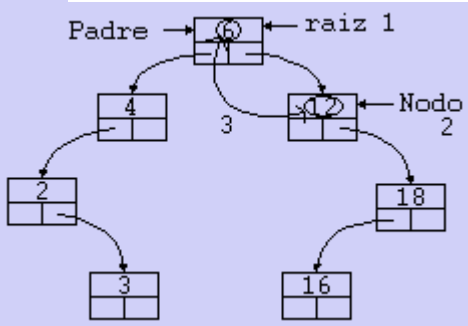
Ejemplo 3: Borrar un nodo rama con intercambio de un nodo rama



Para este ejemplo usaremos otro árbol. En éste borraremos el elemento 6.

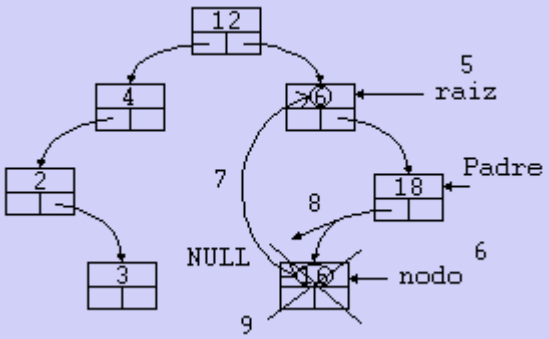
1. Localizamos el nodo a borrar ('raíz').
2. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 12, ya que el árbol

derecho no tiene nodos a su izquierda, si optamos por la rama izquierda, estaremos en un caso análogo. Al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.



Borrar con intercambio de nodo rama (1)

3. Intercambiamos los elementos 6 y 12.
4. Ahora tenemos que repetir el bucle para el nodo 6 de nuevo, ya que no podemos eliminarlo.
5. Localizamos de nuevo el nodo a borrar ('raíz').
6. Buscamos el nodo más a la izquierda del árbol derecho de 'raíz', en este caso el 16, al mismo tiempo que mantenemos un puntero a 'Padre' a 'nodo'.
7. Intercambiamos los elementos 6 y 16.
8. Hacemos que el puntero de 'Padre' que apuntaba a 'nodo', ahora apunte a NULL.
9. Borramos el 'nodo'.



Borrar con intercambio de nodo rama (2)

Este modo de actuar asegura que el árbol sigue siendo ABB.

7.6 Movimientos a través del árbol

No hay mucho que contar. Nuestra estructura se referenciará siempre mediante un puntero al nodo Raiz, este puntero no debe perderse nunca.

Para movernos a través del árbol usaremos punteros auxiliares, de modo que desde cualquier puntero los movimientos posibles serán: moverse al nodo raíz de la rama izquierda, moverse al nodo raíz de la rama derecha o moverse al nodo Raiz del árbol.

7.7 Información

Hay varios parámetros que podemos calcular o medir dentro de un árbol. Algunos de ellos nos darán idea de lo eficientemente que está organizado o el modo en que funciona.

Comprobar si un árbol está vacío

Un árbol está vacío si su raíz es NULL.

Calcular el número de nodos.

Tenemos dos opciones para hacer esto, una es llevar siempre la cuenta de nodos en el árbol al mismo tiempo que se añaden o eliminan elementos. La otra es, sencillamente, contarlos.

Para contar los nodos podemos recurrir a cualquiera de los tres modos de recorrer el árbol: inorden, preorden o postorden, como acción sencillamente incrementamos el contador.

Comprobar si el nodo es hoja

Esto es muy sencillo, basta con comprobar si tanto el árbol izquierdo como el derecho están vacíos. Si ambos lo están, se trata de un nodo hoja.

Calcular la altura de un nodo

No hay un modo directo de hacer esto, ya que no nos es posible recorrer el árbol en la dirección de la raíz. De modo que tendremos

que recurrir a otra técnica para calcular la altura.

Lo que haremos es buscar el elemento del nodo de que queremos averiguar la altura. Cada vez que avancemos un nodo incrementamos la variable que contendrá la altura del nodo.

- Empezamos con el nodo raíz apuntando a Raiz, y la 'Altura' igual a cero.
- Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda y el valor de la altura es 'Altura'.
- Incrementamos 'Altura'.
- Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
- Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

Calcular la altura de un árbol

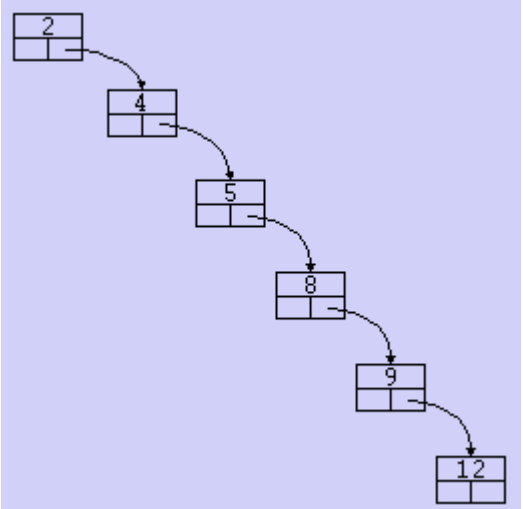
La altura del árbol es la altura del nodo de mayor altura. Para buscar este valor tendremos que recorrer todo el árbol, de nuevo es indiferente el tipo de recorrido que hagamos, cada vez que cambiemos de nivel incrementamos la variable que contiene la altura del nodo actual, cuando lleguemos a un nodo hoja compararemos su altura con la variable que contiene la altura del árbol si es mayor, actualizamos la altura del árbol.

- Iniciamos un recorrido del árbol en postorden, con la variable de altura igual a cero.
- Cada vez que empecemos a recorrer una nueva rama, incrementamos la altura para ese nodo.
- Después de procesar las dos ramas, verificamos si la altura del nodo es mayor que la variable que almacena la altura actual del árbol, si es así, actualizamos esa variable.

7.8 árboles degenerados

Los árboles binarios de búsqueda tienen un gran inconveniente. Por ejemplo, supongamos que creamos un ABB a partir de una lista de valores ordenada:

```
2, 4, 5, 8, 9, 12
```



Árbol degenerado

Difícilmente podremos llamar a la estructura resultante un árbol.

Esto es lo que llamamos un árbol binario de búsqueda degenerado, y en el siguiente capítulo veremos una nueva estructura, el árbol AVL, que resuelve este problema, generando árboles de búsqueda equilibrados.

7.9 Ejemplo de ABB en C

Vamos a ver cómo implementar en C algunas de las funciones que hemos explicado para árboles ABB, al final se incluye un ejemplo completo para árboles de enteros.

Declaración de tipos

Como estamos trabajando con un árbol binario, sólo necesitamos una estructura para referirnos tanto a cualquiera de los nodos como al árbol completo. Recuerda que cualquier nodo puede ser considerado como la raíz de un árbol.

```
typedef struct _nodo {
   int dato;
   struct _nodo *derecho;
   struct _nodo *izquierdo;
} tipoNodo;
```

```
typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

Insertar un elemento en un árbol ABB

Diseñaremos una función que se ajuste al algoritmo que describimos en el punto 7.4:

1. Padre = NULL
2. nodo = Raiz
3. Bucle: mientras actual no sea un árbol vacío o hasta que se encuentre el elemento.
   a. Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo: Padre=nodo, nodo=nodo->izquierdo.
   b. Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho: Padre=nodo, nodo=nodo->derecho.
4. Si nodo no es NULL, el elemento está en el árbol, por lo tanto salimos.
5. Si Padre es NULL, el árbol estaba vacío, por lo tanto, el nuevo árbol sólo contendrá el nuevo elemento, que será la raíz del árbol.
6. Si el elemento es menor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol izquierdo de Padre.
7. Si el elemento es mayor que el Padre, entonces insertamos el nuevo elemento como un nuevo árbol derecho de Padre.

```
void Insertar(Arbol *a, int dat) {
   pNodo padre = NULL; /* (1) */
   pNodo actual = *a;  /* (2) */

   while(!Vacio(actual) && dat != actual->dato) {  /* (3) */
      padre = actual;
      if(dat < actual->dato) actual = actual->izquierdo;  /*
(3-a) */
      else if(dat > actual->dato) actual = actual->derecho;
```

```
 /* (3-b) */
    }

    if(!Vacio(actual)) return;  /* (4) */
    if(Vacio(padre)) {  /* (5) */
       *a = (Arbol)malloc(sizeof(tipoNodo));
       (*a)->dato = dat;
       (*a)->izquierdo = (*a)->derecho = NULL;
    }
    else if(dat < padre->dato) { /* (6) */
       actual = (Arbol)malloc(sizeof(tipoNodo));
       padre->izquierdo = actual;
       actual->dato = dat;
       actual->izquierdo = actual->derecho = NULL;
    }
    else if(dat > padre->dato) { /* (7) */
       actual = (Arbol)malloc(sizeof(tipoNodo));
       padre->derecho = actual;
       actual->dato = dat;
       actual->izquierdo = actual->derecho = NULL;
   }
 }
```

Eliminar un elemento de un árbol ABB

Diseñaremos una función que se ajuste al algoritmo que describimos en el punto 7.5:

1. Padre = NULL
2. Si el árbol está vacío: el elemento no está en el árbol, por lo tanto salimos sin eliminar ningún elemento.
3. Si el valor del nodo raíz es igual que el del elemento que buscamos, estamos ante uno de los siguientes casos:
    a. El nodo raíz es un nodo hoja:
        i. Si 'Padre' es NULL, el nodo raíz es el único del árbol, por lo tanto el puntero al árbol debe ser NULL.
        ii. Si raíz es la rama derecha de 'Padre', hacemos que esa rama apunte a NULL.
        iii. Si raíz es la rama izquierda de 'Padre', hacemos que esa rama apunte a NULL.
        iv. Eliminamos el nodo, y salimos.

b. El nodo no es un nodo hoja:
    i. Buscamos el 'nodo' más a la izquierda del árbol derecho de raíz o el más a la derecha del árbol izquierdo. Hay que tener en cuenta que puede que sólo exista uno de esos árboles. Al mismo tiempo, actualizamos 'Padre' para que apunte al padre de 'nodo'.
    ii. Intercambiamos los elementos de los nodos raíz y 'nodo'.
    iii. Borramos el nodo 'nodo'. Esto significa volver a (3), ya que puede suceder que 'nodo' no sea un nodo hoja.
4. Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
5. Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

```
void Borrar(Arbol *a, int dat) {
   pNodo padre = NULL; /* (1) */
   pNodo actual;
   pNodo nodo;
   int aux;

   actual = *a;
   while(!Vacio(actual)) { /* Búsqueda (2) else implícito */
      if(dat == actual->dato) { /* (3) */
         if(EsHoja(actual)) { /* (3-a) */
            if(padre)/* (3-a-i caso else implícito) */
               if(padre->derecho == actual) padre->derecho =
NULL;  /* (3-a-ii) */
               else if(padre->izquierdo == actual) padre-
>izquierdo = NULL; /* (3-a-iii) */
            free(actual); /* (3-a-iv) */
            actual = NULL;
            return;
         }
         else { /* (3-b) */
            /* Buscar nodo */
            padre = actual; /* (3-b-i) */
            if(actual->derecho) {
               nodo = actual->derecho;
               while(nodo->izquierdo) {
```

```
                padre = nodo;
                nodo = nodo->izquierdo;
             }
          }
          else {
             nodo = actual->izquierdo;
             while(nodo->derecho) {
                padre = nodo;
                nodo = nodo->derecho;
             }
          }
          /* Intercambio */
          aux = actual->dato; /* (3-b-ii) */
          actual->dato = nodo->dato;
          nodo->dato = aux;
          actual = nodo;
       }
    }
    else {
       padre = actual;
       if(dat > actual->dato) actual = actual->derecho; /*
(4) */
       else if(dat < actual->dato) actual = actual-
>izquierdo; /* (5) */
    }
  }
}
```

Buscar un elemento en un árbol ABB

Diseñaremos una función que se ajuste al algoritmo que describimos en el punto 7.3:

1. Si el árbol está vacío, terminamos la búsqueda: el elemento no está en el árbol.
2. Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda con éxito.
3. Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
4. Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

```
int Buscar(Arbol a, int dat) {
   pNodo actual = a;

   while(!Vacio(actual)) {
      if(dat == actual->dato) return TRUE; /* dato
encontrado  (2) */
      else if(dat < actual->dato) actual = actual-
>izquierdo;  /* (3) */
      else if(dat > actual->dato) actual = actual->derecho;
/* (4) */
   }
   return FALSE; /* No está en árbol (1) */
}
```

Comprobar si el árbol está vacío

Esta función es fácil de implementar:

```
int Vacio(Arbol r) {
   return r==NULL;
}
```

Comprobar si un nodo es hoja

Esta función también es sencilla de implementar:

```
int EsHoja(pNodo r) {
   return !r->derecho && !r->izquierdo;
}
```

Contar número de nodos

En el punto 7.7 comentábamos que para contar los nodos podemos recurrir a cualquiera de los tres modos de recorrer el árbol:

inorden, preorden o postorden y como acción incrementamos el contador de nodos. Para implementar este algoritmo recurrimos a dos funciones:

```
int NumeroNodos(Arbol a, int *contador) {
   *contador = 0;

   auxContador(a, contador);
   return *contador;
}

void auxContador(Arbol nodo, int *c) {
   (*c)++; /* Acción: incrementar número de nodos.
(Preorden) */
   if(nodo->izquierdo) auxContador(nodo->izquierdo, c); /*
Rama izquierda */
   if(nodo->derecho)   auxContador(nodo->derecho, c);   /*
Rama derecha */
}
```

Calcular la altura de un árbol

Es un problema parecido al anterior, pero ahora tenemos que contar la altura, no en número de nodos. Cada vez que lleguemos a un nodo hoja, verificamos si la altura del nodo es la máxima, y si lo es, actualizamos la altura del árbol a ese valor:

1. Iniciamos un recorrido del árbol en postorden, con la variable de altura igual a cero.
2. Cada vez que empecemos a recorrer una nueva rama, incrementamos la altura para ese nodo.
3. Después de procesar las dos ramas, verificamos si la altura del nodo es mayor que la variable que almacena la altura actual del árbol, si es así, actualizamos esa variable.

```
int AlturaArbol(Arbol a, int *altura) {
   *altura = 0; /* (1) */
```

```
   auxAltura(a, 0, altura);
   return *altura;
}

void auxAltura(pNodo nodo, int a, int *altura) {
   /* (2) Cada vez que llamamos a auxAltura pasamos como
parámetro a+1 */
   if(nodo->izquierdo) auxAltura(nodo->izquierdo, a+1,
altura); /* Rama izquierda */
   if(nodo->derecho)   auxAltura(nodo->derecho, a+1,
altura);   /* Rama derecha */
   if(EsHoja(nodo) && a > *altura) *altura = a; /* Proceso
(Postorden)  (3) */
}
```

Calcular la altura del nodo que contiene un dato concreto

Lo que haremos será buscar el elemento del nodo del que queremos averiguar la altura. Cada vez que avancemos un nodo incrementamos la variable que contendrá la altura del nodo.

1. Empezamos con el nodo raíz apuntando a Raiz, y la 'Altura' igual a cero.
2. Si el valor del nodo raíz es igual que el del elemento que buscamos, terminamos la búsqueda y el valor de la altura es 'Altura'.
3. Incrementamos 'Altura'.
4. Si el valor del nodo raíz es mayor que el elemento que buscamos, continuaremos la búsqueda en el árbol izquierdo.
5. Si el valor del nodo raíz es menor que el elemento que buscamos, continuaremos la búsqueda en el árbol derecho.

```
int Altura(Arbol a, int dat) {
   int altura = 0;
   pNodo actual = a; /* (1) */

   while(!Vacio(actual)) {
```

```
      if(dat == actual->dato) return altura; /* dato
encontrado. (2) */
      else {
         altura++; /* (3) */
         if(dat < actual->dato) actual = actual->izquierdo;
/* (4) */
         else if(dat > actual->dato) actual = actual-
>derecho; /* (5) */
      }
   }
   return -1; /* No está en árbol */
}
```

Aplicar una función a cada elemento del árbol, según los tres posibles recorridos

Todos los recorridos se aplican de forma recursiva. En este ejemplo crearemos tres funciones, una por cada tipo de recorrido, que aplicarán una función al elemento de cada nodo.

La función a aplicar puede ser cualquiera que admita como parámetro un puntero a un entero, y que no tenga valor de retorno.

InOrden

En Inorden, primero procesamos el subárbol izquierdo, después el nodo actual, y finalmente el subárbol derecho:

```
void InOrden(Arbol a, void (*func)(int*)) {
   if(a->izquierdo) InOrden(a->izquierdo, func); /* Subárbol
izquierdo */
   func(&(a->dato)); /* Aplicar la función al dato del nodo
actual */
   if(a->derecho) InOrden(a->derecho, func);     /* Subárbol
derecho */
}
```

PreOrden

En Preorden, primero procesamos el nodo actual, después el subárbol izquierdo, y finalmente el subárbol derecho:

```
void PreOrden(Arbol a, void (*func)(int*)) {
   func(&(a->dato)); /* Aplicar la función al dato del nodo
actual */
   if(a->izquierdo) PreOrden(a->izquierdo, func); /*
Subárbol izquierdo */
   if(a->derecho) PreOrden(a->derecho, func);     /*
Subárbol derecho */
}
```

PostOrden

En Postorden, primero procesamos el subárbol izquierdo, después el subárbol derecho, y finalmente el nodo actual:

```
void PostOrden(Arbol a, void (*func)(int*)) {
   if(a->izquierdo) PostOrden(a->izquierdo, func); /*
Subárbol izquierdo */
   if(a->derecho) PostOrden(a->derecho, func);     /*
Subárbol derecho */
   func(&(a->dato)); /* Aplicar la función al dato del nodo
actual */
}
```

Fichero con el código fuente

7.10 Ejemplo de ABB en C++

Haremos ahora lo mismo que en el ejemplo en C, pero incluyendo todas las funciones y datos en una única clase.

Declaración de clase ArbolABB

Declaramos dos clases, una para nodo y otra para ArbolABB, la clase nodo la declararemos como parte de la clase ArbolABB, de modo que no tendremos que definir relaciones de amistad, y evitamos que otras clases o funciones tengan acceso a los datos internos de nodo.

```
class ArbolABB {
  private:
   //// Clase local de Lista para Nodo de ArbolBinario:
   class Nodo {
     public:
      // Constructor:
      Nodo(const int dat, Nodo *izq=NULL, Nodo *der=NULL) :
        dato(dat), izquierdo(izq), derecho(der) {}
      // Miembros:
      int dato;
      Nodo *izquierdo;
      Nodo *derecho;
   };

   // Punteros de la lista, para cabeza y nodo actual:
   Nodo *raíz;
   Nodo *actual;
   int contador;
   int altura;

  public:
   // Constructor y destructor básicos:
   ArbolABB() : raíz(NULL), actual(NULL) {}
   ~ArbolABB() { Podar(raíz); }
   // Insertar en árbol ordenado:
   void Insertar(const int dat);
   // Borrar un elemento del árbol:
   void Borrar(const int dat);
   // Función de búsqueda:
   bool Buscar(const int dat);
   // Comprobar si el árbol está vacío:
   bool Vacio(Nodo *r) { return r==NULL; }
   // Comprobar si es un nodo hoja:
   bool EsHoja(Nodo *r) { return !r->derecho && !r-
>izquierdo; }
   // Contar número de nodos:
   const int NumeroNodos();
   const int AlturaArbol();
```

```
    // Calcular altura de un int:
    int Altura(const int dat);
    // Devolver referencia al int del nodo actual:
    int &ValorActual() { return actual->dato; }
    // Moverse al nodo raíz:
    void Raiz() { actual = raíz; }
    // Aplicar una función a cada elemento del árbol:
    void InOrden(void (*func)(int&) , Nodo *nodo=NULL, bool
r=true);
    void PreOrden(void (*func)(int&) , Nodo *nodo=NULL, bool
r=true);
    void PostOrden(void (*func)(int&) , Nodo *nodo=NULL, bool
r=true);
  private:
    // Funciones auxiliares
    void Podar(Nodo* &);
    void auxContador(Nodo*);
    void auxAltura(Nodo*, int);
};
```

Definición de las funciones miembro

Las definiciones de las funciones miembro de la clase no difieren demasiado de las que creamos en C. Tan solo se han sustituido algunos punteros por referencias, y se usa el tipo bool cuando es aconsejable.

Por ejemplo, en las funciones de recorrido de árboles, la función invocada acepta ahora una referencia a un entero, en lugar de un puntero a un entero.

Fichero con el código fuente

7.11 Ejemplo de ABB en C++ usando plantillas

Sólo nos queda generalizar la clase del ejemplo anterior implementándola en forma de plantilla.

El proceso es relativamente sencillo, sólo tenemos que cambiar la declaración de dato, y todas sus referencias. Además de cambiar la sintaxis de las definiciones de las funciones, claro.

Declaración de la plantilla ArbolABB

Se trata de una simple generalización de la clase del punto anterior:

```
template<class DATO>
class ABB {
  private:
   //// Clase local de Lista para Nodo de ArbolBinario:
   template<class DATON>
   class Nodo {
     public:
      // Constructor:
      Nodo(const DATON dat, Nodo<DATON> *izq=NULL,
Nodo<DATON> *der=NULL) :
        dato(dat), izquierdo(izq), derecho(der) {}
      // Miembros:
      DATON dato;
      Nodo<DATON> *izquierdo;
      Nodo<DATON> *derecho;
   };

   // Punteros de la lista, para cabeza y nodo actual:
   Nodo<DATO> *raíz;
   Nodo<DATO> *actual;
   int contador;
   int altura;

  public:
   // Constructor y destructor básicos:
   ABB() : raíz(NULL), actual(NULL) {}
   ~ABB() { Podar(raíz); }
   // Insertar en árbol ordenado:
   void Insertar(const DATO dat);
   // Borrar un elemento del árbol:
   void Borrar(const DATO dat);
   // Función de búsqueda:
   bool Buscar(const DATO dat);
   // Comprobar si el árbol está vacío:
```

```
   bool Vacio(Nodo<DATO> *r) { return r==NULL; }
   // Comprobar si es un nodo hoja:
   bool EsHoja(Nodo<DATO> *r) { return !r->derecho && !r-
>izquierdo; }
   // Contar número de nodos:
   const int NumeroNodos();
   const int AlturaArbol();
   // Calcular altura de un dato:
   int Altura(const DATO dat);
   // Devolver referencia al dato del nodo actual:
   DATO &ValorActual() { return actual->dato; }
   // Moverse al nodo raíz:
   void Raiz() { actual = raíz; }
   // Aplicar una función a cada elemento del árbol:
   void InOrden(void (*func)(DATO&) , Nodo<DATO> *nodo=NULL,
bool r=true);
   void PreOrden(void (*func)(DATO&) , Nodo<DATO>
*nodo=NULL, bool r=true);
   void PostOrden(void (*func)(DATO&) , Nodo<DATO>
*nodo=NULL, bool r=true);
  private:
   // Funciones auxiliares
   void Podar(Nodo<DATO>* &);
   void auxContador(Nodo<DATO>*);
   void auxAltura(Nodo<DATO>*, int);
};
```

Definición de las funciones miembro

Las definiciones de las funciones miembro de la clase no difieren en nada de las que creamos en el ejemplo anterior. Tan solo se han sustituido los tipos del dato por el tipo de dato de la plantilla.

Fichero con el código fuente

Capítulo 8 Árboles AVL

8.1 Árboles equilibrados

Ya vimos al final del capítulo anterior que el comportamiento de los ABB no es siempre tan bueno como nos gustaría. Pues bien, para minimizar el problema de los ABB desequilibrados, sea cual sea el grado de desequilibrio que tengan, se puede recurrir a algoritmos de equilibrado de árboles globales. En cuanto a estos algoritmos, existen varios, por ejemplo, crear una lista mediante la lectura en inorden del árbol, y volver a reconstruirlo equilibrado. Conociendo el número de elementos no es demasiado complicado.

El problema de estos algoritmos es que requieren explorar y reconstruir todo el árbol cada vez que se inserta o se elimina un elemento, de modo que lo que ganamos al acortar las búsquedas, teniendo que hacer menos comparaciones, lo perdemos equilibrando el árbol.

Para resolver este inconveniente podemos recurrir a los árboles AVL.

8.2 Definición

Un árbol AVL (llamado así por las iniciales de sus inventores: Adelson-Velskii y Landis) es un árbol binario de búsqueda en el que para cada nodo, las alturas de sus subárboles izquierdo y derecho no difieren en más de 1.

No se trata de árboles perfectamente equilibrados, pero sí son lo suficientemente equilibrados como para que su comportamiento sea lo bastante bueno como para usarlos donde los ABB no garantizan tiempos de búsqueda óptimos.

El algoritmo para mantener un árbol AVL equilibrado se basa en reequilibrados locales, de modo que no es necesario explorar todo el árbol después de cada inserción o borrado.

8.3 Operaciones en AVL

Los AVL son también ABB, de modo que mantienen todas las operaciones que poseen éstos. Las nuevas operaciones son las de equilibrar el árbol, pero eso se hace como parte de las operaciones de insertado y borrado.

8.4 Factor de equilibrio

Cada nodo, además de la información que se pretende almacenar, debe tener los dos punteros a los árboles derecho e izquierdo, igual que los ABB, y además un miembro nuevo: el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

```
FE = altura subárbol derecho - altura subárbol izquierdo;
```

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

8.5 Rotaciones simples de nodos

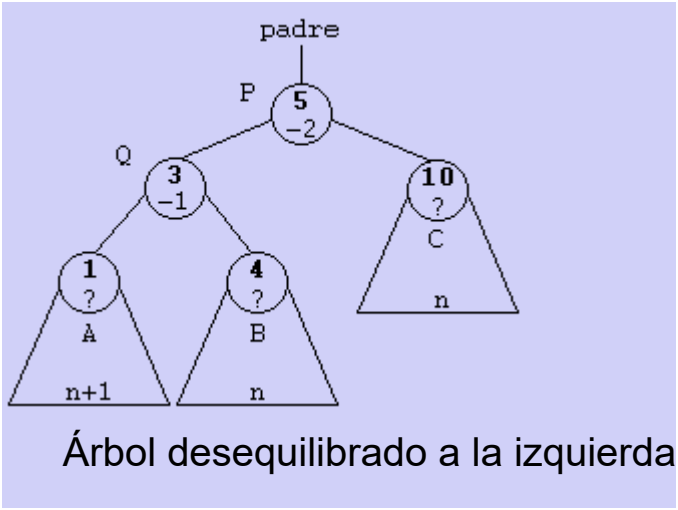
Los reequilibrados se realizan mediante rotaciones, en el siguiente punto veremos cada caso, ahora vamos a ver las cuatro posibles rotaciones que podemos aplicar.

Rotación simple a la derecha (SD)

Esta rotación se usará cuando el subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho, es decir, cuando su FE sea de -2. Y además, la raíz del subárbol izquierdo tenga una FE de -1 ó 0, es decir, que esté cargado a la izquierda o equilibrado.

Procederemos del siguiente modo:

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de -2. Y llamaremos Q al nodo raíz del subárbol izquierdo de P. Además, llamaremos A al subárbol izquierdo de Q, B al subárbol derecho de Q y C al subárbol derecho de P.



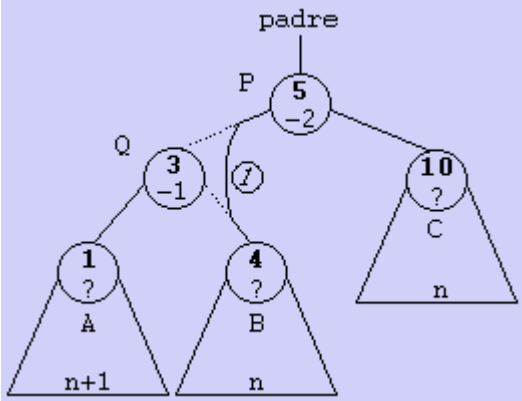
Árbol desequilibrado a la izquierda

En el gráfico que puede observar que tanto B como C tienen la misma altura (n), y A es una unidad mayor (n+1). Esto hace que el FE de Q sea -1, la altura del subárbol que tiene Q como raíz es (n+2) y por lo tanto el FE de P es -2.

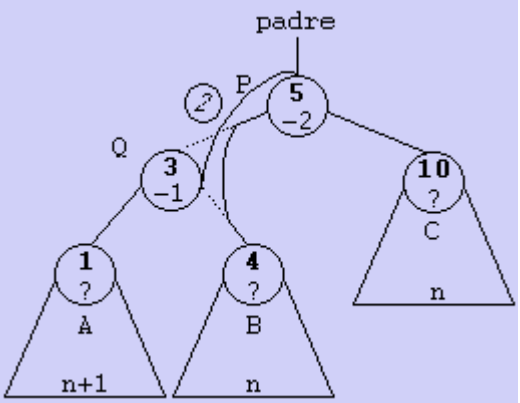
1. Pasamos el subárbol derecho del nodo Q como subárbol izquierdo de P. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de Q siguen estando a la izquierda de P.
2. El árbol P pasa a ser el subárbol derecho del nodo Q.
3. Ahora, el nodo Q pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo Q, en lugar del nodo P. Previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.

En el árbol resultante se puede ver que tanto P como Q quedan equilibrados en cuanto altura.

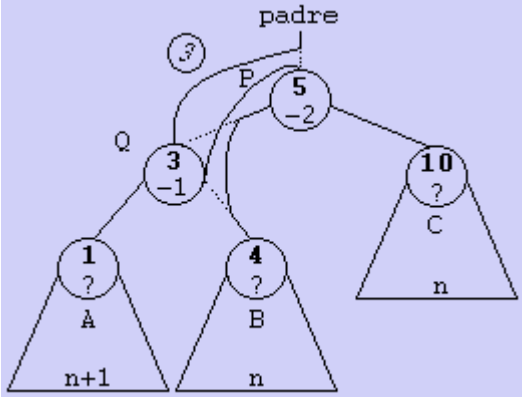
En el caso de P porque sus dos subárboles tienen la misma altura (n), en el caso de Q, porque su subárbol izquierdo A tiene una



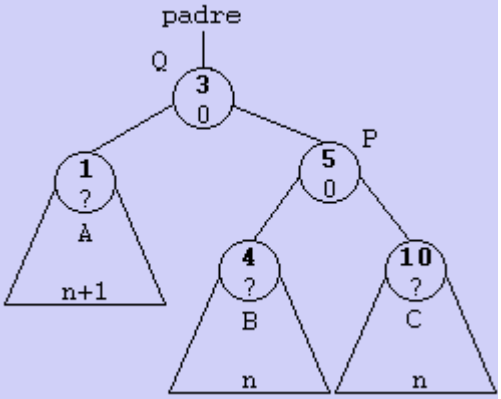
RSD paso 1



RSD paso 2

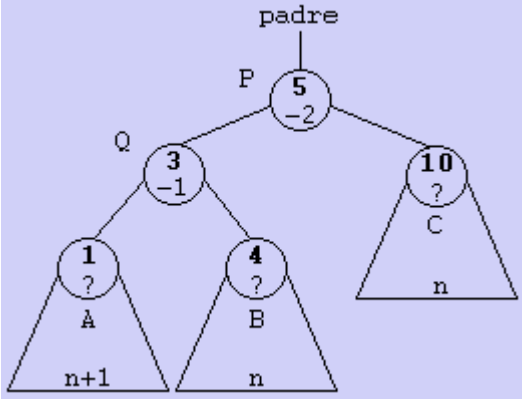


RSD pasos 4 a 6



Árbol equilibrado

altura (n+1) y su subárbol derecho también, ya que a P se añade la altura de cualquiera de sus subárboles.



Árbol desequilibrado a la izquierda caso (b)

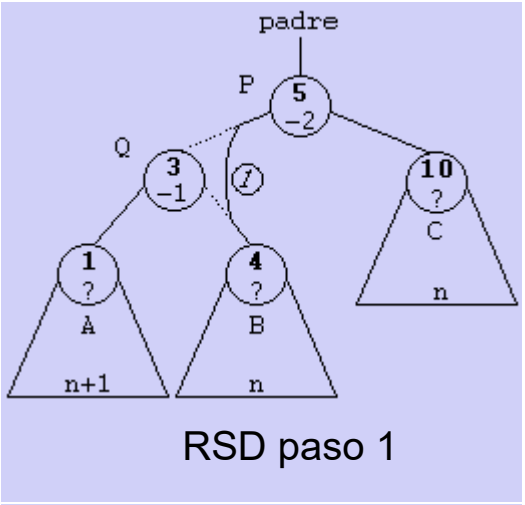
En el caso de que el subárbol izquierdo esté equilibrado, el procedimiento es similar, pero los FE de los nodos P y Q en el árbol resultante son diferentes.

En principio, parece poco probable que nos encontremos un árbol con esta

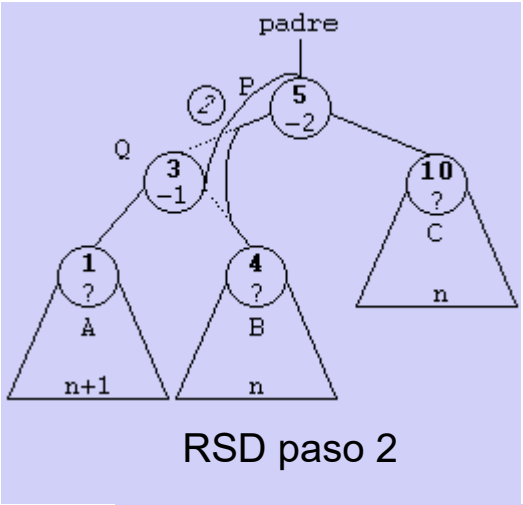
estructura, pero es posible encontrarlos cuando se borran nodos.

Aplicamos el mismo algoritmo para la rotación:

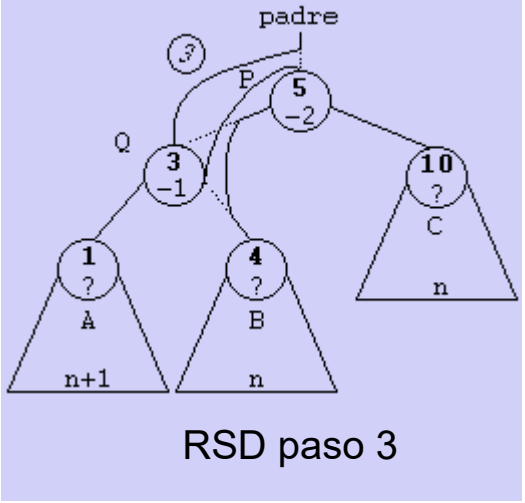
En el árbol resultante se puede ver que tanto P como Q quedan equilibrados en cuanto altura. En el caso de P porque su subárbol izquierdo es una unidad más alto que el derecho, quedando su FE en -1. En el caso de Q, porque su subárbol derecho una altura (n+1) y su subárbol izquierdo, una altura de n.



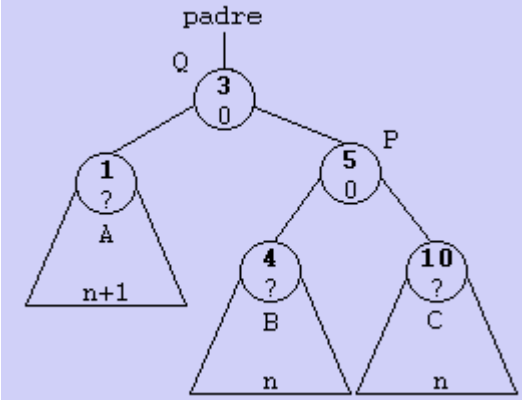
RSD paso 1



RSD paso 2



RSD paso 3

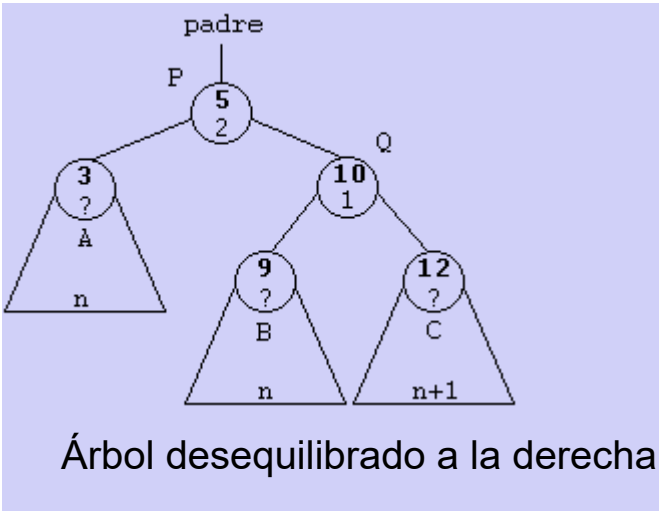


Árbol resultante equilibrado

De modo que, aunque aplicamos el mismo algoritmo, ya que en ambos casos se trata de una rotación simple, deberemos tener en cuenta estos detalles a la hora de ajustar los nuevos valores de FE en nuestro programa.

Rotación simple a la izquierda (SI)

Se trata del caso simétrico del anterior. Esta rotación se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo, es decir, cuando su FE sea de 2. Y además, la raíz del subárbol derecho tenga una FE de 1 ó 0, es decir, que esté cargado a la derecha o esté equilibrado.



Árbol desequilibrado a la derecha

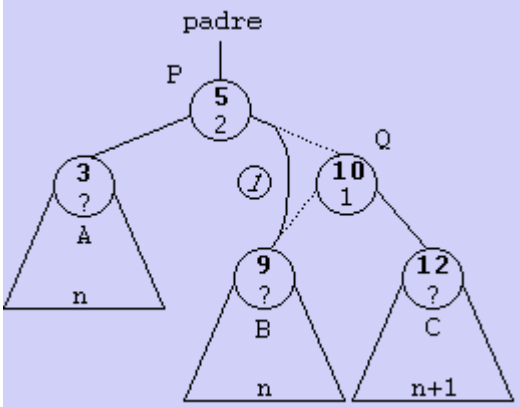
Procederemos del siguiente modo:

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de 2. Y llamaremos Q al nodo raíz del subárbol derecho de P. Además, llamaremos A al subárbol izquierdo de P, B al subárbol izquierdo de Q y C al subárbol derecho de Q.

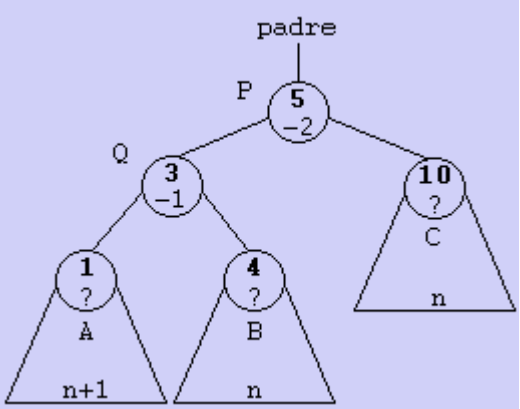
En el gráfico que puede observar que tanto A como B tienen la misma altura (n), y C es una unidad mayor (n+1). Esto hace que el FE de Q sea 1, la altura del subárbol que tiene Q como raíz es (n+2) y por lo tanto el FE de P es 2.

1. Pasamos el subárbol izquierdo del nodo Q como subárbol derecho de P. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de Q siguen estando a la derecha de P.
2. El árbol P pasa a ser el subárbol izquierdo del nodo Q.
3. Ahora, el nodo Q pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo Q, en lugar del nodo P. Previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.

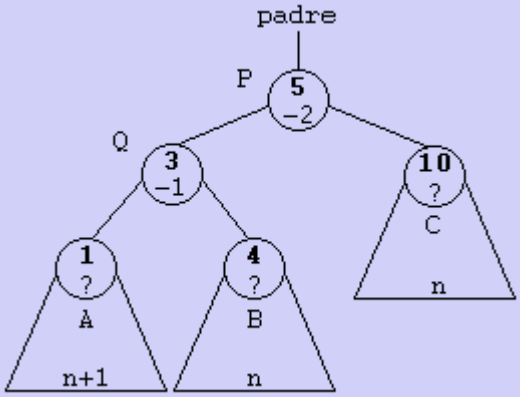
En el árbol resultante se puede ver que tanto P como Q quedan equilibrados en cuanto altura. En el caso de P porque sus dos subárboles tienen la misma altura (n), en el caso de Q, porque su subárbol izquierdo A tiene una altura (n+1) y su subárbol derecho



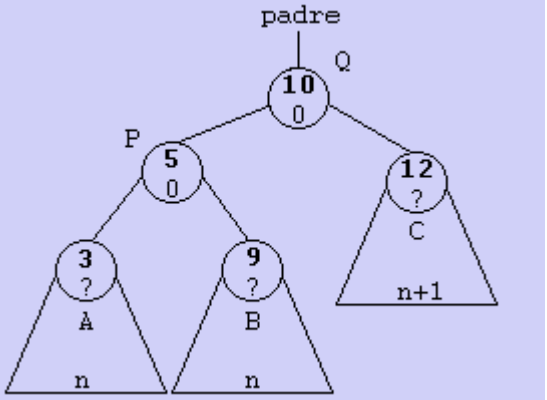
RSI paso 1



RSI paso 2



RSI paso 3



Árbol resultante equilibrado

También, ya que a P se añade la altura de cualquiera de sus subárboles.

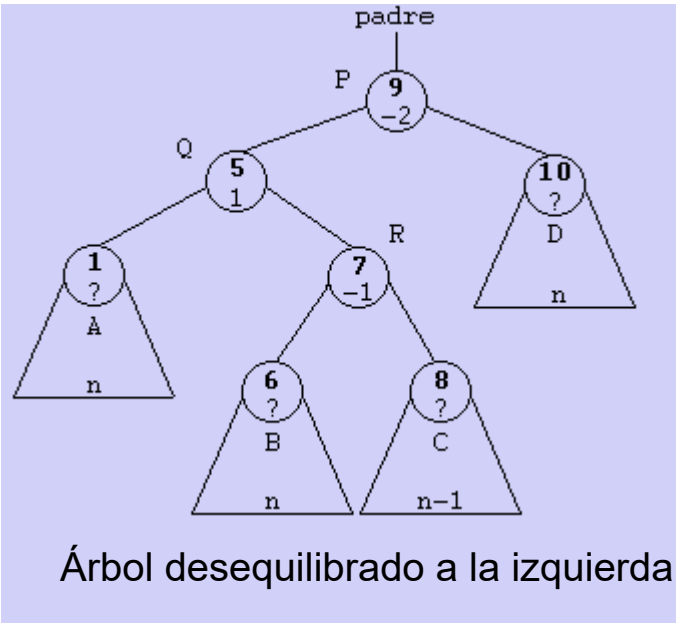
8.

6 Rotaciones dobles de nodos

Rotación doble a la derecha (DD)

Esta rotación se usará cuando el subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho, es decir, cuando su FE sea de -2. Y además, la raíz del subárbol izquierdo tenga una FE de 1, es decir, que esté cargado a la derecha.

Este es uno de los posibles árboles que pueden presentar esta estructura, pero hay otras dos posibilidades. El nodo R puede tener una FE de -1, 0 ó 1. En cada uno de esos casos los árboles izquierdo y derecho de R (B y C) pueden tener alturas de n y n-1, n y n, o n-1 y n, respectivamente.



Árbol desequilibrado a la izquierda

El modo de realizar la rotación es independiente de la estructura del árbol R, cualquiera de las tres produce resultados equivalentes. Haremos el análisis para el caso en que FE sea -1.

En este caso tendremos que realizar dos rotaciones.

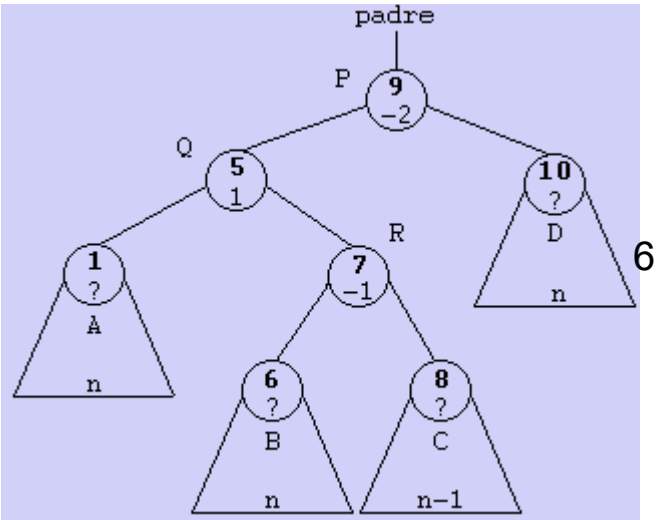
Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de -2. Llamaremos Q al nodo raíz del subárbol izquierdo de P, y R al nodo raíz del subárbol derecho de Q.

1. Haremos una rotación simple de Q a la izquierda.
2. Después, haremos una rotación simple de P a la derecha.

Con más detalle, procederemos del siguiente modo:

1. Pasamos el subárbol izquierdo del nodo R como subárbol derecho de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de Q.
2. Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol izquierdo de P sea el nodo R en lugar de Q.
3. El árbol Q pasa a ser el subárbol izquierdo del nodo R.

4. Pasamos el subárbol derecho del nodo R como subárbol izquierdo de P. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de P.
5. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del



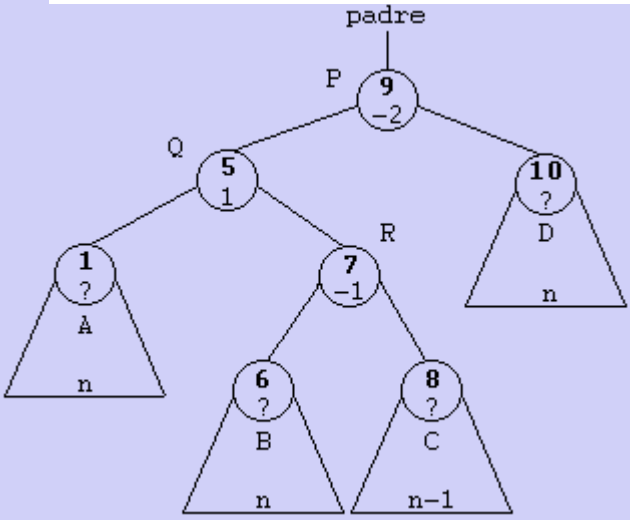
RDD pasos 1 a 3

nodo P. Como en los casos anteriores, previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.

6. El árbol P pasa a ser el subárbol derecho del nodo R.

Rotación doble a la izquierda (DI)

Esta rotación se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo, es decir, cuando su FE sea de 2. Y además, la raíz del subárbol derecho tenga una FE de -1, es decir, que esté cargado a la izquierda. Se trata del caso simétrico del anterior.



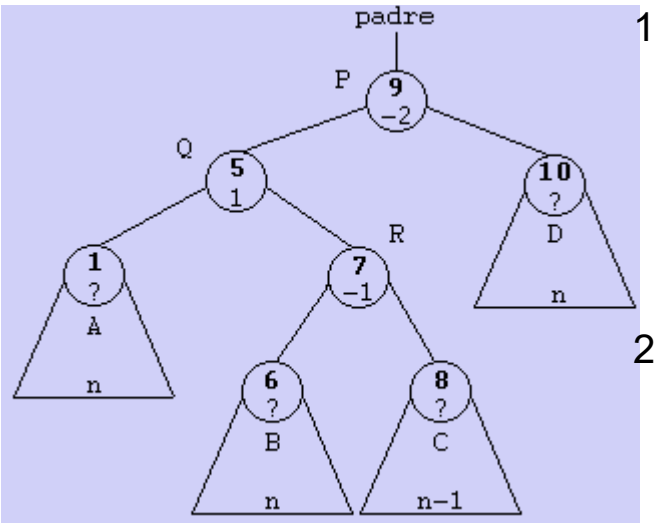
Resultado de primera rotación

En este caso también tendremos que realizar dos rotaciones.

Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de 2. Llamaremos Q al nodo raíz del subárbol derecho de P, y R al nodo raíz del subárbol izquierdo de Q.

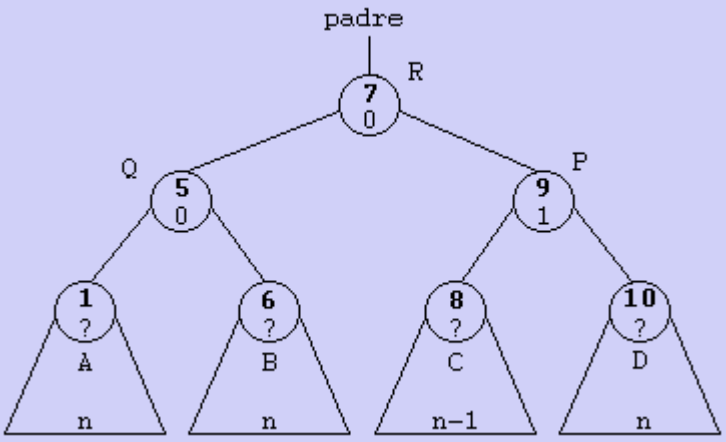
1. Haremos una rotación simple de Q a la derecha.
2. Después, haremos una rotación simple de P a la izquierda.

Con más detalle, procederemos del siguiente modo:



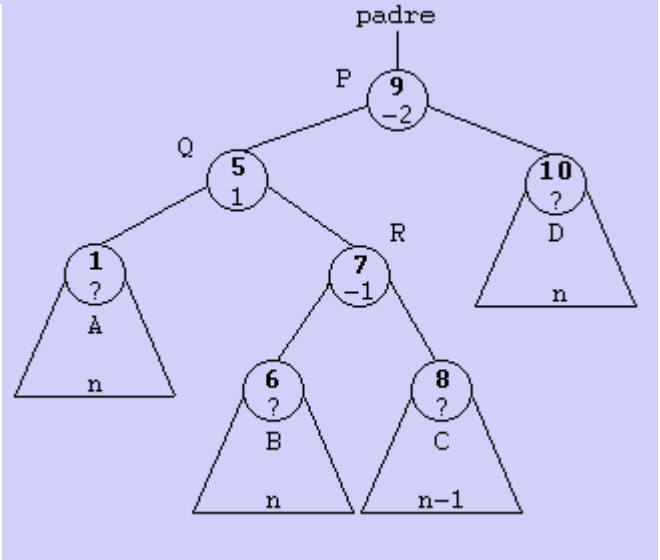
RDD pasos 4 a 6

1. Pasamos el subárbol derecho del nodo R como subárbol izquierdo de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de Q.
2. Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol derecho de P sea el nodo R en lugar de Q.
3. El árbol Q pasa a ser el subárbol derecho del nodo R.



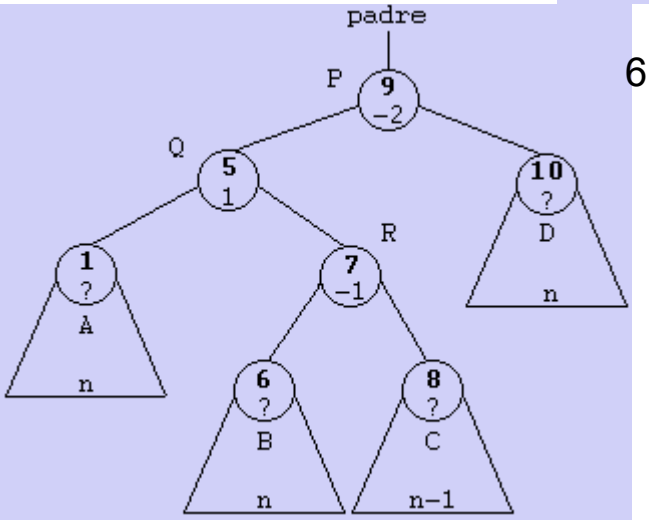
Árbol equilibrado

4. Pasamos el subárbol izquierdo del nodo R como subárbol derecho de P. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de P.
5. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del nodo P. Como en los casos anteriores, previamente, P puede que fuese un árbol completo o un



subárbol de otro nodo de menor altura.

6. El árbol P pasa a ser el subárbol izquierdo del nodo R.

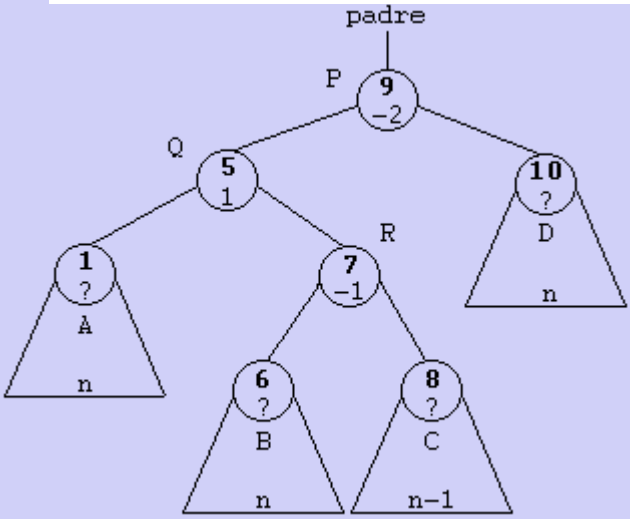


RDI pasos 1 a 3

Árbol desequilibrado a la derecha

8.7 Reequilibrados en árboles AVL

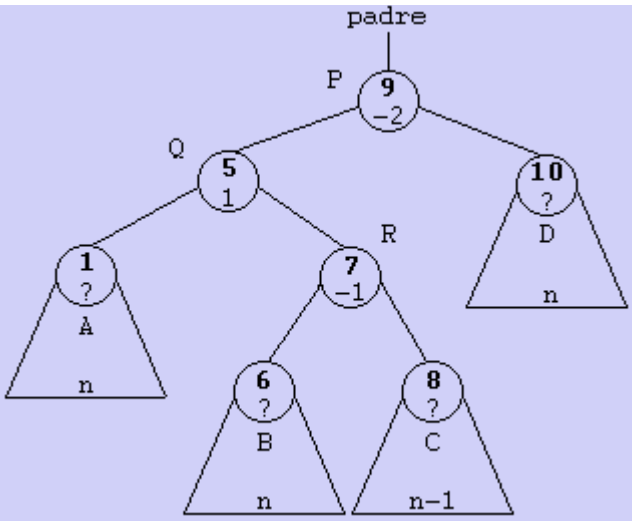
Cada vez que insertemos o eliminemos un nodo en un árbol AVL pueden suceder dos cosas: que el árbol se mantenga como AVL o que pierda esta propiedad. En el segundo caso siempre estaremos en uno de los explicados anteriormente, y recuperaremos el estado AVL aplicando la rotación adecuada.



Resultado de primera rotación

Ya comentamos que necesitamos añadir un nuevo miembro a cada nodo del árbol para averiguar si el árbol sigue siendo AVL, el Factor de Equilibrio. Cada vez que insertemos o eliminemos un nodo deberemos recorrer el camino desde ese nodo hacia el nodo raíz actualizando los valores de FE de cada nodo. Cuando uno de esos valores sea 2 ó -2 aplicaremos la rotación correspondiente.

Debido a que debemos ser capaces de recorrer el árbol en dirección a la raíz, añadiremos un nuevo puntero a cada nodo que apunte al nodo padre. Esto complicará algo las operaciones de

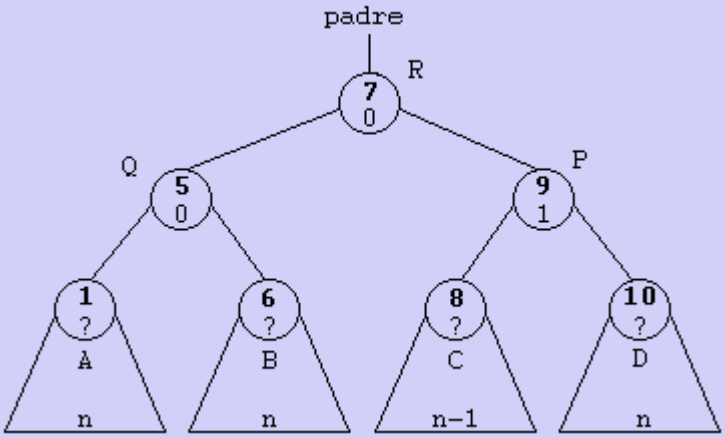


RDI pasos 4 a 6

inserción, borrado y rotación, pero facilita y agiliza mucho el cálculo del FE, y veremos que las complicaciones se compensan en gran parte por las facilidades obtenidas al disponer de este puntero.

Nota:

En rigor, no es necesario ese puntero, podemos almacenar el camino que recorremos para localizar un nodo concreto usando una pila, y después podemos usar la pila para recuperar el camino en orden inverso. Pero esto nos obliga a introducir otra estructura dinámica, y según mi opinión, complica en exceso el algoritmo.



Resultado de segunda rotación

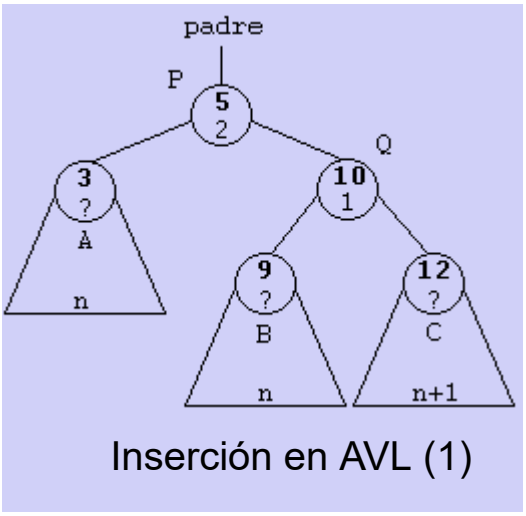
Cuando estemos actualizando los valores de FE no necesitamos calcular las alturas de las dos ramas de cada nodo, sabiendo en valor anterior de FE, y sabiendo en qué rama hemos añadido o eliminado el nodo, es fácil calcular el nuevo valor de FE. Si el nodo ha sido añadido en la rama derecha o eliminado en la izquierda, y ha habido un cambio de altura en la rama, se incrementa el valor de FE; si el nodo ha sido añadido en la rama izquierda o eliminado en

la derecha, y ha habido un cambio de altura en la rama, se decrementa el valor de FE.

Los cambios de altura en una rama se producen sólo cuando el FE del nodo raíz de esa rama ha cambiado de 0 a 1 ó de 0 a -1. En caso contrario, cuando el FE cambia de 1 a 0 ó de -1 a 0, no se produce cambio de altura.

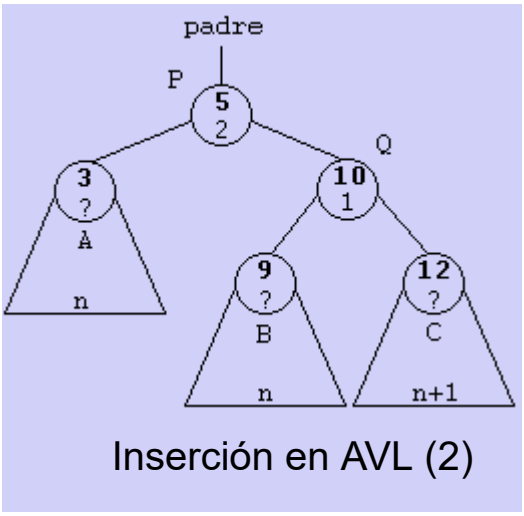
Si no hay cambio de altura, los valores de FE del resto de los nodos hasta el raíz no pueden cambiar, recordemos que el factor de equilibrio se define como la diferencia de altura entre las ramas derecha e izquierda de un nodo, la altura de la rama que no pertenece al camino no puede cambiar, puesto que sigue teniendo los mismos nodos que antes, de modo que si la altura de la rama que pertenece al camino no cambia, tampoco puede cambiar el valor de FE.

Por ejemplo, supongamos que en siguiente árbol AVL insertamos el nodo de valor 8:



Inserción en AVL (1)

Para empezar, cualquier nodo nuevo será un nodo hoja, de modo que su FE será siempre 0.



Inserción en AVL (2)

Ahora actualizamos el valor de FE del nodo padre del que acabamos de insertar (P). El valor previo es 0, y hemos añadido un nodo en su rama izquierda, por lo tanto, el nuevo valor es -1. Esto implica un cambio de altura, por lo tanto, continuamos camino hacia la raíz.

A continuación tomamos el nodo padre de P (Q), cuyo valor previo de FE era 1, y al que también hemos añadido un nodo en su rama izquierda, por lo tanto decrementamos ese valor, y el nuevo será 0. En este caso no ha incremento de altura, la altura del árbol cuya raíz es Q sigue siendo la misma, por lo tanto, ninguno de los valores de FE de los nodos hasta el raíz puede haber cambiado. Es decir, no necesitamos seguir recorriendo el camino.

Si verificamos el valor de FE del nodo R vemos que efectivamente se mantiene, puesto que tanto la altura del subárbol derecho como del izquierdo, siguen siendo las mismas.

Pero algunas veces, el valor de FE del nodo es -2 ó 2, son los casos en los que perdemos la propiedad AVL del árbol, y por lo tanto tendremos que recuperarla.

Reequilibrados en árboles AVL por inserción de un nodo

En ese caso, cuando el valor de FE de un nodo tome el valor -2 ó 2, no seguiremos el camino, sino que, con el valor de FE de el nodo actual y el del nodo derecho si FE es 2 o el del nodo izquierdo si es -2, determinaremos qué tipo de rotación debemos hacer.

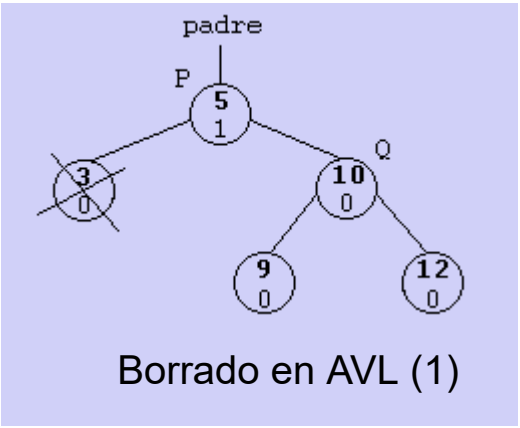
FE nodo actual	FE del nodo derecho	FE del nodo izquierdo	Rotación
-2	No importa	-1	RSD
-2	No importa	1	RDD
2	-1	No importa	RDI
2	1	No importa	RSI

El resto de los casos no nos interesan. Esto es porque en nodos desequilibrados hacia la derecha, con valores de FE positivos, siempre buscaremos el equilibrio mediante rotaciones a la izquierda, y viceversa, con nodos desequilibrados hacia la izquierda, con valores de FE negativos, buscaremos el equilibrio mediante rotaciones a la derecha.

Supongamos que el valor de FE del nodo ha pasado de -1 a -2, debido a que se ha añadido un nodo. Esto implica que el nodo

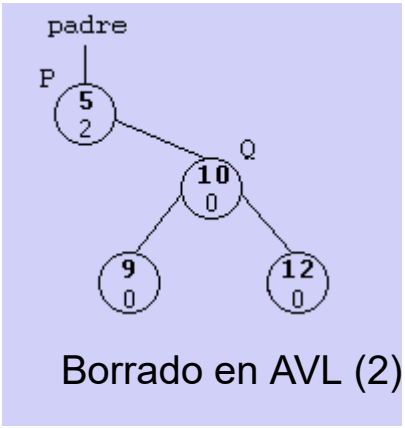
añadido lo ha sido en la rama izquierda, si lo hubiéramos añadido en la derecha el valor de FE nunca podría decrecer.

Reequilibrados en árboles AVL por borrado de un nodo



Borrado en AVL (1)

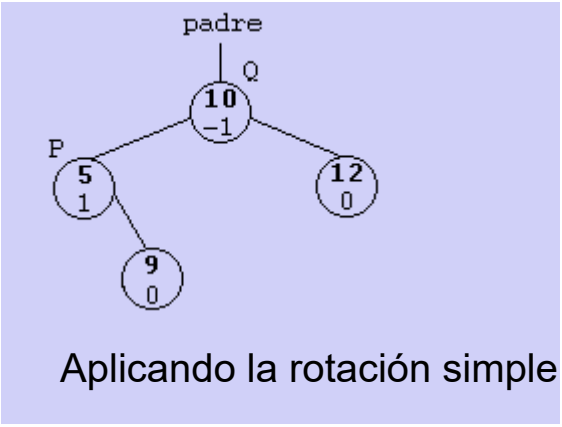
Cuando el desequilibrio se debe a la eliminación de un nodo la cosa puede ser algo diferente, pero veremos que siempre se puede llegar a uno de los casos anteriores.



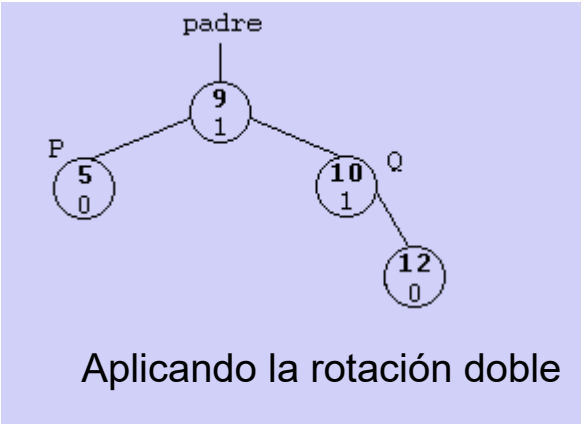
Borrado en AVL (2)

Supongamos el siguiente ejemplo, en el árbol AVL eliminaremos el nodo de valor 3:

El valor de FE del nodo P pasa de 1 a 2, sabemos que cuando el valor de FE de un nodo es 2 siempre tenemos que aplicar una rotación a izquierdas. Para saber cual de las dos rotaciones debemos aplicar miramos el valor de FE del nodo derecho. Pero en este caso, el valor de FE de ese nodo es 0. Esto no quiere decir que no podamos aplicar ninguna de las rotaciones, por el contrario, podremos aplicar cualquiera de ellas. Aunque por economía, lo razonable es aplicar la rotación simple.



Aplicando la rotación simple



Aplicando la rotación doble

Del mismo modo, el valor de FE del nodo derecho podría haber sido 1 ó -1, en ese caso sí está determinado el tipo de rotación a realizar.

El razonamiento es similar cuando se eliminan nodos y el resultado es que se obtiene un nodo con FE de -2, en este caso se realizará una rotación a derechas, y la rotación dependerá del valor de FE del nodo izquierdo al que muestra el desequilibrio. Si es 0 ó -1 haremos una rotación simple, si es 1, haremos una rotación doble.

Tendremos entonces una tabla más general para decidir la rotación a aplicar:

FE nodo actual	FE del nodo derecho	FE del nodo izquierdo	Rotación
-2	No importa	-1	RSD
-2	No importa	0	RSD
-2	No importa	1	RDD
2	-1	No importa	RDI
2	0	No importa	RSI
2	1	No importa	RSI

Los árboles AVL siempre quedan equilibrados después de una rotación

Esto puede comprobarse analizando los métodos de rotación que hemos estudiado, después de efectuada la rotación, la altura del árbol cuya raíz es el nodo rotado se mantiene, por lo tanto, no necesitamos continuar el camino hacia la raíz: sabemos que el árbol es AVL.

8.8 Algoritmos

De inserción de nodo

En general, la inserción de nodos en un árbol AVL es igual que en un árbol ABB, la diferencia es que en un árbol AVL, después de insertar el nodo debemos recorrer el árbol en sentido hacia la raíz, recalculando los valores de FE, hasta que se cumpla una de estas condiciones: que lleguemos a la raíz, que se encuentre un nodo con valor de FE de 2, ó -2, o que se llegue a un nodo cuyo FE no cambie o decrezca en valor absoluto, es decir, que cambie de 1 a 0 ó de -1 a 0.

Podemos considerar que el algoritmo de inserción de nodos en árboles AVL es una ampliación del que vimos para árboles ABB.

De borrado de nodo

Lo mismo pasa cuando se eliminan nodos, el algoritmo es el mismo que en árboles ABB, pero después de eliminar el nodo debemos recorrer el camino hacia la raíz recalculando los valores de FE, y equilibrando el árbol si es necesario.

De recalcular FE

Ya comentamos más atrás que para seguir el camino desde el nodo insertado o borrado hasta el nodo raíz tenemos dos alternativas:

1. Guardar en una pila los punteros a los nodos por los que hemos pasado para llegar al nodo insertado o borrado, es decir, almacenar el camino.
2. Añadir un nuevo puntero a cada nodo que apunte al padre del nodo actual. Esto nos permite recorrer el árbol en el sentido contrario al normal, es decir, en dirección a la raíz.

Para calcular los nuevos valores de FE de los nodos del camino hay que tener en cuenta los siguientes hechos:

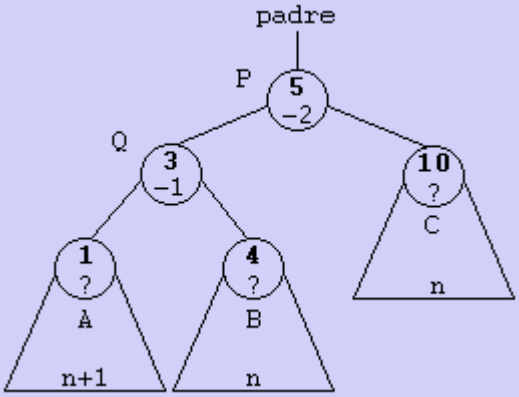
- El valor de FE de un nodo insertado es cero, ya que siempre insertaremos nodos hoja.

- Si el nuevo valor de FE para cualquiera de los siguientes nodos del camino es cero, habremos terminado de actualizar los valores de FE, ya que la rama mantiene su altura, la inserción o borrado del nodo no puede influir en los valores de FE de los siguientes nodos del camino.
- Cuando se elimine un nodo pueden pasar dos cosas. Siempre eliminamos un nodo hoja, ya que cuando no lo es, lo intercambiamos con un nodo hoja antes de eliminarlo. Pero algunas veces, el nodo padre del nodo eliminado se convertirá a su vez en nodo hoja, y en ese caso no siempre hay que dar por terminada la actualización del FE del camino. Por lo tanto, cuando eliminemos un nodo, actualizaremos el valor de FE del nodo padre y continuaremos el camino, independientemente del valor de FE calculado.
- A la hora de actualizar el valor de FE de un nodo, tenemos que distinguir cuando el equilibrado sea consecuencia de una inserción o lo sea de una eliminación. Incrementaremos el valor de FE del nodo si la inserción fue en la rama derecha o si la eliminación fue en la rama izquierda, decrementaremos si la inserción fue en la izquierda o la eliminación en la derecha.
- Si en valor de FE es -2, haremos una rotación doble a la derecha su el valor de FE del nodo izquierdo es 1, y simple si es 1 ó 0.
- Si en valor de FE es 2, haremos una rotación doble a la izquierda su el valor de FE del nodo izquierdo es -1, y simple si es -1 ó 0.
- En cualquiera de los dos casos, podremos dar por terminado el recorrido del camino, ya que la altura del árbol cuya raíz es un nodo rotado no cambia.
- En cualquier otro caso, seguiremos actualizando hasta llegar al nodo raíz.

De rotación simple

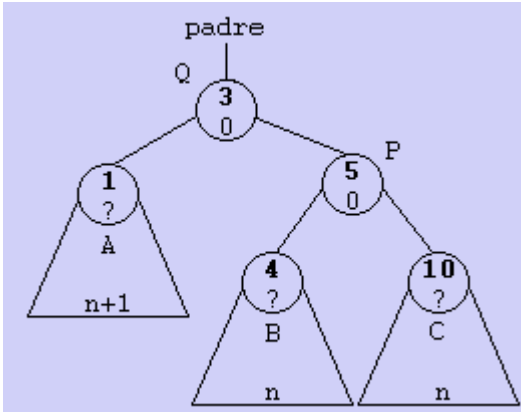
A la hora de implementar los algoritmos que hemos visto para rotaciones simples tenemos dos opciones: seguir literalmente los pasos de los gráficos, o tomar un atajo, y hacerlo mediante

asignaciones. Nosotros lo haremos del segundo modo, ya que resulta mucho más rápido y sencillo.



Árbol desequilibrado a la izquierda

Primero haremos las reasignaciones de punteros, de modo que el árbol resultante responda a la estructura después de la rotación. Después actualizaremos los punteros al nodo padre para los nodos que han cambiado de posición. Por último actualizaremos los valores de FE de esos mismos nodos.



Árbol equilibrado

Para la primera fase usaremos punteros auxiliares a nodo, que en el caso de rotación a la derecha necesitamos un puntero P al nodo con FE igual a -2. Ese será el parámetro de entrada, otro puntero al nodo izquierdo de P: Q. Y tres punteros más a los árboles A, B y C.

En realidad, si nos fijamos en los gráficos, los punteros a A y C no son necesarios, ya que ambos conservan sus posiciones, A sigue siendo el subárbol izquierdo de Q y C el subárbol derecho de P.

Usaremos otro puntero más: Padre, que apunte al padre de P. Disponiendo de los punteros Padre, P, Q y B, realizar la rotación es muy sencillo:

```
if(Padre)
  if(Padre->derecho == P) Padre->derecho = Q;
  else Padre->izquierdo = Q;
else raíz = Q;
```

```
    // Reconstruir árbol:
    P->izquierdo = B;
    Q->derecho = P;
```

Hay que tener en cuenta que P puede ser la raíz de un subárbol derecho o izquierdo de otro nodo, o incluso la raíz del árbol completo. Por eso comprobamos si P tiene padre, y si lo tiene, cual de sus ramas apunta a P, cuando lo sabemos, hacemos que esa rama apunte a Q. Si Padre es NULL, entonces P era la raíz del árbol, así que hacemos que la nueva raíz sea Q.

Sólo nos queda trasladar el subárbol B a la rama izquierda de P, y Q a la rama derecha de P.

La segunda fase consiste en actualizar los punteros padre de los nodos que hemos cambiado de posición: P, B y Q.

```
    P->padre = Q;
    if(B) B->padre = P;
    Q->padre = Padre;
```

El padre de P es ahora Q, el de Q es Padre, y el de B, si existe es P.

La tercera fase consiste en ajustar los valores de FE de los nodos para los que puede haber cambiado.

Esto es muy sencillo, después de una rotación simple, los únicos valores de FE que cambian son los de P y Q, y ambos valen 0.

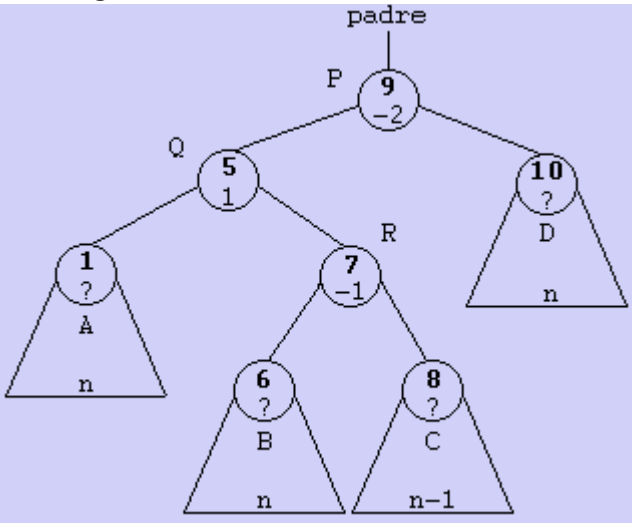
```
// Rotación simple a derechas
void RSD(Nodo* nodo) {
    Nodo *Padre = nodo->padre;
    Nodo *P = nodo;
    Nodo *Q = P->izquierdo;
    Nodo *B = Q->derecho;

    if(Padre)
      if(Padre->derecho == P) Padre->derecho = Q;
```

```
      else Padre->izquierdo = Q;
    else raíz = Q;

    // Reconstruir árbol:
    P->izquierdo = B;
    Q->derecho = P;

    // Reasignar padres:
    P->padre = Q;
    if(B) B->padre = P;
    Q->padre = Padre;

    // Ajustar valores de FE:
    P->FE = 0;
    Q->FE = 0;
}
```

La rotación a izquierdas es simétrica.

De rotación doble

Para implementar las rotaciones dobles trabajaremos de forma análoga.

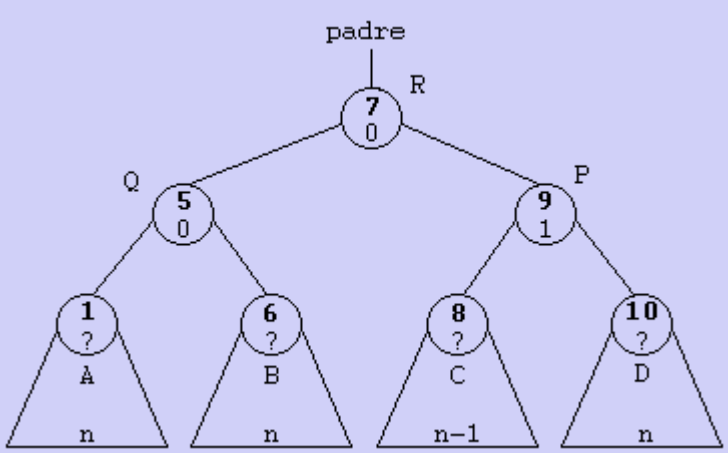


Árbol desequilibrado a la izquierda

Primero haremos las reasignaciones de punteros, de modo que el árbol resultante responda a la estructura después de la rotación. Después actualizaremos los punteros al nodo padre para los nodos que han cambiado de posición. Por último actualizaremos los valores de FE de esos mismos nodos.

Para la primera fase usaremos punteros auxiliares a nodo, que en el caso de rotación a la derecha necesitamos un puntero P al

nodo con FE igual a -2. Ese será el parámetro de entrada, otro puntero al nodo izquierdo de P: Q. Un tercero al nodo derecho de Q: R. Y cuatro punteros más a los árboles A, B, C y D.



Árbol equilibrado

En realidad, si nos fijamos en los gráficos, los punteros a A y D no son necesarios, ya que ambos conservan sus posiciones, A sigue siendo el subárbol izquierdo de Q y D el subárbol derecho de P.

También en este caso usaremos otro puntero más: Padre, que apunte al padre de P. Disponiendo de los punteros Padre, P, Q, R, B y C, realizar la rotación es muy sencillo:

```
if(Padre)
  if(Padre->derecho == nodo) Padre->derecho = R;
  else Padre->izquierdo = R;
else raíz = R;

// Reconstruir árbol:
Q->derecho = B;
P->izquierdo = C;
R->izquierdo = Q;
R->derecho = P;
```

Ahora también hay que tener en cuenta que P puede ser la raíz de un subárbol derecho o izquierdo de otro nodo, o incluso la raíz del árbol completo. Por eso comprobamos si P tiene padre, y si lo tiene, cual de sus ramas apunta a P, cuando lo sabemos, hacemos que esa rama apunte a R. Si Padre es NULL, entonces P era la raíz del árbol, así que hacemos que la nueva raíz sea R.

Sólo nos queda trasladar el subárbol B a la rama derecha de Q, C a la rama izquierda de P, Q a la rama izquierda de R y P a la rama derecha de R.

La segunda fase consiste en actualizar los punteros padre de los nodos que hemos cambiado de posición: P, Q, R, B y C.

```
R->padre = Padre;
P->padre = Q->padre = R;
if(B) B->padre = Q;
if(C) C->padre = P;
```

El padre de R es ahora Padre, el de P y Q es R, y el de B, si existe es Q, y el de C, si existe, es P.

La tercera fase consiste en ajustar los valores de FE de los nodos para los que puede haber cambiado.

En las rotaciones dobles esto se complica un poco ya que puede suceder que el valor de FE de R antes de la rotación sea -1, 0 o 1. En cada caso, los valores de FE de P y Q después de la rotación serán diferentes.

```
// Ajustar valores de FE:
switch(R->FE) {
   case -1: Q->FE = 0; P->FE = 1; break;
   case 0:  Q->FE = 0; P->FE = 0; break;
   case 1:  Q->FE = -1; P->FE = 0; break;
}
R->FE = 0;
```

Si la altura de B es n-1 y la de C es n, el valor de FE de R es 1. Después de la rotación, la rama B pasa a ser el subárbol derecho de Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será -1.

Si la altura de B es n y la de C es n-1, el valor de FE de R es -1. Después de la rotación, la rama B pasa a ser el subárbol derecho de

Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será 0.

Por último, si la altura de B y C es n, el valor de FE de R es 0. Después de la rotación, la rama B pasa a ser el subárbol derecho de Q, por lo tanto, la FE de Q, dado que la altura de su rama izquierda es n, será 0. La rama C pasa a ser el subárbol izquierdo de P, y dado que la altura de la rama derecha es n, la FE de P será 0.

```
// Rotación doble a derechas
void RDD(Nodo* nodo) {
   Nodo *Padre = nodo->padre;
   Nodo *P = nodo;
   Nodo *Q = P->izquierdo;
   Nodo *R = Q->derecho;
   Nodo *B = R->izquierdo;
   Nodo *C = R->derecho;

   if(Padre)
     if(Padre->derecho == nodo) Padre->derecho = R;
     else Padre->izquierdo = R;
   else raíz = R;

   // Reconstruir árbol:
   Q->derecho = B;
   P->izquierdo = C;
   R->izquierdo = Q;
   R->derecho = P;

   // Reasignar padres:
   R->padre = Padre;
   P->padre = Q->padre = R;
   if(B) B->padre = Q;
   if(C) C->padre = P;

   // Ajustar valores de FE:
   switch(R->FE) {
      case -1: Q->FE = 0; P->FE = 1; break;
      case 0:  Q->FE = 0; P->FE = 0; break;
      case 1:  Q->FE = -1; P->FE = 0; break;
   }
   R->FE = 0;
}
```

8.9 Ejemplo de árbol AVL en C

No ha demasiado que añadir, construiremos los ejemplos de árboles AVL basándonos en los ejemplos que hicimos para árboles binarios de búsqueda.

Sólo tenemos que añadir cinco nuevas funciones: una para equilibrar el árbol, y cuatro para las cuatro posibles rotaciones.

Además, modificaremos ligeramente las funciones de inserción y borrado para que se equilibre el árbol automáticamente después de cada inserción o borrado.

En la estructura de nodo para árbol AVL añadiremos un puntero al nodo padre y un valor entero para almacenar el factor de equilibrio.

Cuando se inserten nuevos nodos hay que ajustar los valores de los nuevos miembros de nodo: FE y padre. Seguidamente, llamaremos a la función "Equilibrar", salvo que el nodo insertado sea el raíz, ya que en ese caso no es necesario, evidentemente.

El procedimiento de equilibrar consiste en la implementación del algoritmo que vimos en el punto anterior:

```
/* Equilibrar árbol AVL partiendo de un nodo*/
void Equilibrar(Arbol *a, pNodo nodo, int rama, int nuevo) {
   int salir = FALSE;

   /* Recorrer camino inverso actualizando valores de FE: */
   while(nodo && !salir) {
      if(nuevo)
         if(rama == IZQUIERDO) nodo->FE--; /* Depende de si
añadimos ... */
         else                  nodo->FE++;
      else
         if(rama == IZQUIERDO) nodo->FE++; /* ... o borramos
*/
         else                  nodo->FE--;
      if(nodo->FE == 0) salir = TRUE; /* La altura de las
rama que
```

```
                                          empieza en nodo no
ha variado,
                                          salir de Equilibrar
*/
      else if(nodo->FE == -2) { /* Rotar a derechas y salir:
*/
         if(nodo->izquierdo->FE == 1) RDD(a, nodo); /*
Rotación doble  */
         else RSD(a, nodo);                          /*
Rotación simple */
         salir = TRUE;
      }
      else if(nodo->FE == 2) {  /* Rotar a izquierdas y
salir: */
         if(nodo->derecho->FE == -1) RDI(a, nodo); /*
Rotación doble  */
         else RSI(a, nodo);                        /*
Rotación simple */
         salir = TRUE;
      }
      if(nodo->padre)
         if(nodo->padre->derecho == nodo) rama = DERECHO;
else rama = IZQUIERDO;
      nodo = nodo->padre; /* Calcular FE, siguiente nodo del
camino. */
   }
}
```

Las funciones para rotar son también sencillas, por ejemplo, la rotación simple a la derecha:

```
/* Rotación simple a derechas */
void RSD(Arbol *a, pNodo nodo) {
   pNodo Padre = nodo->padre;
   pNodo P = nodo;
   pNodo Q = P->izquierdo;
   pNodo B = Q->derecho;

   if(Padre)
     if(Padre->derecho == P) Padre->derecho = Q;
     else Padre->izquierdo = Q;
   else *a = Q;

   /* Reconstruir árbol: */
```

```
   P->izquierdo = B;
   Q->derecho = P;

   /* Reasignar padres: */
   P->padre = Q;
   if(B) B->padre = P;
   Q->padre = Padre;

   /* Ajustar valores de FE: */
   P->FE = 0;
   Q->FE = 0;
}
```

Y la rotación doble a la derecha:

```
/* Rotación doble a derechas */
void RDD(Arbol *raíz, Arbol nodo) {
   pNodo Padre = nodo->padre;
   pNodo P = nodo;
   pNodo Q = P->izquierdo;
   pNodo R = Q->derecho;
   pNodo B = R->izquierdo;
   pNodo C = R->derecho;

   if(Padre)
     if(Padre->derecho == nodo) Padre->derecho = R;
     else Padre->izquierdo = R;
   else *raíz = R;

   /* Reconstruir árbol: */
   Q->derecho = B;
   P->izquierdo = C;
   R->izquierdo = Q;
   R->derecho = P;

   /* Reasignar padres: */
   R->padre = Padre;
   P->padre = Q->padre = R;
   if(B) B->padre = Q;
   if(C) C->padre = P;

   /* Ajustar valores de FE: */
   switch(R->FE) {
      case -1: Q->FE = 0; P->FE = 1; break;
      case 0:  Q->FE = 0; P->FE = 0; break;
```

```
        case 1:  Q->FE = -1; P->FE = 0; break;
    }
    R->FE = 0;
}
```

8.10 Ejemplo de árbol AVL en C++

Usando clases el ejemplo también está basado en el que hicimos para árboles binarios de búsqueda, añadiendo las cinco funciones anteriores: Equilibrar, RSD, RSI, RDD y RDI. Además de las modificaciones en Insertar y Borrar.

8.11 Ejemplo de árbol AVL en C++ con plantillas

Usando plantillas no hay más que generalizar el ejemplo de clases, cambiando en tipo de dato a almacenar por el parámetro de la plantilla.

Fichero con el código fuente

Capítulo 9 Vectores

9.1 Definición

Los vectores son *arrays* de una dimensión.

Aunque C y C++ disponen de estructuras para crear arrays de una dimensión como parte de su sintáxis básica, estas estructuras tienen una limitación importante: que hay que elegir su tamaño en la fase de diseño, no pudiendo variar a lo largo de la ejecución del programa.

Por supuesto, podemos usar una estructura dinámica, reservando memoria y usando un puntero como si se tratase de un *array*, pero esto también tiene sus limitaciones. Por ejemplo, sigue siendo necesario especificar un tamaño máximo

En otros lenguajes, es posible añadir y eliminar elementos de un *array* durante la ejecución, sin preocuparse de si hay o no espacio para hacerlo. En C y C++ también, claro, pero es necesario crear TADs específicos para esta tarea.

9.2 Métodos

En las bibliotecas de TDAs, se añaden muchos métodos a los vectores, con el fin de hacerlos lo más genéricos y flexibles posible.

En un vector debemos distinguir entre capacidad y tamaño.

Capacidad es el número máximo de elementos que el vector puede contener en un momento dado.

Medida es el número de elementos que el vector contiene realmente. Este valor será siempre menor o igual que el de capacidad.

Los métodos más básicos e imprescindibles, que implementaremos en nuestros ejemplos, son:

- Reservar (reserve): Cambia la capacidad (número de elementos) del vector, en general, este valor no puede disminuir durante la ejecución del programa. De modo que el valor suministrado es la capacidad mínima del vector.
- Redimensionar (resize): Modifica el tamaño o medida del vector.
- Medida (size): Devuelve el tamaño (número de elementos) actual del vector.
- Vacio (empty): Devuelve un valor verdadero o distinto de cero si el vector está vacío
- Capacidad (capacity): Puede haber espacio sin usar, de modo que la capacidad puede ser mayor o igual que el valor de *Medida*.
- [] o En (at): Operador de acceso, obtiene una referencia al elemento indicado.
- Agregar (push_back): Añade un elemento al final del vector.
- Sustraer (pop_back): Elimina un elemento del final del vector.
- Primero (front): Obtiene una referencia al primer elemento.
- Ultimo (back): Obtiene una referencia al último elemento.
- Insertar (insert): Inserta un elemento en la posición anterior a la dada. El elemento insertado ocupará la posición indicada.
- Eliminar (erase): Elimina un elemento del vector.
- Borrar (clear): Elimina todos los elementos.

9.3 Implementación de un vector

Hay muchas formas de implementar vectores, dependiendo de la eficiencia que se pretenda conseguir, ya sea en la velocidad de ejecución o en el manejo de la memoria.

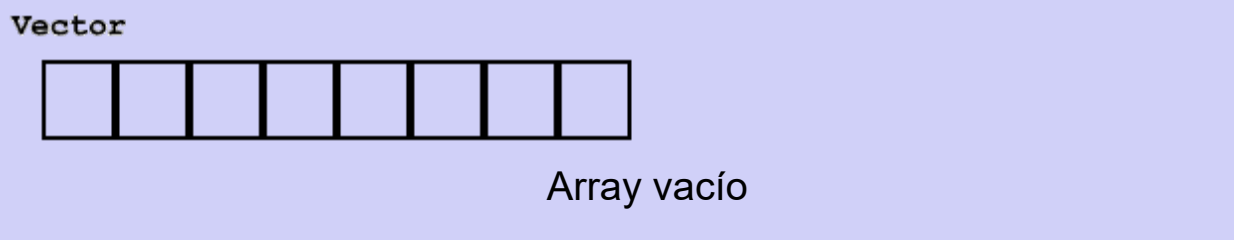
Las operaciones más costosas, en términos de tiempo de ejecución, son las implican cambiar el tamaño de la estructura en memoria: *Redimensionar*, *Ampliar*, *Reducir*, *Insertar* o *Eliminar* elementos.

Una condición para que un vector lo sea, la más evidente, es que tiene que almacenarse en un bloque de memoria con direcciones contiguas. Esto nos facilita el acceso a los elementos, ya que es sencillo acceder a uno en particular. Además, estos vectores son *compatibles* con los tipos estructurados de los que disponemos en C y C++. El problema de esta solución es que nos obliga a reubicar el vector cada vez que sea necesario añadir elementos. Generalmente esto implica reservar un nuevo bloque de memoria, copiar todos los elementos actuales a la nueva ubicación, y borrar el bloque anterior.

Una primera optimización para evitar este inconveniente, al menos en parte, consiste en ampliar y reducir el número de elementos en bloques de memoria con capacidad para varios de ellos. Esto explica porqué tenemos dos métodos con finalidades parecidas: *Medida* y *Capacidad*. Si cada vez que nos quedamos sin sitio para añadir nuevos elementos, ampliamos el vector en una cantidad (en principio desconocida), el valor devuelto por *Capacidad* siempre será mayor o igual al devuelto por *Medida*.

En la práctica se suele hacer que cada vez que se debe ampliar el vector, se duplica el número de elementos actual. Sin embargo, esto no es más que una de las posibles soluciones, también podríamos añadir un número constante y predefinido de elementos.

Por ejemplo, supongamos que hemos creado un vector para contener 8 enteros:

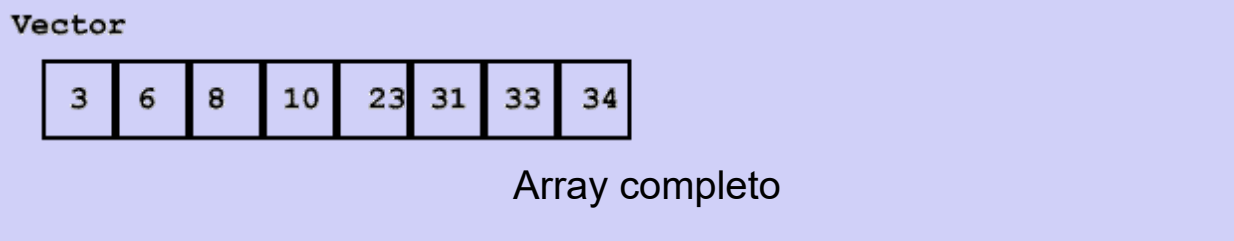


Array vacío

Si aplicamos el método *Medida* sobre este vector, el resultado será 0. Si aplicamos el método *Capacidad* el resultado será 8.

Durante la ejecución del programa vamos añadiendo elementos al vector: 3, 6, 8, 10, 23, 31, 33 y 34; hasta ocupar toda la capacidad del vector:

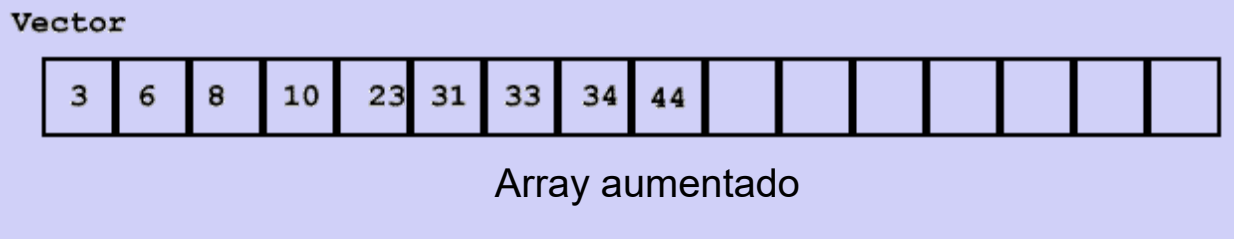
Ahora, tanto el método *Medida* como *Capacidad* devuelven el



Array completo

mismo valor: 8.

Si intentamos insertar un nuevo valor, 44, antes deberemos ampliar la capacidad del vector. En este caso, duplicamos el número de elementos que podemos almacenar hasta 16, e insertamos el nuevo valor:



Array aumentado

Ahora, para el método *Cuenta* obtenemos el valor 9, y para *Capacidad* 16.

En el diseño del vector deberemos tomar algunas decisiones que afectan a la integridad de la estructura. Por ejemplo, ¿qué debemos hacer si el usuario intenta asignar un valor a un elemento cuyo índice sea mayor que medida y menor que capacidad?

¿Y si el índice indicado es mayor que capacidad?

¿Y si el valor del índice es muy grande?

¿Y si se indica un índice negativo?

Podríamos sencillamente obviar los límites y acceder a esas posiciones sin verificar nada, como pasa con los arrays normales de C y C++. Pero también podemos implementar un funcionamiento más seguro, de modo que los programas no dejen de funcionar en esos casos al intentar modificar posiciones de memoria que no pertenecen a la estructura de datos.

9.4 Descripción de los métodos

Veamos ahora en qué consiste cada uno de los métodos propuestos.

Pero antes, para empezar, necesitamos algunos valores referentes al vector, que debemos mantener actualizados:

- primero: un puntero al primer elemento del vector.
- capacidad: capacidad actual del *array*, número máximo de elementos que puede almacenar.
- contador: número de elementos usados actualmente.

También responderemos a las preguntas que planteamos anteriormente sobre qué debe hacer nuestro vector cuando se accede a elementos fuera de él.

Aunque pueda parecer tentador, no implementaremos rutinas de seguridad para verificar los márgenes. El motivo es que haciendo eso penalizaríamos la eficiencia de la estructura, y perjudicamos al usuario que tiene cuidado de no sobrepasar los límites del vector, sobre aquellos que son más descuidados.

Reservar

Este método se refiere a la capacidad del vector, es decir, al número máximo de elementos que puede contener.

Como cuando creamos vectores usando tipos agregados, nuestro vector necesitará un valor que indique el número de elementos inicial en la declaración. Cuando encapsulemos el TDA vector en una clase o plantilla esta tarea la realizará el constructor.

Así, si declaramos un vector de 30 elementos, nuestro TDA puede reservar espacio para 30 ó más elementos, y el valor de contador se iniciará a 30. Es decir, podremos acceder al menos a 30 elementos, pero la capacidad podría ser mayor.

Si usamos este método después de la declaración del vector, el valor indicado se interpreta como la nueva capacidad mínima del vector. Si ese valor es menor que la capacidad actual, el método no hace nada. Si es mayor implica una reubicación de la memoria. Se

reserva un nuevo bloque suficientemente grande para la nueva capacidad, se copia el contenido del bloque anterior y se libera esa memoria.

Redimensionar

También podemos usar este método durante la ejecución para aumentar o disminuir el tamaño, el número de elementos, del vector.

Si usamos este método podemos distinguir dos casos:

- El nuevo valor es mayor que el de la capacidad actual: en ese caso la capacidad se aumenta al nuevo valor. Esto implica reservar un bloque de memoria con la nueva capacidad, copiar los elementos desde el bloque anterior, y liberar esa memoria.
- El nuevo valor es menor que el de la capacidad actual: en ese caso la capacidad se disminuye al valor indicado. Esto también implica una reubicación de la memoria del vector. Si la nueva capacidad es menor que el número de elementos ya almacenados, se conservan los primeros y los sobrantes se pierden.

Medida

Este método es sencillo, sólo debe devolver el valor del *contador*.

Vacio

También es un método simple, devolverá un valor verdadero o distinto de cero si *contador* es cero.

Capacidad

Devolverá el valor de *capacidad*.

[] o En

En una implementación en C usaremos una función *En* para acceder a la posición indicada, en una implementación C++ sobrecaragaremos el operador [].

Agregar

Añade un elemento al final del vector. Si esta operación implica un cambio de capacidad se producirá una reubicación de los datos.

Sustraer

Elimina un elemento del final del vector

Primero

Obtiene una referencia al primer elemento. Esto sólo en C++. Hay un caso especial, cuando la medida es cero. En ese caso, dado que tenemos que devolver una referencia, tendremos que decidir qué referencia retornar.

Ultimo

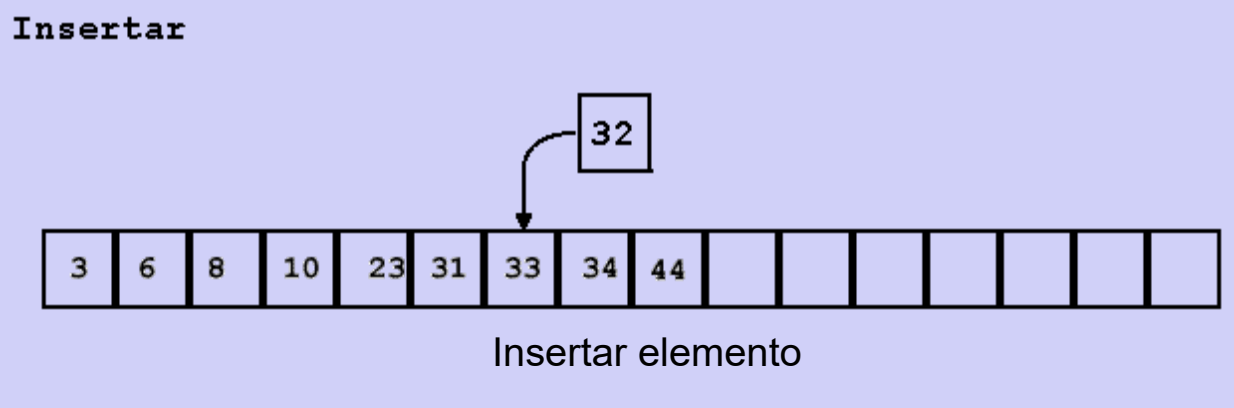
Obtiene una referencia al último elemento. Sólo en C++. Como en el caso anterior, también hay un caso especial, cuando la medida es cero. En ese caso también tendremos que decidir qué referencia retornar.

Insertar

Inserta un elemento en la posición anterior a la dada. El elemento insertado ocupará la posición indicada.

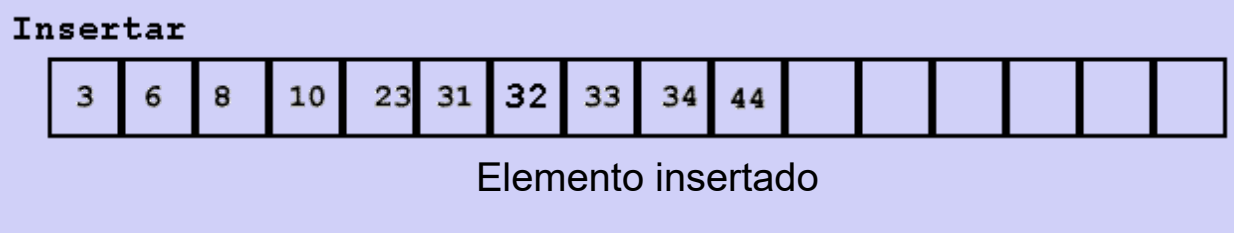
Por ejemplo, insertaremos el elemento '32' en la posición 6, es decir, en la que actualmente ocupa el elemento '33':

Para hacerlo necesitamos desplazar todos los elementos a partir de la posición 6 a la siguiente posición, dejando un hueco para



Insertar elemento

insertar el nuevo:



Elemento insertado

Si la inserción de un nuevo elemento implica aumentar la capacidad del vector se producirá una reubicación de los datos.

Eliminar

Elimina un elemento del vector. Se trata de la operación inversa a *Insertar*.

Borrar

Elimina todos los elementos del vector, dejando la capacidad intacta.

9.5 Inconvenientes sobre tipos estructurados

El principal inconveniente es que la dirección de cualquier elemento del array puede cambiar durante la ejecución del programa.

Con un array normal, la dirección de memoria de un elemento cualquiera se mantiene desde la declaración del array hasta su

destrucción. De modo que si conservamos una referencia a un elemento cualquiera usando un puntero, existe una relación unívoca entre ambos.

Con un array del tipo que estamos diseñando, creado mediante clases o plantillas, no podemos asumir que la dirección de un elemento se mantendrá constante a lo largo del programa. A medida que nos quedamos sin espacio para añadir elementos, no sólo redimensionaremos el array, sino que en general, cambiaremos la ubicación en memoria de sus elementos.

De este modo, si mantenemos un puntero a un elemento del array, no podremos estar seguros de que apuntará al mismo elemento después de añadir más valores nuevos durante la ejecución.

9.6 Implementación en C

En este tipo de estructuras no trabajaremos con nodos, como en las anteriores. Todos los datos de un vector se almacenan en direcciones consecutivas de memoria, de modo que sólo necesitamos un bloque de memoria obtenido de forma dinámica.

```
typedef struct {
    int *datos;
        int capacidad;
        int medida;
} stvector;
```

Hay una operación especial que realizaremos con frecuencia en nuestros vectores, cada vez que nos quedemos sin espacio o sea necesario modificar la capacidad. Me refiero a la reubicación de la memoria en la que se almacenan los datos.

Esta función debe realizar las siguientes tareas:

1. Obtener un nuevo bloque de memoria con la nueva capacidad especificada.
2. Copiar los elementos desde la ubicación anterior a la nueva. Hay que tener en cuenta que si la nueva tiene menor capacidad, sólo se copiarán los elementos que quepan en la nueva ubicación.
3. Liberar la memoria de la antigua ubicación.
4. Hacer que la ubicación del vector sea la nueva.

Programa completo en C

```
/*
 * TDA Vector C
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */

#include <stdio.h>
#include <stdlib.h>

int min(a,b) {
    return (a<b) ? a : b;
}

/* Declaración de estructura para implementar un vector */
typedef struct {
    int *datos;
        int capacidad;
        int medida;
} stvector;

void Reservar(stvector *v, int nt);

/* Función auxiliar para reubicar los datos a una nueva zona
de memoria */
void Reubicar(stvector *v, int nc) {
    int *datos2;
    int i;

    datos2 = (int*)malloc(nc*sizeof(int));
    for(i = 0; i < min(v->capacidad, nc); i++) datos2[i] =
v->datos[i];
    if(v->datos) free(v->datos);
```

```
    v->datos = datos2;
}

/* Inicia el vector con una capacidad cap */
void Iniciar(stvector *v, int cap) {
    v->datos = NULL;
    v->capacidad = v->medida = 0;
    Reservar(v, cap);
}

/* Libera la memoria asignada al vector */
void Liberar(stvector *v) {
    free(v->datos);
    v->datos = NULL;
    v->capacidad = v->medida = 0;
}

/* Aumenta la capacidad del vector a la nueva capacidad ncap
*/
void Reservar(stvector *v, int ncap) {
    if(ncap > v->capacidad) Reubicar(v, ncap);
    v->capacidad = ncap;
}

/* Aumenta la capacidad del vector para que pueda almacenar
al menos nm elementos */
void Redimensionar(stvector *v, int nm) {
    int cap = v->capacidad;
    while(nm > cap) cap *= 2;
    Reubicar(v, cap);
    v->capacidad = cap;
}

/* Retorna un puntero al elemento n del vector */
int *En(stvector *v, int n) {
    return &(v->datos[n]);
}

/* Retorna un puntero al primer elemento del vector */
int *Vector(stvector *v) {
    return v->datos;
}

/* Añade un elemento de valor d al final del vector */
void Agregar(stvector *v, int d) {
    if(v->medida == v->capacidad) Redimensionar(v, v-
>medida+1);
```

```
    v->datos[v->medida++] = d;
}

/* Elimina el último elemento del vector, y devuelve su
valor */
int Sustraer(stvector *v) {
    if(v->medida > 0) return v->datos[--v->medida];
    else return 0;
}

/* Asigna el valor dato a los primeros cuenta elementos del
vector */
void Asignar(stvector *v, int cuenta, int dato) {
    int i;
    for(i = 0; i < cuenta; i++) Agregar(v, dato);
}

/* Devuelve el valor del primer elemento del vector */
int Primero(stvector *v) {
    if(v->medida > 0) return v->datos[0];
    else return 0;
}

/* Devuelve el valor del último elemento del vector */
int Ultimo(stvector *v) {
    if(v->medida > 0) return v->datos[v->medida-1];
    else return 0;
}

/* Inserta el valor val en la posición pos del vector */
void Insertar(stvector *v, int pos, int val) {
    int i;

    if(v->medida == v->capacidad) Redimensionar(v, v-
>medida+1);
    for(i = v->medida; i > pos; i--) v->datos[i] = v-
>datos[i-1];
    v->datos[pos] = val;
    v->medida++;
}

/* Elimina el valor de la posición pos del vector */
void Eliminar(stvector *v, int pos) {
    int i;

    if(pos < v->medida) {
        for(i = pos; i < v->medida-1; i++) v->datos[i] = v-
>datos[i+1];
```

```
        v->medida--;
    }
}

/* Borra todos los elementos del vector */
void Borrar(stvector *v) {
    v->medida = 0;
}

/* Devuelve el número de elementos del vector */
int Medida(stvector *v) { return v->medida; }
/* Devuelve un valor no nulo si el vector está vacío */
int Vacio(stvector *v) { return !v->medida; }
/* Devuelve la capacidad actual del vector */
int Capacidad(stvector *v) { return v->capacidad; }


int main() {
    stvector v;
    int i;

    Iniciar(&v, 10);
    Asignar(&v, 10, 0);

    for(i = 0; i < 10; i++) Vector(&v)[i] = i;
    for(i = 0; i < 10; i++) Agregar(&v, 100+i);
    Agregar(&v, 23);

    Insertar(&v, 15, 1000);
    Eliminar(&v, 16);
    Eliminar(&v, 108);

    for(i = 0; i < Medida(&v); i++)
        printf("v[%d] = %d\n", i, Vector(&v)[i]);
    *En(&v, 3) = 30;
    printf("v[3] = %d\n", *En(&v, 3));

    printf("Primero = %d\n", Primero(&v));
    printf("Ultimo = %d\n", Ultimo(&v));

    printf("Medida = %d\n", Medida(&v));
    printf("v[ultimo] = %d\n", Sustraer(&v));
    printf("Medida = %d\n", Medida(&v));
    printf("v[ultimo] = %d\n", Sustraer(&v));
    printf("Medida = %d\n", Medida(&v));
    printf("Capacidad = %d\n", Capacidad(&v));
    Liberar(&v);
```

```
    return 0;
}
```

9.7 Implementación en C++

```
/*
 * TDA Vector C++
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */

#include <iostream>

using namespace std;

class vector {
    public:
        vector(int m);
        vector(vector& v);
        ~vector();
        void Reservar(int ncap);
        void Redimensionar(int nm);
        int &operator[](int n);
        void Agregar(int d);
        int Sustraer();
        void Asignar(int cuenta, int dato);
        int &Primero();
        int &Ultimo();
        void Insertar(int pos, int val);
        void Eliminar(int pos);
        /* Borra todos los elementos del vector */
        void Borrar() { medida = 0; }
        /* Devuelve el número de elementos del vector */
        int Medida() { return medida; }
        /* Devuelve un valor true si el vector está vacío */
        bool Vacio() { return !medida; }
        /* Devuelve la capacidad actual del vector */
        int Capacidad() { return capacidad; }

    private:
        void Reubicar(int ncap);

        int *datos;
```

```
        int medida;
        int capacidad;
};

/* Inicia el vector con una capacidad cap */
vector::vector(int cap) : medida(0), capacidad(cap) {
    datos = new int[cap];
}

/* Constructor copia */
vector::vector(vector& v) : medida(v.medida),
capacidad(v.capacidad) {
    datos = new int[v.capacidad];
    for(int i = 0; i < medida; i++)
        datos[i] = v.datos[i];
}

vector::~vector() {
    delete[] datos;
}

void vector::Reubicar(int ncap) {
    int *datos2;
    int i;

    datos2 = new int[ncap];
    for(i = 0; i < min(capacidad, ncap); i++) datos2[i] =
datos[i];
    if(datos) delete[] datos;
    datos = datos2;
}

/* Aumenta la capacidad del vector a la nueva capacidad ncap
*/
void vector::Reservar(int ncap) {
    if(ncap > capacidad) Reubicar(ncap);
    capacidad = ncap;
}

/* Aumenta la capacidad del vector para que pueda almacenar
al menos nm elementos */
void vector::Redimensionar(int nm) {
    int cap = capacidad;
    while(nm > cap) cap *= 2;
    Reubicar(cap);
    capacidad = cap;
}
```

```
/* Retorna un puntero al elemento n del vector */
int &vector::operator[](int n) {
    return datos[n];
}

/* Añade un elemento de valor d al final del vector */
void vector::Agregar(int d) {
    if(medida == capacidad) Redimensionar(medida+1);

    datos[medida++] = d;
}

/* Elimina el último elemento del vector, y devuelve su
valor */
int vector::Sustraer() {
    if(medida > 0) return datos[--medida];
    else return 0;
}

/* Asigna el valor dato a los primeros cuenta elementos del
vector */
void vector::Asignar(int cuenta, int dato) {
    for(int i = 0; i < cuenta; i++) Agregar(dato);
}

/* Devuelve el valor del primer elemento del vector */
int &vector::Primero() {
    if(medida > 0) return datos[0];
    else return datos[0];
}

/* Devuelve el valor del último elemento del vector */
int &vector::Ultimo() {
    if(medida > 0) return datos[medida-1];
    else return datos[0];
}

/* Inserta el valor val en la posición pos del vector */
void vector::Insertar(int pos, int val) {
    if(medida == capacidad) Redimensionar(medida+1);
    for(int i = medida; i > pos; i--) datos[i] = datos[i-1];
    datos[pos] = val;
    medida++;
}

/* Elimina el valor de la posición pos del vector */
void vector::Eliminar(int pos) {
    if(pos < medida) {
```

```
        for(int i = pos; i < medida-1; i++) datos[i] =
datos[i+1];
        medida--;
    }
}

int main() {
    vector v(10);
    int i;

    v.Asignar(10, 0);

    for(i = 0; i < 10; i++) v[i] = i;
    for(i = 0; i < 10; i++) v.Agregar(100+i);
    vector v2(v);

    v.Agregar(23);

    v.Insertar(15, 1000);
    v.Eliminar(16);
    v.Eliminar(108);

    for(i = 0; i < v.Medida(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
    v[3] = 30;
    cout << "v[3] = " << v[3] << endl;

    cout << "Primero = " << v.Primero() << endl;
    cout << "Ultimo = " << v.Ultimo() << endl;

    cout << "v[ultimo] = " << v.Sustraer() << endl;
    cout << "v[ultimo] = " << v.Sustraer() << endl;

    for(i = 0; i < v2.Medida(); i++)
        cout << "v2[" << i << "] = " << v2[i] << endl;

    return 0;

}
```

9.8 Implementación en C++ con plantillas

```
/*
```

```
 * TDA Vector C++ con Plantillas
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */
#include <iostream>
#include "CCadena.h"

using namespace std;

template<class TIPO> class vector {
    public:
        vector(int m);
        vector(vector& v);
        ~vector();
        void Reservar(int ncap);
        void Redimensionar(int nm);
        TIPO &operator[](int n);
        void Agregar(TIPO dato);
        TIPO Sustraer();
        void Asignar(int cuenta, TIPO dato);
        TIPO &Primero();
        TIPO &Ultimo();
        void Insertar(int pos, TIPO val);
        void Eliminar(int pos);
        /* Borra todos los elementos del vector */
        void Borrar() { medida = 0; }
        /* Devuelve el número de elementos del vector */
        int Medida() { return medida; }
        /* Devuelve un valor true si el vector está vacío */
        bool Vacio() { return !medida; }
        /* Devuelve la capacidad actual del vector */
        int Capacidad() { return capacidad; }

    private:
        void Reubicar(int ncap);

        TIPO *datos;
        int medida;
        int capacidad;
};

/* Inicia el vector con una capacidad cap */
template<class TIPO>
vector<TIPO>::vector(int cap) : medida(0), capacidad(cap) {
    datos = new TIPO[cap];
}

/* Constructor copia */
```

```
template<class TIPO>
vector<TIPO>::vector(vector& v) : medida(v.medida),
capacidad(v.capacidad) {
    datos = new TIPO[v.capacidad];
    for(int i = 0; i < medida; i++)
        datos[i] = v.datos[i];
}

template<class TIPO>
vector<TIPO>::~vector() {
    delete[] datos;
}

template<class TIPO>
void vector<TIPO>::Reubicar(int ncap) {
    TIPO *datos2;
    int i;

    datos2 = new TIPO[ncap];
    for(i = 0; i < min(capacidad, ncap); i++) datos2[i] =
datos[i];
    if(datos) delete[] datos;
    datos = datos2;
}

/* Aumenta la capacidad del vector a la nueva capacidad ncap
*/
template<class TIPO>
void vector<TIPO>::Reservar(int ncap) {
    if(ncap > capacidad) Reubicar(ncap);
    capacidad = ncap;
}

/* Aumenta la capacidad del vector para que pueda almacenar
al menos nm elementos */
template<class TIPO>
void vector<TIPO>::Redimensionar(int nm) {
    int cap = capacidad;
    while(nm > cap) cap *= 2;
    Reubicar(cap);
    capacidad = cap;
}

/* Retorna un puntero al elemento n del vector */
template<class TIPO>
TIPO &vector<TIPO>::operator[](int n) {
    return datos[n];
}
```

```
/* Añade un elemento de valor d al final del vector */
template<class TIPO>
void vector<TIPO>::Agregar(TIPO d) {
    if(medida == capacidad) Redimensionar(medida+1);

    datos[medida++] = d;
}

/* Elimina el último elemento del vector, y devuelve su
valor */
template<class TIPO>
TIPO vector<TIPO>::Sustraer() {
    if(medida > 0) return datos[--medida];
    else return 0;
}

/* Asigna el valor dato a los primeros cuenta elementos del
vector */
template<class TIPO>
void vector<TIPO>::Asignar(int cuenta, TIPO dato) {
    for(int i = 0; i < cuenta; i++) Agregar(dato);
}

/* Devuelve el valor del primer elemento del vector */
template<class TIPO>
TIPO &vector<TIPO>::Primero() {
    if(medida > 0) return datos[0];
    else return datos[0];
}

/* Devuelve el valor del último elemento del vector */
template<class TIPO>
TIPO &vector<TIPO>::Ultimo() {
    if(medida > 0) return datos[medida-1];
    else return datos[0];
}

/* Inserta el valor val en la posición pos del vector */
template<class TIPO>
void vector<TIPO>::Insertar(int pos, TIPO val) {
    if(medida == capacidad) Redimensionar(medida+1);
    for(int i = medida; i > pos; i--) datos[i] = datos[i-1];
    datos[pos] = val;
    medida++;
}

/* Elimina el valor de la posición pos del vector */
```

```
template<class TIPO>
void vector<TIPO>::Eliminar(int pos) {
    if(pos < medida) {
        for(int i = pos; i < medida-1; i++) datos[i] =
datos[i+1];
        medida--;
    }
}

int main() {
    vector<int> v(10);
    int i;

    v.Asignar(10, 0);

    for(i = 0; i < 10; i++) v[i] = i;
    for(i = 0; i < 10; i++) v.Agregar(100+i);
    vector<int> v2(v);
    v.Agregar(23);

    v.Insertar(15, 1000);
    v.Eliminar(16);
    v.Eliminar(108);

    for(i = 0; i < v.Medida(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
    v[3] = 30;
    cout << "v[3] = " << v[3] << endl;

    cout << "Primero = " << v.Primero() << endl;
    cout << "Ultimo = " << v.Ultimo() << endl;

    cout << "v[ultimo] = " << v.Sustraer() << endl;
    cout << "v[ultimo] = " << v.Sustraer() << endl;


    for(i = 0; i < v2.Medida(); i++)
        cout << "v2[" << i << "] = " << v2[i] << endl;

    vector<Cadena> vc(10);
    vc.Agregar("primero");
    vc.Agregar("segundo");
    vc.Agregar("tercero");
    vc.Agregar("cuarto");
    vc.Agregar("quinto");
    vc.Agregar("sexto");

    for(i = 0; i < vc.Medida(); i++)
```

```
        cout << "vc[" << i << "] = " << vc[i] << endl;

    return 0;
}
```

Capítulo 10 Montículos binarios

Siguiendo con las estructuras en forma de árbol, nos encontramos con la siguiente: los montículos.

Existen varios tipos de montículos (heap, en inglés). Empezaremos por el más sencillo, los montículos binarios.

10.1 Qué son

Básicamente, se trata de árboles binarios balanceados, como los árboles AVL, la diferencia de altura de cada rama no es mayor de uno. Además, son árboles llenos, salvo el último nivel, que puede estar incompleto. Y cada nivel se completa de izquierda a derecha.

Existe otra propiedad, esta en lo referente al ordenamiento. Existen dos tipos de montículos binarios: de máximos y de mínimos. En los primeros se cumple que el valor de cada nodo es mayor o igual que los valores de sus nodos hijos. En los segundos, el valor de cada nodo es menor o igual que los valores de sus nodos hijos.

Dado que para cada nodo se cumple esta segundo propiedad, cada nodo se comporta como un montículo.

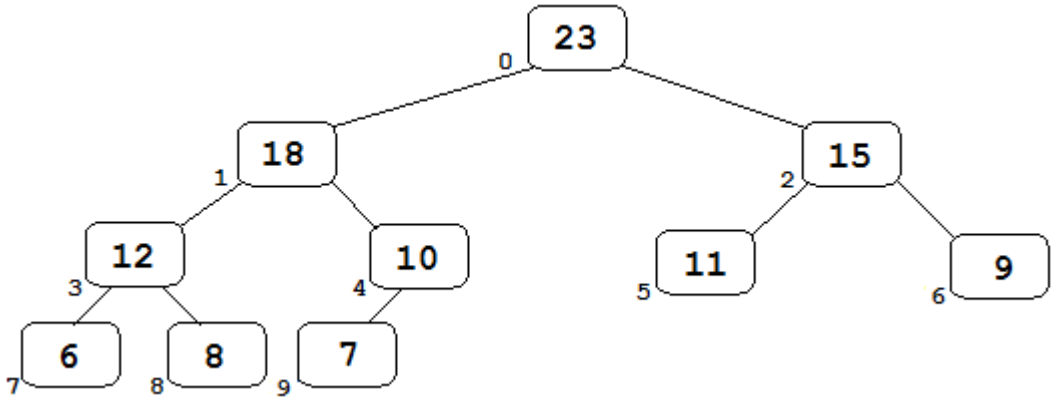
Resumiendo, un montículo binario debe tener las siguientes propiedades:

- Todos los niveles están completos, excepto el último, que puede no estarlo.
- Se llena por niveles, y cada nivel se llena de izquierda a derecha.
- Están ordenados, pero de arriba a abajo.

Dado el orden en que se llena el árbol, cada nodo puede numerarse. Si empezamos en 0, al primer nodo, correspondiente a la raíz, le corresponde el número 0. A sus dos hijos les corresponden los números 1 y 2, izquierda y derecha,

respectivamente. A los nodos hijos del 2, le corresponden los números 3 y 4, etc.

Debido a esto, es posible crear un montículo usando directamente un *array*, ya que sus propiedades hacen que la posición de cada hijo se pueda calcular fácilmente en una estructura lineal. Esto tiene la ventaja de que no es necesario almacenar los punteros a los nodos hijos.



Montículo de máximos

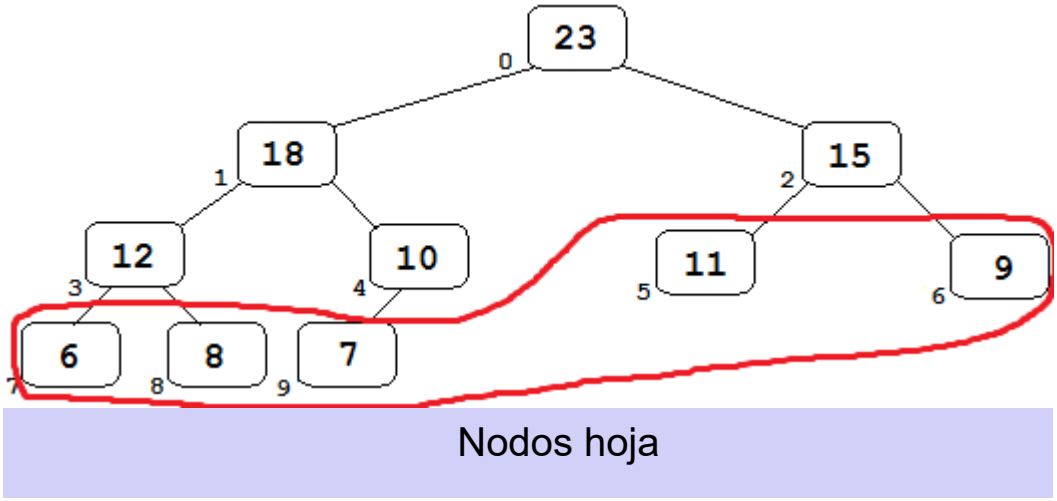
Para calcular los índices de los nodos hijos de cualquier nodo basta con multiplicar el índice del nodo padre por dos, para el hijo izquierdo, y multiplicar por dos y sumar uno para el hijo derecho: `hijoIzq(i) = 2*i+1, hijoDer(i) = 2*i+2`.

Para calcular el índice del nodo padre de cualquier nodo basta con tomar la parte entera de la división entre dos del índice del nodo menos uno, o sencillamente, hacer la división entera del índice menos uno entre dos: `padreNodo(i) = (i-1)/2`.

Otra propiedad interesante es que la segunda mitad de los elementos del montículo son nodos hoja, es decir, nodos sin hijos. Dado cualquier montículo con *n* elementos los *n/2* primeros son nodos con hijos, es decir, pueden considerarse montículos, y el resto son nodos hoja. Veremos la importancia de esto cuando tengamos que convertiro un array cualquiera en un montículo.

10.2 Insertar un nodo

Dado que los montículos se llenan por niveles, insertar un valor nuevo implica hacerlo en la última posición. En general, después de insertar un valor, el montículo habrá perdido la propiedad del orden, de modo que ésta deberá ser restaurada.



Para hacer esto compararemos la clave insertada con la del nodo padre, si no se cumple la propiedad de orden, se intercambian los valores. En un montúculo de máximos, si la clave del nodo hijo es mayor que la del nodo padre, se intercambian. El proceso se repite con el nodo padre, hasta llegar al nodo raíz o hasta que no se produzca un intercambio.

A este proceso se le denomina "subir", ya que las claves suben por el árbol hasta encontrar su posición final.

10.3 Eliminar un nodo

De forma análoga, al eliminar un nodo cualquiera del árbol, para mantener el orden de llenado, lo que haremos será sustituir el valor almacenado en el nodo a borrar por el del último nodo del árbol. Esto elimina el último nodo, pero probablemente también hará que el montículo pierda su propiedad de orden.

Para restaurar el orden compararemos la clave actual del nodo sustituido con las de sus dos hijos y si es menor que alguna de ellas, la intercambiaremos con la mayor. El proceso se repite con el nodo hijo cuya clave hemos intercambiado, hasta que lleguemos a

un nodo hoja, y se interrumpirá cuando no sea necesario hacer un intercambio.

En este algoritmo hay más condiciones y casos posibles. Dado que un nodo puede tener dos, uno o ningún nodo hijo.

A este proceso se le demonima "bajar", ya que las claves bajan por el árbol hasta encontrar su posición final.

10.4 Propiedades y funciones a implementar en un montículo binario

Si se usase una estructura dinámica para implementar el montículo, tendríamos que almacenar como propiedades un puntero a la raíz del árbol, y otro al último nodo insertado. La estructura de esos nodos tendrá cuatro punteros, uno lo usaremos para apuntar al siguiente nodo según el orden de llenado, y el segundo apuntará al nodo padre, esto es necesario porque no podremos usar una fórmula sencilla para calcular la posición del nodo padre. También sería necesario almacenar otros dos punteros a los nodos hijos, ya que veremos que será necesario acceder a ellos cuando se eliminen nodos. Esto hace que usar una estructura en árbol sea poco eficiente para implementar montículos.

Es más habitual usar un *array* para almacenar los valores del montículo. En ese caso almacenaremos el propio array y el índice del último elemento ocupado.

En cuanto a los métodos, implementaremos los siguientes:

Constructor: podemos construir el mónticulo vacío, con una capacidad máxima, o a partir de un array cuyos elementos tengan un orden arbitrario. En el caso de implementarlo en C++, se puede añadir un constructor copia.

Destructor.

Convertir: a partir de un array con los elementos desordenados, convertirlo en un montículo.

Insertar: insertar un nuevo elemento, de modo que el montículo resultante siga teniendo las propiedades de montículo.

Borrar: elimina el valor en la posición indicada.

ExtraerMaximo: devuelve el valor máximo almacenado en el montículo, y lo elimina.

Añadiremos algunas funciones auxiliares:

Subir: sube el valor de un nodo hasta que llegue a su posición final.

Bajar: baja el valor de un nodo hasta que llegue a su posición final.

Padre: calcula el valor del índice del nodo padre de un nodo dado su índice.

Izquierdo: calcula el valor del índice del nodo hijo izquierdo de un nodo dado su índice.

Derecho: calcula el valor del índice del nodo hijo derecho de un nodo dado su índice.

10.5 Algoritmos

Veamos ahora con más detalle algunos de los algoritmos usados para implementar las funciones anteriores. Para estos algoritmos asumiremos que los índices empiezan en 0, es decir, que la raíz estará almacenada en la posición 0 del array.

También asumiremos que se trata de un montículo de máximos. Los algoritmos se pueden modificar fácilmente para adapatarlos a montículos de mínimos.

10.5.1 Subir

Veremos el algoritmo definido de forma recursiva, ya que resulta más sencillo de comprender. En los programas de ejemplo implementaremos una forma iterativa, ya que resulta más eficiente.

- Subir nodo i:
- Si (i no es la raíz) y (clave[i] > clave[padre de i])
    - Intercambiar clave[i] con clave[padre de i]

    - Subir(padre de i)

10.5.2 Bajar

Al igual que con el algoritmo de "Subir", veremos la definición recursiva. En los programas de ejemplo implementaremos una forma iterativa.

- Bajar nodo i:
- Si (i < nElementos/2) *No se trata de un nodo hoja, podemos bajar más.*
    - maximo = i: *No sabemos que nodo contiene el máximo, empezamos asumiendo que lo tiene el nodo actual.*
    - Si (existe hijo_derecho(i)) y (clave[hijo_derecho(i)] > clave[i])
        - maximo = hijo_derecho(i)
    - Si (clave[hijo_izquierdo(i)] > clave[maximo])
        - maximo = hijo_izquierdo[i]
    - Si (maximo != i)
        - Intercambiar clave[i] con clave[maximo]
        - Bajar(maximo)

10.5.3 Insertar

Insertar un nuevo valor es sencillo, disponiendo de la función *Subir*. Asumiremos que los nodos se numeran a partir de 0. De modo que el número de elementos, *nElementos* es siempre una unidad mayor que el índice de la última posición ocupada. Es decir, en un montículo con *n* valores almacenados, el último valor estará almacenado en la posición *n-1*.

- Insertar(v)
- Insertar v[nElementos]: *(En última posición.)*
- Incrementar nElementos
- Subir(nElementos-1)

10.5.4 Borrar

Borrar el máximo equivale a eliminar el primer elemento del montículo, o más concretamente, a sustituirlo por el último, decremetar el número de elementos, y bajar la clave en la raíz a su posición final.

De forma más general, borrar un elemento cualquiera consiste en sustituir su valor por el del último elemento, decrementar el número de elementos, y bajar la clave en la posición borrada hasta su posición final.

- Borrar(i)
- Decrementar nElementos.
- clave[i] = clave[nElementos]
- Bajar(i)

10.5.5 Extraer máximo

Extraer el máximo equivale a eliminar el primer elemento del montículo, o más concretamente, a sustituirlo por el último, decremetar el número de elementos, y bajar la clave en la raíz a su posición final. Deberemos almacenar el valor máximo antes de borrarlo, para poder retornalo.

- ExtraerMaximo
- maximo = clave[0]
- Borrar(0)
- Retornar maximo

10.5.6 Convertir

Dado un array y su número de elementos, este algoritmo convierte el array en un montículo. El método es sencillo, basta con bajar cada uno de los nodos que no son hojas a su posición final, empezando por el último.

- Convertir
- i = nElementos/2: *(Empezamos por el último nodo no hoja)*
- Mientras (i >= 0): *(Hasta llegar a la raíz)*
    - Bajar(i)
    - i = i-1

10.6 Usos

Los montículos están diseñados para implementar colas con prioridad, sobre todo cuando el tiempo de ejecución es crítico, ya que los tiempos de inserción y borrado son constantes e dependen poco del número de elementos en el montículo.

También se pueden usar para ordenar *arrays*, de hecho, son la base del método de ordenamiento heapsort.

El algoritmo usado para crear un montículo desde un array cualquiera ilustra el funcionamiento de este algoritmo.

10.7 Codificación en C

Ejemplo de implementación de un montículo de máximos para enteros en C.

```
/*
 * TDA Heap C
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */

#include <stdio.h>
#include <stdlib.h>

int random(int rango) {
    return (rand()*rango)/RAND_MAX;
}

typedef struct {
    int *heap;
    int capacidad;
```

```
    int iUltimo;
} Heap;

void Construir(Heap *h, int tam);
void MonticuloDesdeArray(Heap *h, int* vec, int tam);
void Destruir(Heap *h);

void Convertir(Heap h);
void Insertar(Heap *h, int v);
void Borrar(Heap *h, int i);
int ExtraerMaximo(Heap *h);
int Vacio(Heap h);
void Mostrar(Heap h);

/* Funciones auxiliares */
void Reubicar(Heap *h, int ncap);
void Intercambia(Heap h, int i1, int i2);
void Subir(Heap h, int i);
void Bajar(Heap h, int i);
int Padre(Heap h, int i);
int Izquierdo(Heap h, int i);
int Derecho(Heap h, int i);

void Construir(Heap *h, int tam){
    h->capacidad = tam;
    h->iUltimo = 0;
    h->heap = (int*)malloc(sizeof(int) * tam);
}

void MonticuloDesdeArray(Heap *h, int *vec, int tam) {
    int i;

    h->capacidad = tam;
    h->iUltimo = tam;
    h->heap = (int*)malloc(sizeof(int) * h->capacidad);
    for(i = 0; i < h->iUltimo; i++) h->heap[i] = vec[i];
    Convertir(*h);
}

void Destruir(Heap *h) {
    free(h->heap);
}

void Convertir(Heap h) {
    int i;

    for(i = h.iUltimo/2; i >= 0 ; i--) {
        Bajar(h, i);
```

```
    }
}

void Insertar(Heap *h, int v) {
    if(h->iUltimo >= h->capacidad) Reubicar(h, h-
>capacidad*2);

    h->heap[h->iUltimo++] = v;
    Subir(*h, h->iUltimo-1);
}

void Borrar(Heap *h, int i) {
    if(i < h->iUltimo) {
        h->heap[i] = h->heap[--h->iUltimo];
        Bajar(*h, i);
    }
}

int ExtraerMaximo(Heap *h) {
    int retval = h->heap[0];

    Borrar(h, 0);

    return retval;
}

void Reubicar(Heap *h, int ncap) {
    int *heap2;
    int i;

    heap2 = (int*)malloc(sizeof(int) * ncap);
    for(i = 0; i < h->iUltimo; i++) heap2[i] = h->heap[i];
    if(h->heap) free(h->heap);
    h->heap = heap2;
    h->capacidad = ncap;
}

void Intercambia(Heap h, int i1, int i2) {
    int aux;

    aux = h.heap[i1];
    h.heap[i1] = h.heap[i2];
    h.heap[i2] = aux;
}

// Algoritmo no recursivo para subir
void Subir(Heap h, int i) {
    int iPadre;
```

```
    while(i > 0 && h.heap[i] > h.heap[iPadre=Padre(h, i)]) {
        Intercambia(h, i, iPadre);
        i = iPadre;
    }
}

void Bajar(Heap h, int i){
    int iIzq, iDer, maximo;

    maximo = i;
    do{
        i = maximo;
        iIzq=Izquierdo(h, i);
        iDer=Derecho(h, i);
        if(iDer < h.iUltimo && h.heap[iDer] >
h.heap[maximo]) maximo = iDer;
        if(iIzq < h.iUltimo && h.heap[iIzq] >
h.heap[maximo]) maximo = iIzq;
        if(i != maximo) Intercambia(h, i, maximo);
    } while (i != maximo && maximo < h.iUltimo/2);
}

int Padre(Heap h, int i) {
    return (i-1)/2;
}

int Izquierdo(Heap h, int i) {
    return 2*i+1;
}

int Derecho(Heap h, int i) {
    return 2*i+2;
}

int Vacio(Heap h) {
    return h.iUltimo == 0;
}

void Mostrar(Heap h) {
    int i;

    if(Vacio(h)) printf("Heap vacio\n");
    else {
        for(i = 0; i < h.iUltimo; i++) printf("%d,",
h.heap[i]);
        printf("\n");
    }
```

```
}

int main()
{
    Heap h1;
    int i;

    Construir(&h1, 10);
    for(i = 0; i < 20; i++) Insertar(&h1, random(200));

    Mostrar(h1);
    while(!Vacio(h1)) {
        printf("%d:", ExtraerMaximo(&h1));
    }
    printf("\n");
    Destruir(&h1);

    int v[] = {43, 23, 12, 4, 45, 32, 18, 6, 17};
    Heap h2;
    MonticuloDesdeArray(&h2, v, sizeof(v)/sizeof(int));
    Mostrar(h2);
    while(!Vacio(h2)) {
        printf("%d:", ExtraerMaximo(&h2));
    }
    Destruir(&h2);

    return 0;
}
```

10.8 Codificación en C++

Ejemplo de implementación de un montículo de máximos para enteros en C++.

```
/*
 * TDA Heap C++
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */

#include <iostream>
#include <cstdlib>
```

```
using namespace std;

int random(int rango) {
    return (rand()*rango)/RAND_MAX;
}

class Heap {
    public:
        Heap(int tam=10);
        Heap(int *vec, int tam);
        Heap(Heap &);
        ~Heap();
        void Convertir();
        void Insertar(int v);
        void Borrar(int i);
        int ExtraerMaximo();
        bool Vacio() { return iUltimo == 0; }
        void Mostrar();
    private:
        void Reubicar(int ncap);
        void Intercambia(int i1, int i2);
        void Subir(int i);
        void Bajar(int i);
        int Padre(int i) { return (i-1)/2; }
        int Izquierdo(int i) { return 2*i+1; }
        int Derecho(int i) { return 2*i+2; }
        int *heap;
        int capacidad;
        int iUltimo;
};

Heap::Heap(int tam) : capacidad(tam), iUltimo(0) {
    heap = new int[tam];
}

Heap::Heap(int *vec, int tam) : capacidad(tam), iUltimo(tam)
{
    heap = new int[capacidad];
    for(int i = 0; i < iUltimo; i++) heap[i] = vec[i];
    Convertir();
}

Heap::Heap(Heap &h) : capacidad(h.capacidad),
iUltimo(h.iUltimo) {
    heap = new int[capacidad];
    for(int i = 0; i < iUltimo; i++) heap[i] = h.heap[i];
}
```

```
Heap::~Heap() {
    delete[] heap;
}

// Para cada elemento, verificar si el valor de sus hijos es
menor
// Si alguno es mayor, intercambiar valor del elemento con
el mayor de los hijos.
//           0
//      1          2
//   3    4     5     6
// 7  8  9 10 11 12 13 14
void Heap::Convertir() {
    for(int i = iUltimo/2; i >= 0 ; i--) {
        Bajar(i);
    }
}

void Heap::Insertar(int v) {
    if(iUltimo >= capacidad) Reubicar(capacidad*2);

    heap[iUltimo++] = v;
    Subir(iUltimo-1);
}

void Heap::Borrar(int i) {
    if(i < iUltimo) {
        heap[i] = heap[--iUltimo];
        Bajar(i);
    }
}

int Heap::ExtraerMaximo() {
    int retval = heap[0];

    Borrar(0);

    return retval;
}

void Heap::Reubicar(int ncap) {
    int *heap2;

    heap2 = new int[ncap];
    for(int i = 0; i < iUltimo; i++) heap2[i] = heap[i];
    if(heap) delete[] heap;
    heap = heap2;
    capacidad = ncap;
```

```
}

void Heap::Intercambia(int i1, int i2) {
    int aux;

    aux = heap[i1];
    heap[i1] = heap[i2];
    heap[i2] = aux;
}

// Algoritmo no recursivo para subir
void Heap::Subir(int i) {
    int iPadre;

    while(i > 0 && heap[i] > heap[iPadre=Padre(i)]) {
        Intercambia(i, iPadre);
        i = iPadre;
    }
}

void Heap::Bajar(int i){
    int iIzq, iDer, maximo;

    maximo = i;
    do{
        i = maximo;
        iIzq=Izquierdo(i);
        iDer=Derecho(i);
        if(iDer < iUltimo && heap[iDer] > heap[maximo])
maximo = iDer;
        if(iIzq < iUltimo && heap[iIzq] > heap[maximo])
maximo = iIzq;
        if(i != maximo) Intercambia(i, maximo);
    } while (i != maximo && maximo < iUltimo/2);
}

void Heap::Mostrar() {
    if(Vacio()) cout << "Heap vacio" << endl;
    else {
        for(int i = 0; i < iUltimo; i++) cout << heap[i] <<
",";
        cout << endl;
    }
}

int main()
{
    Heap h1(10);
```

```
    for(int i = 0; i < 20; i++) h1.Insertar(random(200));

    h1.Mostrar();
    while(!h1.Vacio()) {
        cout << h1.ExtraerMaximo() << ":";
    }
    cout << endl;

    int v[] = {43, 23, 12, 4, 45, 32, 18, 6, 17};
    Heap h2(v, sizeof(v)/sizeof(int));
    h2.Mostrar();
    while(!h2.Vacio()) {
        cout << h2.ExtraerMaximo() << ":";
    }

    return 0;
}
```

10.9 Codificación en C++ con plantillas

Ejemplo de implementación de un montículo de máximos en C++ usando plantillas.

```
/*
 * TDA Vector C++ con Plantillas
 * Salvador Pozo Coronado
 * Febrero 2013 Con Clase
 */
 #include <iostream>

using namespace std;

template<class TIPO> class vector {
    public:
        vector(int m);
        vector(vector& v);
        ~vector();
        void Reservar(int ncap);
        void Redimensionar(int nm);
        TIPO &operator[](int n);
        void Agregar(TIPO dato);
        TIPO Sustraer();
```

```
        void Asignar(int cuenta, TIPO dato);
        TIPO &Primero();
        TIPO &Ultimo();
        void Insertar(int pos, TIPO val);
        void Eliminar(int pos);
        /* Borra todos los elementos del vector */
        void Borrar() { medida = 0; }
        /* Devuelve el número de elementos del vector */
        int Medida() { return medida; }
        /* Devuelve un valor true si el vector está vacío */
        bool Vacio() { return !medida; }
        /* Devuelve la capacidad actual del vector */
        int Capacidad() { return capacidad; }

    private:
        void Reubicar(int ncap);

        TIPO *datos;
        int medida;
        int capacidad;
};

/* Inicia el vector con una capacidad cap */
template<class TIPO>
vector<TIPO>::vector(int cap) : medida(0), capacidad(cap) {
    datos = new TIPO[cap];
}

/* Constructor copia */
template<class TIPO>
vector<TIPO>::vector(vector& v) : medida(v.medida),
capacidad(v.capacidad) {
    datos = new TIPO[v.capacidad];
    for(int i = 0; i < medida; i++)
        datos[i] = v.datos[i];
}

template<class TIPO>
vector<TIPO>::~vector() {
    delete[] datos;
}

template<class TIPO>
void vector<TIPO>::Reubicar(int ncap) {
    int *datos2;
    int i;

    datos2 = new TIPO[ncap];
```

```
    for(i = 0; i < min(capacidad, ncap); i++) datos2[i] =
datos[i];
    if(datos) delete[] datos;
    datos = datos2;
}

/* Aumenta la capacidad del vector a la nueva capacidad ncap
*/
template<class TIPO>
void vector<TIPO>::Reservar(int ncap) {
    if(ncap > capacidad) Reubicar(ncap);
    capacidad = ncap;
}

/* Aumenta la capacidad del vector para que pueda almacenar
al menos nm elementos */
template<class TIPO>
void vector<TIPO>::Redimensionar(int nm) {
    int cap = capacidad;
    while(nm > cap) cap *= 2;
    Reubicar(cap);
    capacidad = cap;
}

/* Retorna un puntero al elemento n del vector */
template<class TIPO>
TIPO &vector<TIPO>::operator[](int n) {
    return datos[n];
}

/* Añade un elemento de valor d al final del vector */
template<class TIPO>
void vector<TIPO>::Agregar(TIPO d) {
    if(medida == capacidad) Redimensionar(medida+1);

    datos[medida++] = d;
}

/* Elimina el último elemento del vector, y devuelve su
valor */
template<class TIPO>
TIPO vector<TIPO>::Sustraer() {
    if(medida > 0) return datos[--medida];
    else return 0;
}

/* Asigna el valor dato a los primeros cuenta elementos del
vector */
```

```
template<class TIPO>
void vector<TIPO>::Asignar(int cuenta, TIPO dato) {
    for(int i = 0; i < cuenta; i++) Agregar(dato);
}

/* Devuelve el valor del primer elemento del vector */
template<class TIPO>
TIPO &vector<TIPO>::Primero() {
    if(medida > 0) return datos[0];
    else return datos[0];
}

/* Devuelve el valor del último elemento del vector */
template<class TIPO>
TIPO &vector<TIPO>::Ultimo() {
    if(medida > 0) return datos[medida-1];
    else return datos[0];
}

/* Inserta el valor val en la posición pos del vector */
template<class TIPO>
void vector<TIPO>::Insertar(int pos, TIPO val) {
    if(medida == capacidad) Redimensionar(medida+1);
    for(int i = medida; i > pos; i--) datos[i] = datos[i-1];
    datos[pos] = val;
    medida++;
}

/* Elimina el valor de la posición pos del vector */
template<class TIPO>
void vector<TIPO>::Eliminar(int pos) {
    if(pos < medida) {
        for(int i = pos; i < medida-1; i++) datos[i] =
datos[i+1];
        medida--;
    }
}

int main() {
    vector<int> v(10);
    int i;

    v.Asignar(10, 0);

    for(i = 0; i < 10; i++) v[i] = i;
    for(i = 0; i < 10; i++) v.Agregar(100+i);
    vector<int> v2(v);
    v.Agregar(23);
```

```
    v.Insertar(15, 1000);
    v.Eliminar(16);
    v.Eliminar(108);

    for(i = 0; i < v.Medida(); i++)
        cout << "v[" << i << "] = " << v[i] << endl;
    v[3] = 30;
    cout << "v[3] = " << v[3] << endl;

    cout << "Primero = " << v.Primero() << endl;
    cout << "Ultimo = " << v.Ultimo() << endl;

    cout << "v[ultimo] = " << v.Sustraer() << endl;
    cout << "v[ultimo] = " << v.Sustraer() << endl;


    for(i = 0; i < v2.Medida(); i++)
        cout << "v2[" << i << "] = " << v2[i] << endl;
    return 0;
}
```

Tabla de contenido