NOTAS DEL CURSO:

ESTRUCTURAS DE DATOS I

PROFESOR:

LUIS ALBERTO MUÑOZ GÓMEZ

Módulo I. Introducción

Operaciones básicas sobre un arreglo de registros (el ABC):

- Alta (inserción)
- Baja (eliminación)
- Consulta
- Cambio (modificación)

Apuntador.- indicador que señala otra celda de la memoria.

Variables automáticas.- son las variables locales, parámetros y cualquier otra variable declarada dentro del ámbito de un bloque de código (delimitado por { } en C/C++); para estas variables se requiere memoria en tiempo de ejecución al momento de declararlas y su espacio de memoria es vigente para usarlo solo mientras se permanece dentro del bloque. Nótese que sería inapropiado que en una función se intentara devolver la dirección de memoria de una variable local pues esta memoria desaparece al terminar la función.

Recursividad

Es la forma en la cual se especifica un proceso basado en su propia definición.

Factorial:

$$f(n)= \begin{cases} n=0: 1 \\ n>0: n*f(n-1) \\ \text{otro: indefinido} \end{cases}$$

Sucesión de Fibonacci:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ para } n = 2, 3, 4, 5, \ldots$$

1.4 Corrimiento de Bits

```
00000000 00000000 00000000 00000001       1d
<<1
00000000 00000000 00000000 00000010
<<1
00000000 00000000 00000000 00000100
...
00000000 00000000 00000000 10000000
<<1
00000000 00000000 00000001 00000000       256d
>>1
00000000 00000000 00000000 10000000


11111111 11111111 11111111 11111111+
00000000 00000000 00000000 00000001
-----------------------------------
00000000 00000000 00000000 00000000

11111111 11111111 11111111 00000000       -256d
00000000 00000000 00000001 00000000+
-----------------------------------
00000000 00000000 00000000 00000000


11111111 11111111 11111111 00000000
>>1
11111111 11111111 11111111 10000000
>>1
11111111 11111111 11111111 11000000

...
11111111 11111111 11111111 11111111


10000000 00000000 00000000 00000000+
00000000 00000000 00000000 00000001
-----------------------------------
10000000 00000000 00000000 00000001


10000000 00000000 00000000 00000001
<<1
00000000 00000000 00000000 00000010
```

Prueba de bit encendido

```
00000000 00000000 00000000 00000001      test1
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00000001      resultado

00000000 00000000 00000000 00000010      test2
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00000010      resultado

00000000 00000000 00000000 00000100      test3
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00000100      resultado

00000000 00000000 00000000 00001000      test4
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00001000      resultado

00000000 00000000 00000000 10000000      test8
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00000000      resultado

00000000 00000000 00000001 00000000      test9
00000000 00000000 00000000 01111111&     valor1=127d
-----------------------------------------------------
00000000 00000000 00000000 00000000      resultado

00000000 00000000 00000000 10000000      test8
00000000 00000000 00000001 10000000&     valor1=256+128=384
-----------------------------------------------------
00000000 00000000 00000000 10000000      resultado

00000000 00000000 00000001 00000000      test9
00000000 00000000 00000001 10000000&     valor1=384
-----------------------------------------------------
00000000 00000000 00000001 00000000      resultado
```

Módulo II. Clasificación y Búsqueda

Búsqueda Lineal

También llamada búsqueda secuencial. Se utiliza cuando el vector no está ordenado previamente. Consiste en buscar el elemento buscándolo secuencialmente con cada elemento del arreglo hasta encontrarlo, o hasta que se llegue al final. La existencia se puede asegurar cuando el elemento es localizado, sin embargo la no existencia solo se puede asegurar hasta haber examinado todos los elementos del array.

Ordenamiento por Burbuja

Consiste en que en el primer recorrido del arreglo, el registro más “ligero” (el de la clave menor) sube hasta la superficie. En el segundo recorrido la segunda clave menor sube hasta la segunda posición y así sucesivamente.

Ordenamiento por Inserción

Se denomina así porque en el i-ésimo recorrido se “inserta” el i-ésimo elemento A[i] en el lugar correcto, entre los elementos del arreglo que fueron ordenados previamente.

Ordenamiento por Selección

En el i-ésimo recorrido se selecciona el j-ésimo registro con la clave más pequeña de entre los que síguen al i-ésimo elemento, y se intercambia el seleccionado con el i-ésimo. Como resultado todos los elementos hasta el i-ésimo ya estarán ordenados.

Shell Sort

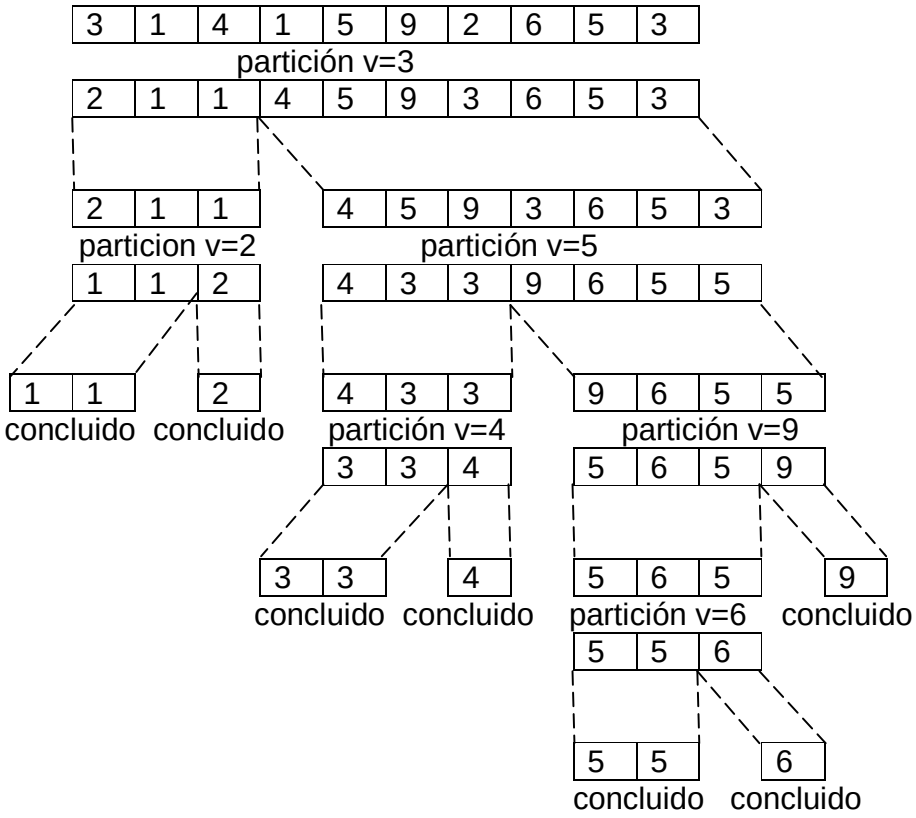
Shell sort es una generalización del algoritmo por inserción teniendo en cuenta dos observaciones acerca del ordenamiento por inserción:

1. es eficiente si la entrada está “casi ordenada”
2. es ineficiente, en general, porque mueve los valores solo una vez

Shell sort mejora el algoritmo por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga “pasos más grandes” hacia su posición esperada. Cada vez los pasos son más pequeños. El último paso de Shell sort es un ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del arreglo están casi ordenados.

Quick Sort

Consiste en clasificar el arreglo tomando un valor clave *v* como elemento pivote alrededor del cual reorganizar los elementos del arreglo. Se permutan los elementos del arreglo con el fin de que para alguna *j* todos los registros con claves menores a *v* aparezcan desde el primer elemento del arreglo hasta el A[*j*] y todos aquellos con claves *v* o mayores aparezcan desde A[*j*+1] en adelante teniendo así dos particiones. Después se aplica recursivamente la clasificación a cada una de las particiones.



Merge Sort

Es un algoritmo basado en la técnica “divide y vencerás”. El funcionamiento es el siguiente:

1. Si la longitud de la lista es 0 o 1, entonces ya está ordenada. En otro caso:
2. Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño
3. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla
4. Mezclar las dos sublistas en una lista ordenada

Búsqueda Binaria

Probablemente se trate de la aplicación más sencilla de divide y vencerás. A partir de un arreglo ordenado por orden no decreciente, tal que $A[i] \leq A[j]$ siempre que sea $1 \leq i \leq j \leq n$; el problema se trata de encontrar x en A, si es que está.

En comparación con la búsqueda secuencial, para acelerar la búsqueda, deberíamos buscar x o bien en la primera mitad del arreglo, o bien en la segunda. Para averiguar en cual mitad, se examina el elemento A[k] a la mitad del arreglo. Si $x \leq A[k]$ entonces se buscará en la primera mitad; de lo contrario en la segunda.

Módulo III. Implementación del TDA Lista con Arreglos

*T.D.A.(*Tipo de Dato Abstracto*)*.- No tiene representación; Modelo matemático con un conjunto de operaciones definidas sobre ese modelo. Se utilizan para gestionar información. Ejem: el modelo matemático conjunto con sus operaciones unión, intersección y diferencia.

Estructura de Datos.- Representa el modelo básico de un T.D.A. y son un conjunto de variables. Conjunto de datos, quizás de distintos tipos, organizados de una manera específica, conectados entre sí de diversas formas, dependiendo de lo que se desea representar. Su componente básico es la celda. La estructura de datos más simple es el arreglo unidimensional.

Celda.- Caja capaz de almacenar un valor tomado de algún tipo de datos, básico (entero, flotante, etc.) o compuesto (registro).

TDA Lista

Las listas constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera; los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista.

Las listas también pueden concatenarse entre sí o dividirse en sublistas; se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación.

Matemáticamente, una lista es una secuencia de cero o más elementos de un tipo determinado (por lo general denominado TipoElemento)

$$a_1, a_2, \ldots, a_n$$

Operaciones:

0. FIN(L). Devuelve la posición que sigue a la posición n en una lista L de n elementos.
1. INSERTA(*x*, *p*, *L*). Inserta *x* en la posición *p* de la lista *L*, pasando los elementos de la posición *p* y siguientes a la posición inmediata posterior. Si *L* es a_1, a_2, …, a_n se convierte en a_1, a_2, …, a_{p-1}, *x*, a_p, … , a_n Si *p* es FIN(*L*), entonces *L* se convierte en a_1, a_2, …, a_n, *x*. Si la lista *L* no tiene posición *p*, el resultado es indefinido
2. LOCALIZA(*x*, *L*). Devuelve la posición de *x* en la lista *L*. Si *x* figura más de una vez en *L*, la posición de la primera aparición de *x* es la que se devuelve. Si *x* no figura en la lista, entonces se devuelve FIN(*L*)
3. RECUPERA(*p*, *L*). Devuelve el elemento que está en la posición *p* de la lista *L*. El resultado no está definido si *p*=FIN(*L*) o si *L* no tiene posición *p*.
4. SUPRIME(*p*, *L*). Elimina el elemento en la posición *p* de la lista *L*. El resultado no está definido si *L* no tiene posición *p* o si *p*=FIN(*L*).
5. SIGUIENTE(*p*, *L*) y ANTERIOR(*p*, *L*) devuelven las posiciones siguiente y anterior, respectivamente, a *p* en la lista *L*. Si *p* es la última posición de *L*, SIGUIENTE(*p*, *L*)=FIN(*L*). SIGUIENTE no está definida si *p* es FIN(*L*). ANTERIOR no está definida si *p* es primero(L). Ambas funciones están indefinidas cuando *L* no tiene posición *p*.
6. INICIALIZA(*L*). Ocasiona que *L* se convierta en la lista vacía y devuelve la posición FIN(*L*).
7. PRIMERO(*L*). Devuelve la primera posición de la lista *L*. Si *L* está vacía, la posición que se devuelve es FIN(*L*).
8. IMPRIME_LISTA(*L*). Imprime los elementos de L en su orden de aparición en la lista

A partir de las operaciones básicas anteriores se pueden diseñar otras adicionales según las necesidades, como:

- ESTA_VACIA(*L*). Devuelve *verdadero* si la lista no tiene elementos; de lo contrario *falso*
- ESTA_LLENA(*L*). Devuelve *verdadero* si ya no hay más espacio para almacenar elementos; de lo contrario *falso*
- MISMO(x, y). Compara los elementos *x* y *y* devolviendo verdadero si son iguales en su información comparable; de lo contrario, falso
- PURGA(*L*). Elimina los elementos duplicados de la lista L

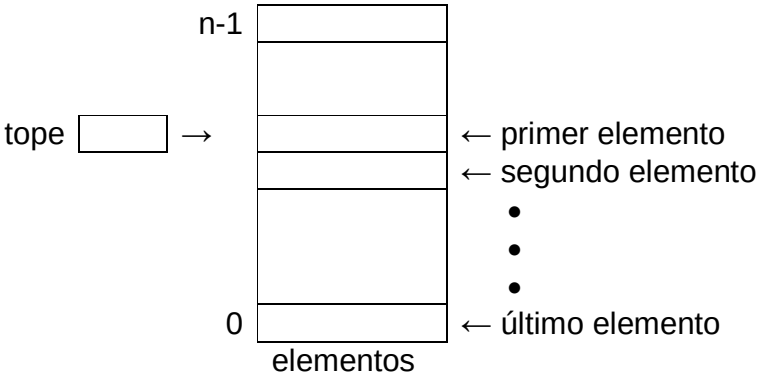
Al principio puede parecer tedioso escribir procedimientos que rijan todos los accesos a las estructuras subyacentes de un programa. Sin embargo, si se logra establecer la disciplina de escribir programas en función de operaciones de manipulación de tipos de datos abstractos en lugar de usar ciertos detalles de implantación particulares, es posible modificar los programas más fácilmente con solo aplicar de nuevo las operaciones, en lugar de buscar en todos los programas aquellos lugares donde se hacen accesos a las estructuras de datos subyacentes. La flexibilidad que se obtiene con esto puede ser especialmente importante en proyectos grandes.

Módulo IV. El TDA Pila y Recursión

Una *pila* es un tipo especial de lista en la que todas las inserciones y supresiones tienen lugar en un extremo denominado *tope*. A las listas se les llama también “listas LIFO” (*last in first out*) o listas “último en entrar, primero en salir”. Un tipo de datos abstracto de la familia PILA incluye a menudo las cinco operaciones siguientes:

1. INICIALIZA(*P*) convierte la pila P en una pila vacía. Esta operación es exactamente la misma que para las listas generales.
2. TOPE(*P*) devuelve el valor del elemento de la parte superior de la pila *P*. TOPE(*P*) puede escribirse en función de operaciones con listas como RECUPERA(PRIMERO(P),P)
3. DESAPILA(*P*) , en inglés *POP*, suprime el elemento superior de la pila, es decir, equivale a SUPRIME(PRIMERO(*P*),*P*). Algunas veces resulta conveniente implantar DESAPILA como una función que devuelve el elemento que acaba de suprimir.
4. APILA(*x*, *P*), en inglés *PUSH*, inserta el elemento *x* en la parte superior de la pila *P*. El anterior tope se convierte en el siguiente elemento, y así sucesivamente. En función de operaciones primitivas con listas, esta operación es INSERTA(*x*, PRIMERO(*P*),*P*).
5. ESTA_VACIA(*P*) devuelve verdadero si la pila *P* está vacía, y falso en caso contrario.

- Se puede optimizar la implantación de la pila si se utiliza el campo “tope” de la estructura lista para representar el tope de la pila, como sigue:



TDA Pila como clase

```
Pila pila(9);
int x;
pila.apila(12);//supongo que tiene espacio, pero cuidado
pila.apila(17);
pila.apila(18);
pila.apila(21);
pila.apila(25);

x=20;
while(!pila.estaLlena()){
  pila.apilar(x);
  x+=2;
}

while(!pila.estaVacia()){
  x=pila.desapila();
  cout << x << endl;
}
```

- Torres de Hanoi
- Conversión de notación infija a posfija

Módulo V. El TDA Cola

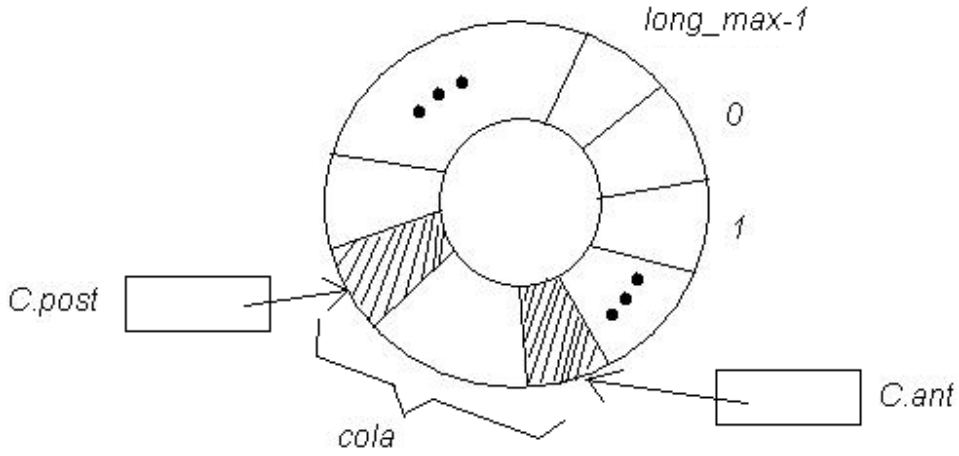
Una *cola* es otro tipo especial de lista en el cual los elementos se insertan en un extremo (el *posterior*) y se suprimen en el otro (el *anterior* o *frente*). Las colas se conocen también como listas “FIFO” (*first-in first-out*) o listas “primero en entrar, primero en salir”. Las operaciones para una cola son análogas a las de las pilas; las diferencias sustanciales consisten en que las inserciones se hacen al final de la lista, y no al principio, y en que la terminología tradicional para colas y listas no es la misma. Las operaciones sobre colas son:

1. INICIALIZA(*C*) convierte la cola C en una cola vacía.
2. FRENTE(*C*) es una función que devuelve el valor del primer elemento de la cola *C*. FRENTE(*C*) se puede escribir en función de operaciones con listas, como RECUPERA(PRIMERO(*C*), *C*).
3. ENCOLA(*x*, *C*) inserta el elemento *x* al final de la cola *C*. En función de operaciones con listas, ENCOLA(x, *C*) es INSERTA(*x*, FIN(*C*), *C*).
4. DESENCOLA(*C*) suprime el primer elemento de *C*; es decir, DESENCOLA(*C*) es SUPRIME(PRIMERO(*C*), *C*).
5. ESTA_VACIA(*C*) devuelve verdadero si, y sólo si, *C* es una cola vacía.

Colas Circulares:

La representación de listas para una cola como se planteó anteriormente no es muy eficiente. Es cierto que usando el campo “tope” de la lista, es posible ejecutar ENCOLA en un número fijo de pasos, pero DESENCOLA, que suprime el primer elemento, requiere que la cola completa ascienda una posición en el arreglo. Así pues, DESENCOLA lleva un tiempo acorde la longitud de la cola.

Para evitar este gasto, se debe adoptar un punto de vista diferente. Imaginarse un arreglo como un círculo en el que la primera posición sigue a la última.



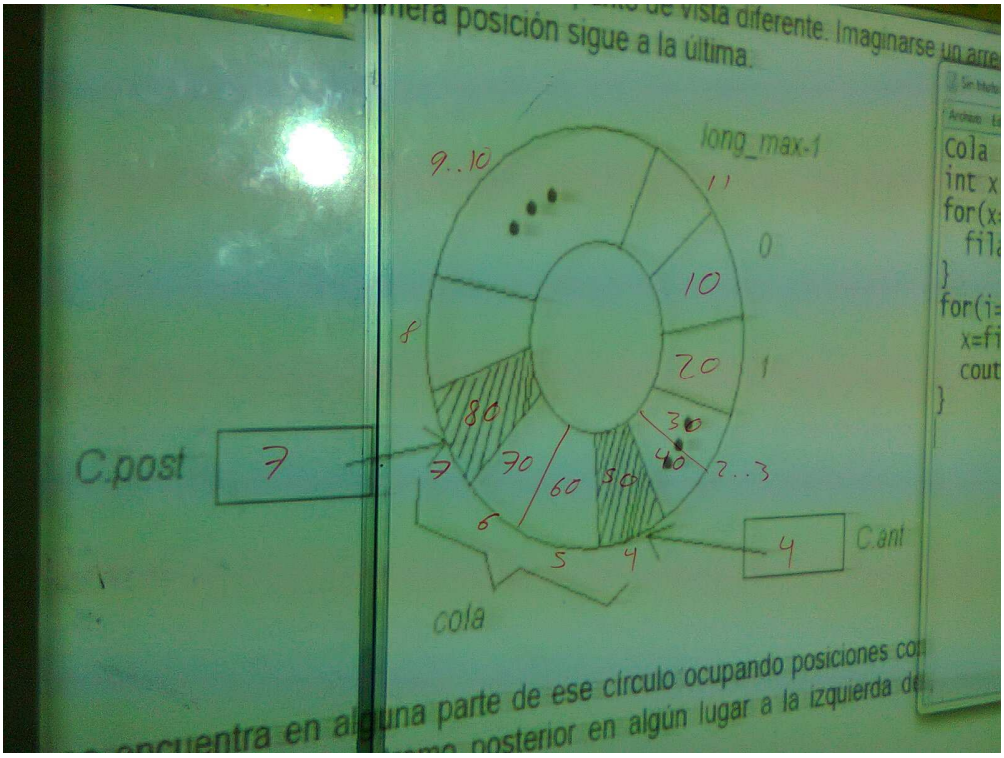
La cola se encuentra en alguna parte de ese círculo ocupando posiciones consecutivas (en sentido circular), con el extremo posterior en algún lugar a la izquierda del extremo anterior.

Para insertar un elemento en la cola, se mueve el apuntador *C.post* a una posición en el sentido de las manecillas del reloj, y se escribe el elemento en esa posición. Para suprimir, simplemente se mueve *C.ant* una posición en el sentido de las manecillas del reloj. De esa manera, la cola se mueve en ese mismo sentido conforme se insertan y suprimen elementos. Así los procedimientos ENCOLA y DESENCOLA se pueden escribir de tal manera que su ejecución se realice en un número constante de pasos.

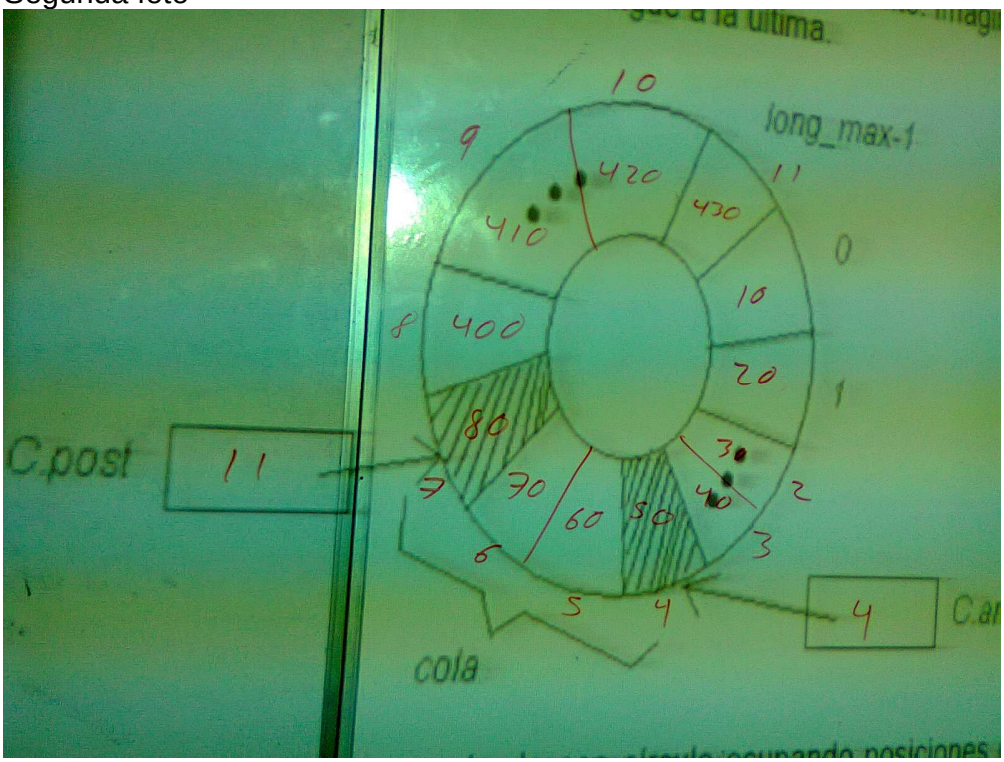
El TDA Cola como clase

```
Cola fila(12);//debe solicitar 12 celdas, pero solo caben 11
int x,i,j;
for(x=10,i=0,j=8;i<j;i++,x+=10){
  fila.encolar(x);//encola de 10a80
}
for(i=0,j=4;i<j;i++){
  x=fila.desencola();
  cout << x << endl;//imprime 10a40
}
//primera foto
for(x*=10,i=0,j=4;i<j;i++,x+=10){
  fila.encolar(x);//encola 400a430
}
//segunda foto
for(x*=10,i=0,j=4;i<j;i++,x+=10){
  fila.encolar(x);//encola 4400a4420
  /*imprime error de programador
    en la cuarta vuelta*/
}
//tercera foto
```

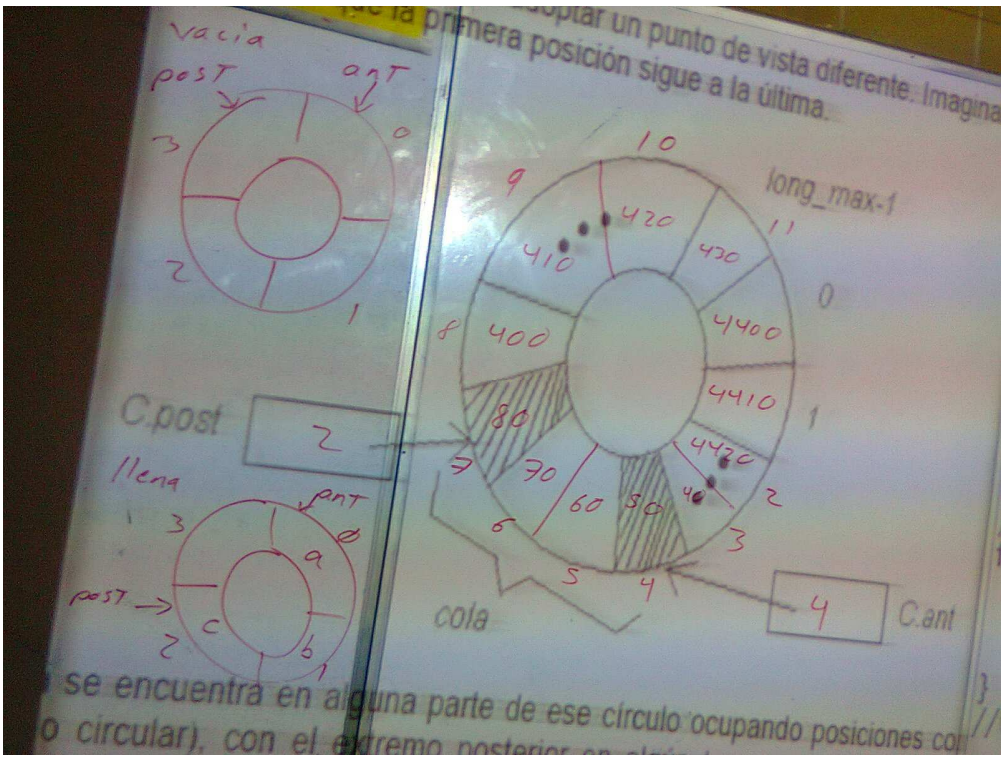
Primera foto



Segunda foto



Tercera foto



Módulo VI. TDA Lista con Cursores

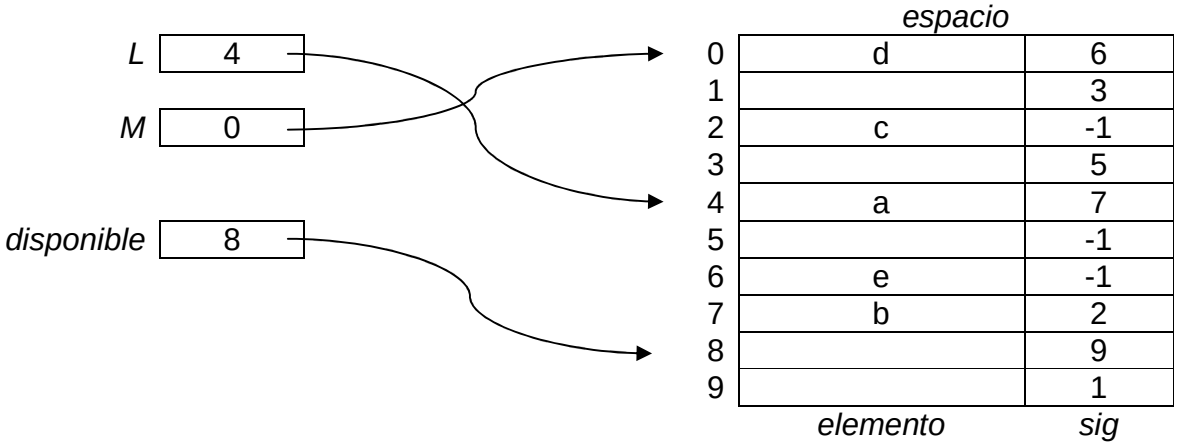
Algunos lenguajes, como FORTRAN y ALGOL, no tienen apuntadores. Si se trabaja con un lenguaje tal, se pueden simular los apuntadores mediante cursores; esto es, con enteros que indican posiciones en arreglos.

Se crea un arreglo de registros para almacenar todas las listas de elementos cuyo tipo es TipoElemento; cada registro contiene un elemento y un entero que se usa como cursor, o sea, el índice del siguiente elemento en la lista.

Un valor de -1 ya sea en una lista *L* o en el campo *sig* representa “apuntador a null”, esto es, la lista está vacía o no hay un elemento siguiente, respectivamente.

Se adoptará la convención de que la posición *i* de la lista *L* es el índice de la celda de espacio que contiene el elemento *i*-1 de *L*, ya que el campo *sig* de esa celda contendrá el cursor al elemento *i*. Como caso especial, la posición 0 de cualquier lista se representa por -1. Dado que el nombre de la lista es siempre un parámetro de las operaciones que usan posiciones, es posible distinguir entre las primeras posiciones de distintas listas. La posición fin(*L*) es el índice del último elemento de *L*.

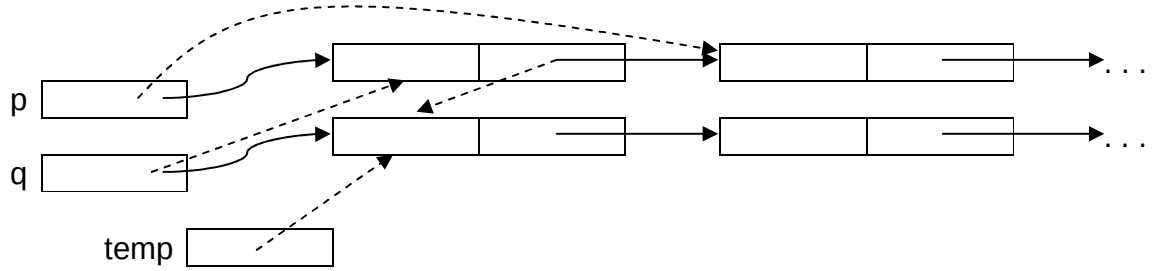
A continuación se muestran dos listas, *L = a, b, c* y *M = d, e*, que comparten el arreglo *espacio* de tamaño 10:



Obsérvese que las celdas del arreglo no contenidas en *L* ni *M* están enlazadas a otra lista llamada *disponible*. Tal lista es necesaria para obtener una celda vacía cuando se desee hacer una inserción, y disponer de lugar donde poner las celdas suprimidas para su uso posterior.

Para insertar un elemento *x* en la lista *L*, se toma la primera celda de la lista disponible y se coloca en la posición correcta en la lista *L*. El elemento x se pone entonces en el campo *elemento* de esa celda. Para suprimir un elemento *x* de la lista *L*, se quita de *L* la celda que contiene *x* y se devuelve al principio de la lista *disponible*. Estas dos acciones pueden verse como casos especiales del acto de tomar una celda *C* apuntada por un cursor *p* y hacer que otro cursor *q* apunte a *C*, a la vez que se hace que *p* quede apuntando hacia donde *C* apuntaba y *C* quede apuntando hacia donde *q* apuntaba.. De esta forma, *C* se inserta entre *q* y aquello hacia lo que apuntaba *q*.

Por ejemplo, si se suprime *b* de la lista *L* en la figura anterior, *C* es la fila 7 de espacio, *p* es *espacio*[4].*sig* y *q* es *disponible*. Los cursores se operan como se muestra a continuación; los apuntadores antiguos se representan por medio de líneas de trazo continuo y los nuevos por medio de líneas punteadas:



- Las líneas sólidas son los cursores antes de mover la celda y las líneas punteadas son los cursores después del movimiento

Módulo VII. TDA Lista mediante apuntadores

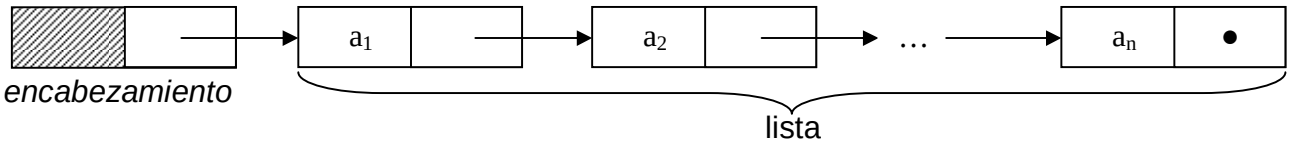
La implantación de una estructura de datos mediante apuntadores permite eludir el empleo de memoria contigua para almacenar una lista y, por tanto, también elude los desplazamientos de elementos para hacer inserciones o rellenar espacios creados por la eliminación de elementos. No obstante, por esto hay que pagar el precio de un espacio adicional para los apuntadores. Nótese que lo mismo sucede con las listas basadas en cursores.

Lista simplemente ligada lineal con encabezado

Esta implantación permite eludir el empleo de memoria contigua para almacenar una lista y, por tanto, también elude los desplazamientos de elementos para hacer inserciones o rellenar vacíos creados por la eliminación de elementos. No obstante, por esto hay que pagar el precio de un espacio adicional para los apuntadores.

Si la lista es a_1, a_2, …, a_n la celda (o nodo) que contiene a_i tiene un apuntador a la celda que contiene a a_{i+1} para i = 1, 2, …, n-1. La celda que contiene a_n posee un apuntador a **null** (dirección nula).

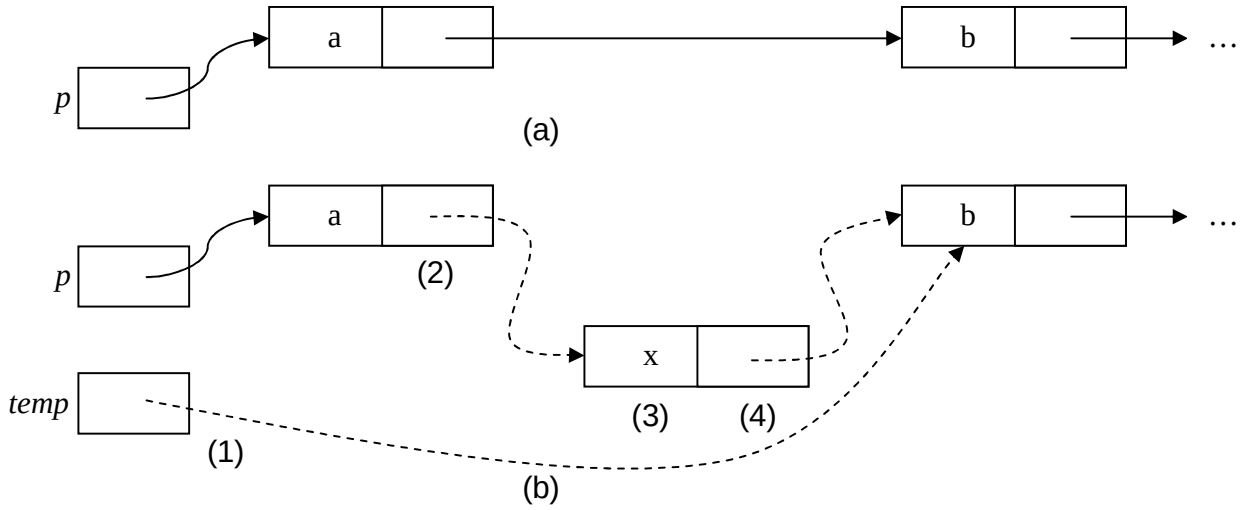
Existe también una celda de encabezamiento que apunta a la celda que contiene a_1; esta celda de encabezamiento no tiene ningún elemento. En el caso de una lista vacía, el apuntador del encabezamiento es **null** y no se tienen más celdas.



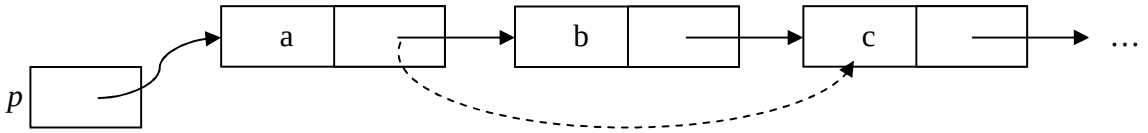
Como con los cursores, la posición *i* será un apuntador a la celda que contiene el apuntador a a*i* para *i* = 2, 3, …, *n*. La posición 1 es un apuntador al encabezamiento, y la posición FIN(*L*) es un apuntador a la última celda de *L*.

A diferencia de las implementaciones con arreglos y cursores, muchas de las operaciones no necesitan el parámetro *L*, la lista.

La operación INSERTA, para insertar un nodo con elemento *x*, opera por pasos como la siguiente figura; los apuntadores antiguos se representan por medio de líneas de trazo continuo y los nuevos por medio de líneas punteadas; *p* es un apuntador a la celda de la lista que contiene el apuntador a b;



La operación SUPRIME, para eliminar el nodo que contiene al elemento *b*, funciona como sigue:



Comparación de los métodos de listas ligadas y con arreglos

1. La realización con arreglos requiere especificar el tamaño máximo de una lista en tiempo de compilación. Si no es posible acotar la probable longitud de la lista, quizá deba optarse por una realización con apuntadores
2. Ciertas operaciones son más lentas en una realización que en otra. Por ejemplo INSERTA y SUPRIME tienen un número constante de pasos para una lista enlazada (igual con cursores), pero cuando se utiliza la realización con arreglos, requieren un tiempo proporcional al número de elementos que siguen. A la inversa, la ejecución de ANTERIOR y FIN requieren un tiempo constante con la realización con arreglos, pero se usa un tiempo proporcional a la longitud de la lista si se usan apuntadores.
3. Como principio general, los apuntadores se deben utilizar con gran cuidado y moderación; es posible “extraviar” bloques de memoria del heap del programa.
4. La realización con arreglos puede malgastar espacio, puesto que usa la cantidad máxima de espacio, independientemente del número de elementos que en realidad tiene la lista en un momento dado. La realización con apuntadores utiliza solo el espacio necesario para los elementos que actualmente tiene la lista, pero requiere espacio para el apuntador en cada celda. Así, cualquiera de los dos métodos podría usar más espacio que el otro, dependiendo de las circunstancias.

Es importante observar que el nodo de encabezado se crea tras la operación anula, y la memoria dinámica reservada para este encabezado debe ser liberada explícitamente, pues el encabezado es una variable apuntador y cualquier bloque de memoria reservado dinámicamente debe ser devuelto al heap de la misma forma.

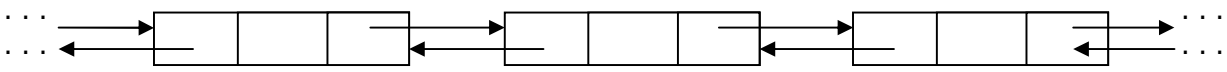
Lista simplemente ligada lineal sin encabezado

Es posible usar apuntadores como encabezamientos si se pretende aplicar operaciones así las inserciones y las eliminaciones al principio de la lista se manejan de manera especial.

Es posible un ahorro por la celda de encabezado, la implementación se asemeja a cursores, pero tiene un considerable aumento en su dificultad de implementación y eficiencia de ejecución, por la validación de casos especiales para insertar y eliminar al principio de la lista para reemplazar el encabezado; además es más difícil darle mantenimiento al código fuente, tanto el correspondiente a las operaciones de las listas, como al código que hace uso de las operaciones.

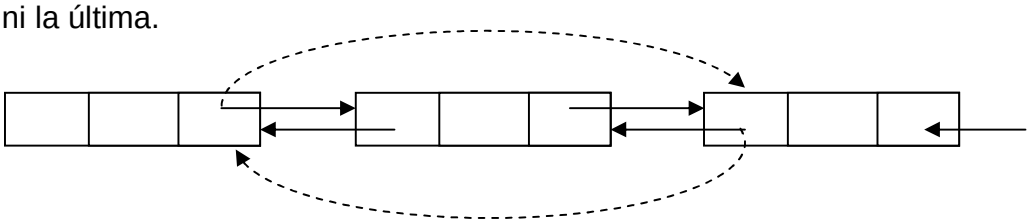
Lista doblemente ligada lineal con encabezado

En algunas aplicaciones puede ser deseable recorrer eficientemente una lista, tanto hacia adelante como hacia atrás. O, dado un elemento podría desearse determinar con rapidez el siguiente y el anterior. En tales situaciones, quizás se quisiera poner en cada celda de una lista un apuntador a la siguiente celda y otro a la anterior.



Otra ventaja importante de las listas doblemente enlazadas es que permiten usar un apuntador a la celda que contiene el *i*-ésimo elemento de una lista para representar la posición *i*, en vez de usar el apuntador a la celda anterior, que es menos natural. El único precio es la presencia de un apuntador adicional en cada celda, y los procedimientos algo más lentos para algunas de las operaciones básicas con listas.

A continuación se muestran los cambios causador en los apuntadores por ejecutar la operación SUPRIME; los apuntadores anteriores se representan con líneas de trazo continuo, y los nuevos con líneas punteadas, en el supuesto de que la celda suprimida no es la primera ni la última.



Lista doblemente ligada circular con encabezado

Es práctica común hacer que el encabezamiento de una lista doblemente enlazada sea una celda que efectivamente “cierre el círculo”. Esto es, que el campo *ant* del encabezado apunte a la última celda y que su campo *sig* apunte a la primera. De esta manera no se necesita hacer verificaciones de campos NULL.

Módulo VIII. Representación con Apuntadores del TDA Pila

Una pila dinámica con apuntadores puede implementarse en función de operaciones con listas simplemente ligadas, tal como se explicó en la sección de la Pila basada en arreglos. El único límite de esta estructura es el heap asignado al programa.

Una implementación específica para la pila, sin usar la librería para listas, ahorra algo de espacio en código objeto, pues las operaciones de listas, por ser de uso genérico proveen operaciones y realizan validaciones adicionales al ejecutar las operaciones.

La implementación puede ser con o sin encabezado, pero a diferencia de las listas genéricas, una pila sin encabezado no tiene complicación para implementar las operaciones METE y SACA, como en el caso de las listas sin encabezado, pues siempre se reemplaza el apuntador a la pila, y con la ventaja adicional de no preocuparse por liberar la memoria reservada para un encabezado.

Módulo IX. Representación con Apuntadores del TDA Cola

Igual que en el caso de las pilas, cualquier realización de listas es lícita para las colas. A diferencia de la pila dinámica, los límites de esta estructura son el heap asignado al programa, y la más importante, la necesidad de recorrer toda la lista de principio a fin, cada que se ejecuta la operación PONE_EN_COLA para que el nuevo nodo sea colocado al final.

Para aumentar la eficacia de PONE_EN_COLA, es posible aprovechar el hecho de que las inserciones se efectúan solo en el extremo posterior, de modo que se puede mantener un apuntador (o un cursor) al último elemento; lo anterior además de que una implementación específica para la cola, sin usar la librería para listas, ahorra espacio en código objeto.

Como en las listas de cualquier clase, también se mantiene un apuntador al frente de la lista; en las colas, ese apuntador es útil para ejecutar mandatos del tipo FRENTE o QUITA_DE_COLA.

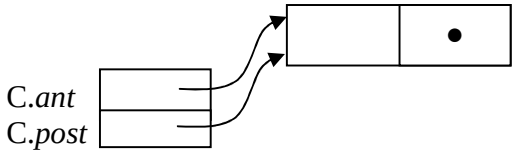
La implementación con encabezado contra la de sin encabezado, permite manejar convenientemente una cola vacía.

Se puede desarrollar una realización análoga basada en cursores, pero en el caso de las colas se dispone de una representación orientada a arreglos, mejor que la que se puede obtener por imitación directa de apuntadores mediante cursores.

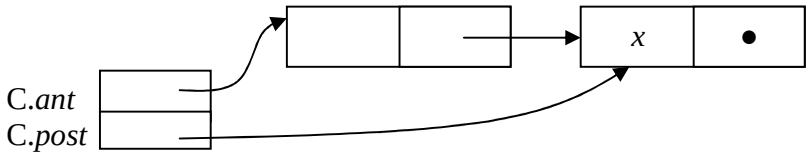
Por tanto, es posible definir una cola como una estructura que consiste en apuntadores al extremo anterior de la lista y al extremo posterior. La primera celda de una cola es una celda de encabezamiento cuyo campo elemento se ignora.

A continuación se muestran los resultados originados por la sucesión de las operaciones de colas:

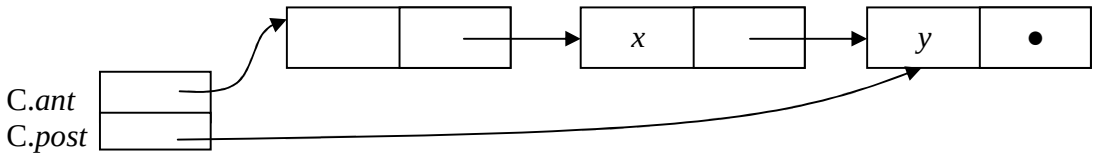
ANULA(*C*)



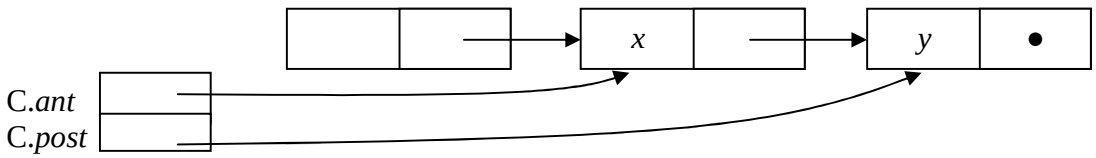
PONE_EN_COLA(*x*, *C*)



PONE_EN_COLA(*y*, *C*)



QUITA_DE_COLA(*C*)



Módulo X. Árbol Binario de Búsqueda

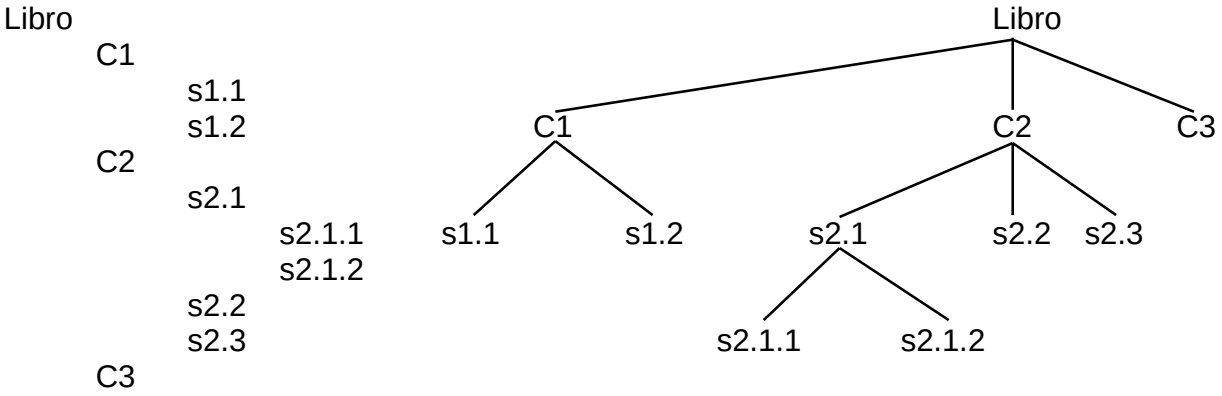
Árboles

Un árbol impone una estructura jerárquica sobre una colección de objetos. Los árboles genealógicos y los organigramas son ejemplos comunes de árboles. Entre otras cosas, los árboles son útiles para analizar circuitos eléctricos y para representar la estructura de fórmulas matemáticas. También se presentan naturalmente en diversas áreas de computación. Por ejemplo, se usan para organizar información en sistemas de bases de datos y para representar la estructura sintáctica de un programa fuente en los compiladores.

Árbol.- colección de elementos llamados *nodos*, uno de los cuales se distingue como *raíz*, junto con una relación (de "paternidad") que impone una estructura jerárquica sobre los nodos. Formalmente se, se puede definir de manera recursiva como sigue:

1. Un solo nodo es, por si mismo, un árbol. Ese nodo es también la raíz de dicho árbol.
2. Supóngase que n es un nodo y que A_1, A_2, …, A_k son árboles con raíces n_1, n_2, …, n_k, respectivamente. Se puede construir un nuevo árbol haciendo que n se constituya en el *padre* de los nodos n_1, n_2, …, n_k. En dicho árbol, n es la raíz y A_1, A_2, …, A_k son los *subárboles* de la raíz. Los nodos n_1, n_2, …, n_k reciben el nombre de *hijos* del nodo n.

A veces, conviene incluir entre los árboles el *árbol nulo*, un "árbol" sin nodos que se representa mediante Λ.



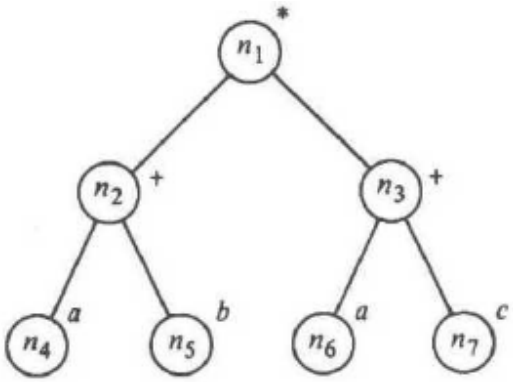
Si n_1, n_2, …, n_k, es una sucesión de nodos de un árbol tal que n_i es el padre de n_i+1 para 1≤i<k, entonces la secuencia se denomina *camino* del nodo n_1 al nodo n_k. La *longitud* del camino es el número de nodos del camino menos 1. Por tanto, hay un camino de longitud cero de cualquier nodo a sí mismo.

Si existe un camino de un nodo *a* a otro *b*, entonces *a* es un *antecesor* de *b*, y *b* es un *descendiente* de *a*. Un antecesor o un descendiente de un nodo que no sea él mismo recibe el nombre de *antecesor propio* o *descendiente propio*, respectivamente. En un árbol, la raíz es el único nodo que no tiene antecesores propios. Un nodo sin descendientes propios se denomina *hoja*. Un subárbol es un nodo junto con todos sus descendientes.

La *altura* de un nodo en un árbol es la longitud del camino más largo de ese nodo a una hoja.

A menudo los hijos de un nodo se ordenan de izquierda a derecha. El orden de izquierda a derecha de los *hermanos* (hijos del mismo nodo) se puede extender para comparar dos nodos cualesquiera entre los cuales no exista la relación antecesor-descendiente. Si *a* y *b* son hermanos y *a* está a la izquierda de *b*, entonces todos los descendientes de *a* están a la izquierda de todos los descendientes de *b*.

A menudo es útil asociar una *etiqueta*, o valor, a cada nodo de un árbol, siguiendo la misma idea con que se asoció un valor a un elemento de una lista. Esto es, la etiqueta de un nodo no será el nombre del nodo, sino un valor “almacenado” en él. En algunas aplicaciones, incluso se cambiaría la etiqueta de un nodo sin modificar su nombre. Una analogía útil a este respecto es: árbol es a lista como etiqueta es a elemento y nodo es a posición.



<u>Recorridos en preorden, enorden y postorden</u>

Existen varias maneras útiles de ordenar sistemáticamente todos los nodos de un árbol. Los tres ordenamientos más importantes se llaman son preorden u orden previo (preorder), enorden u órden simétrico (inorder) y postorden u orden posterior (postorder); tales ordenamientos se definen recursivamente como sigue:

- Si un árbol A es nulo, entonces la lista vacía es el listado de los nodos de A en los órdenes previo, simétrico y posterior.
- Si A contiene un solo nodo, entonces ese nodo constituye el listado de los nodos de A en los órdenes previo, simétrico y posterior.

Si ninguno de los anteriores es el caso, sea *A* un árbol con raíz n y subárboles A_1 y A_2:

1. El *listado en orden previo* (o *recorrido en orden previo*) de los nodos de *A* está formado por la raíz de *A*, seguida de los nodos de A_1 en orden previo y luego por los nodos de A_2 en orden previo.
2. El listado en *orden simétrico* de los nodos de *A* está constituido por los nodos de A_1 en orden simétrico, seguidos de *n* y luego por los nodos de A_2 en orden simétrico.
3. El listado en *orden posterior* de los nodos de *A* tiene los nodos de A_1 en orden posterior, luego los nodos de A_2 en orden posterior y por último la raíz *n*.

Cuando se recorre un árbol, se prefiere listar las etiquetas de los nodos, en vez de sus nombres. El listado de las etiquetas de la figura anterior será entonces:

a) en orden previo o representación *prefija*: *+ab+ac
b) en orden posterior, conocido como representación *postfija* (o *polaca*): ab+ac+*
c) en orden simétrico o representación infija: a+b * a+c

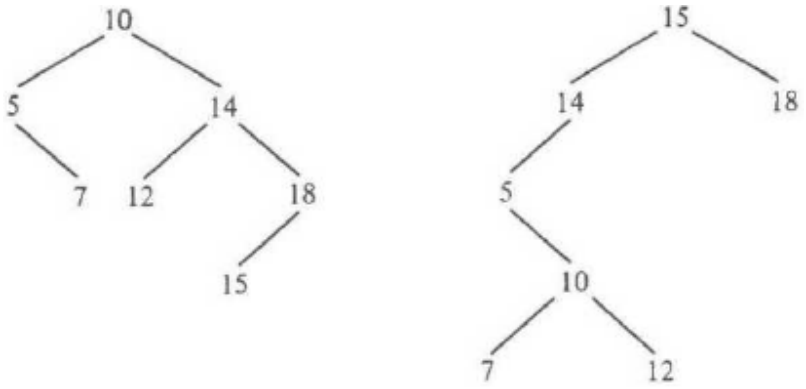
Árbol Binario de Búsqueda

Un *conjunto* es una colección de *miembros* o elementos, la cual no puede contener dos copias de un mismo elemento.

Un *árbol binario de búsqueda* es una estructura de datos básica para la representación de conjuntos cuyos elementos están clasificados de acuerdo con algún orden lineal. Como de costumbre, se designará ese orden por medio del signo <.

Esta estructura de datos es útil cuando se tiene un conjunto de elementos de un universo de gran tamaño. Un árbol binario de búsqueda puede manejar las operaciones de conjuntos INSERTA, SUPRIME, MIEMBRO y MIN, tomando en promedio $O(\log n)$ pasos por operación para un conjunto de *n* elementos.

Un *árbol binario de búsqueda* es un árbol binario en el cual los nodos están etiquetados con elementos de un conjunto. La propiedad importante de este tipo de árboles es que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo *x* son menores que el elemento almacenado en *x*, y todos los elementos almacenados en el subárbol derecho de *x* son mayores que el elemento almacenado en ese sitio. Esta condición, conocida como *propiedad del árbol binario de búsqueda*, se cumple para todo nodo de un árbol binario de búsqueda, incluyendo la raíz.



(a) (b)

Fig 5.1. Dos árboles binarios de búsqueda

Las siguientes son operaciones para el TDA Árbol:

1. El procedimiento ANULA(*A*), hace que el árbol nulo sea el valor de la variable árbol *A*.
2. ESTA_VACIO(*A*) devuelve *verdadero* si el árbol no tiene nodos; de lo contrario *falso*.
3. El procedimiento INSERTA(*x*, *A*), hace de *x* un miembro de *A*. Si *x* ya es miembro de *A*, entonces INSERTA(*x*, *A*) no modifica el árbol A.
4. SUPRIME(*x*, *A*) elimina *x* de *A*. Si *x* no está originalmente en *A*, SUPRIME(*x*, *A*) deja sin cambios al árbol *A*.
5. La función MIEMBRO(*x*, *A*) toma el árbol *A* y el objeto *x*, cuyo tipo es el de los elementos de *A*, y devuelve un valor booleano; verdadero si *x* pertenece a *A* y falso de lo contrario.
6. ETIQUETA(*n*) devuelve la etiqueta del nodo *n* en un árbol. Sin embargo, no se requiere que haya etiquetas definidas para cada árbol.
7. La función ENCUENTRA(*x*, *A*) devuelve el nombre del (único) nodo del cual *x* es miembro.
8. La función MIN(*A*) devuelve el elemento menor del árbol *A*. También se utiliza la función MAX, cuyo significado es obvio.
9. La función SUPRIME_MIN(*A*) elimina el elemento más pequeño de un árbol no vacío y devuelve el valor del elemento eliminado.
10. PADRE(n, *A*) devuelve el padre del nodo *n* en el árbol *A*. Si *n* es la raíz, que no tiene padre, de devuelve Λ. En este contexto, Λ es un “nodo nulo”, que se usa como señal de que se ha salido del árbol.

El pseudocódigo de algunas operaciones se muestra a continuación:

```
typedef struct Nod{
    TipoElemento elemento
    struct Nod* hijo_izq
    struct Nod* hijo_der
}Nodo

typedef Nodo* Arbol
```

```
booleano miembro(TipoElemento x, Arbol a)
inicio
    si (a = NULO) entonces
        regresa FALSO
    de lo contrario si (x = a->elemento) entonces
        regresa VERDADERO
    de lo contrario si (x < a->elemento) entonces
        regresa miembro(x, a->hijo_izq)
    de lo contrario //x > a->elemento
        regresa miembro(x, a->hijo_der)
fin

vacio inserta(TipoElemento x, Arbol* a)  //Arbol* implica que puede modificar “a”
inicio
    si (a = NULO) entonces inicio
        solicitar memoria para el nodo a
        a->elemento:=x
        a->hijo_izq:=NULO
        a->hijo_der:=NULO
    fin
    de lo contrario si (x < a->elemento) entonces
        inserta(x, a->hijo_izq)
    de lo contrario si (x > a->elemento) entonces
        inserta(x, a->hijo_der)
    //si x = a->elemento hacer nada; x ya está en el árbol
Fin

TipoElemento suprimeMin(Arbol* a)
inicio
    TipoElemento x
    si (a->hijo_izq = NULO) entonces inicio
        x:=a->elemento
        a:=a->hijo_der  //y devolver la memoria de “a” al heap
        regresa x
    fin
    de lo contrario inicio
        //el nodo apuntado por "a" tiene una hijo izquierdo
        regresa suprimeMin(a->hijo_izq)
    fin
fin
```

```
vacio suprime(TipoElemento x, Arbol* a)
inicio
    si (a <> NULO) entonces inicio
        si (x < a->elemento) entonces
            suprime(x, a->hijo_izq)
        de lo contrario si (x > a->elemento) entonces
            suprime(x, a->hijo_der)
        //si se llega aquí, x es el nodo apuntado por "a"
        de lo contrario si (a->hijo_izq = NULO) Y (a->hijo_der = NULO) entonces
            a:=NULO //suprime la hoja que contiene a x
        de lo contrario si (a->hijo_izq = NULO) entonces
            a:=a->hijo_der
        de lo contrario si (a->hijo_der = NULO) entonces
            a:=a->hijo_izq;
        de lo contrario //ambos nodos están presentes
            a->elemento:=suprimeMin(a->hijo_der)
    fin
fin
```

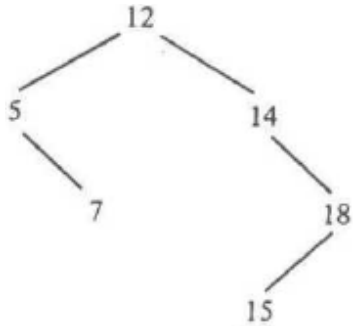
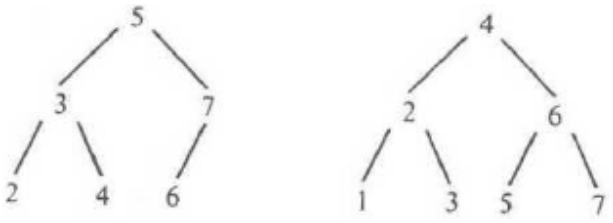


Fig. 5.2. Árbol de la figura 5.1(a) después de suprimir 10.

Módulo XI. El TDA Árbol AVL

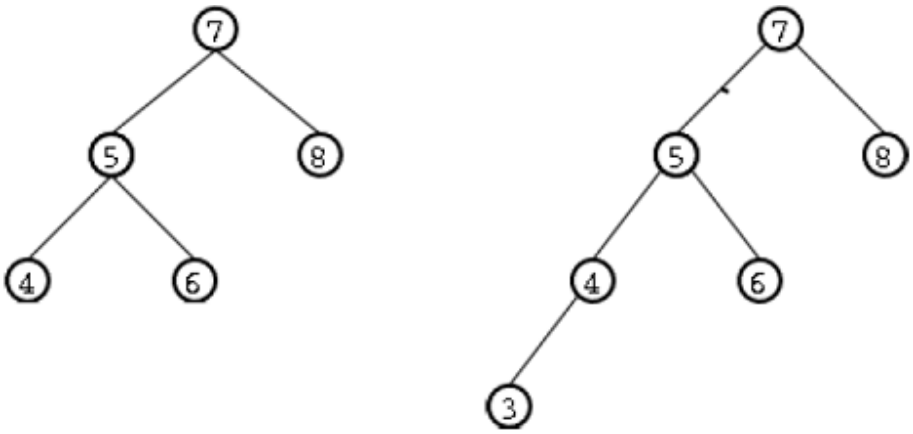
El árbol AVL ó *árbol balanceado por altura*, es un árbol binario de búsqueda en el cual no se permite que las alturas de dos hermanos difieran en más de uno.



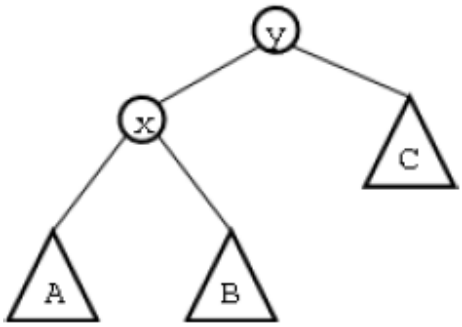
(a) (b)

Árbol completo

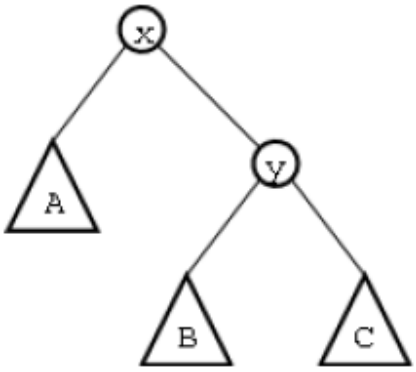
El tiempo de ejecución de las operaciones sobre estos árboles es, a lo sumo O(log(n)) en el peor caso; donde el peor caso para un árbol binario sin balancear es O(n). El problema a resolver es evitar que la inserción de un nuevo nodo desequilibre el árbol, tal como sucedería con la inserción ordinaria en un árbol binario de búsqueda; tal como sucedería al insertar el nodo “3” en el siguiente árbol, donde los hermanos 5 y 8 quedaron en desequilibrio por “diferencia de altura” > 1:



La rotación de un árbol servirá para balancear el árbol; en la siguiente figura se ilustra una *rotación simple izquierda* (el árbol de más altura es el izquierdo con raíz x); obsérvese que después de la rotación, se conserva la propiedad x < y:

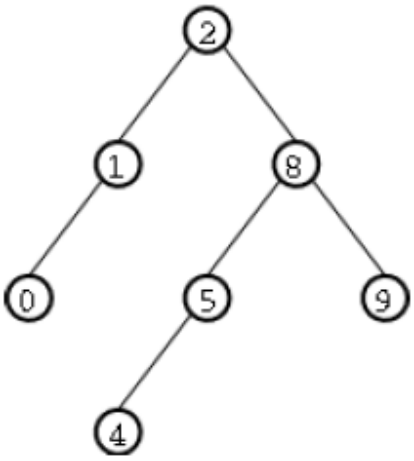


Árbol antes de la rotación

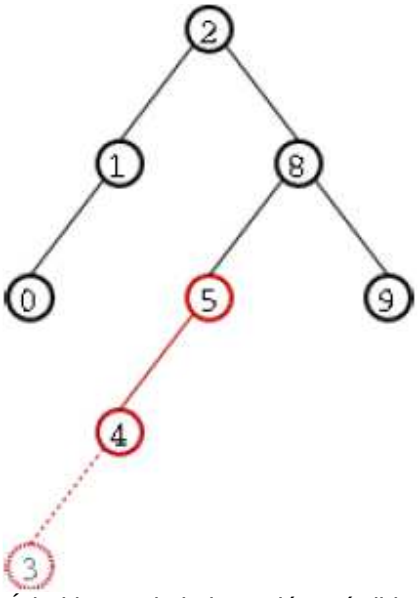


Árbol después de la rotación

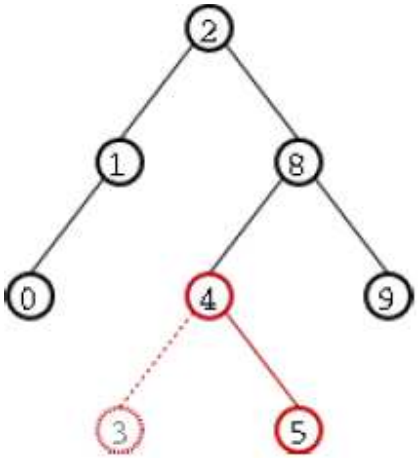
Ejemplo con rotación simple izquierda:



Árbol antes de insertar nodo 3

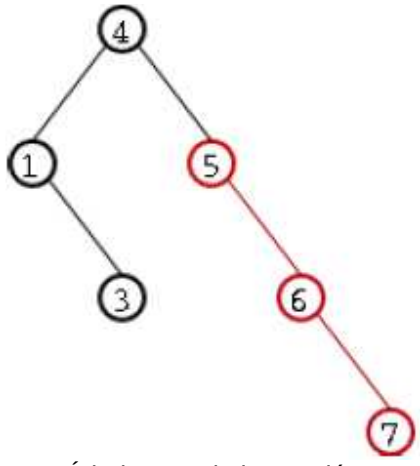


Árbol luego de la inserción: pérdida de la propiedad de equilibrio

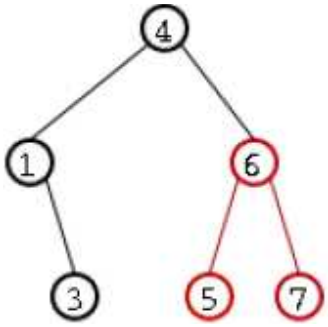


Restablecimiento de la propiedad de equilibrio mediante rotación simple del nodo 5

Ejemplo con rotación simple derecha:



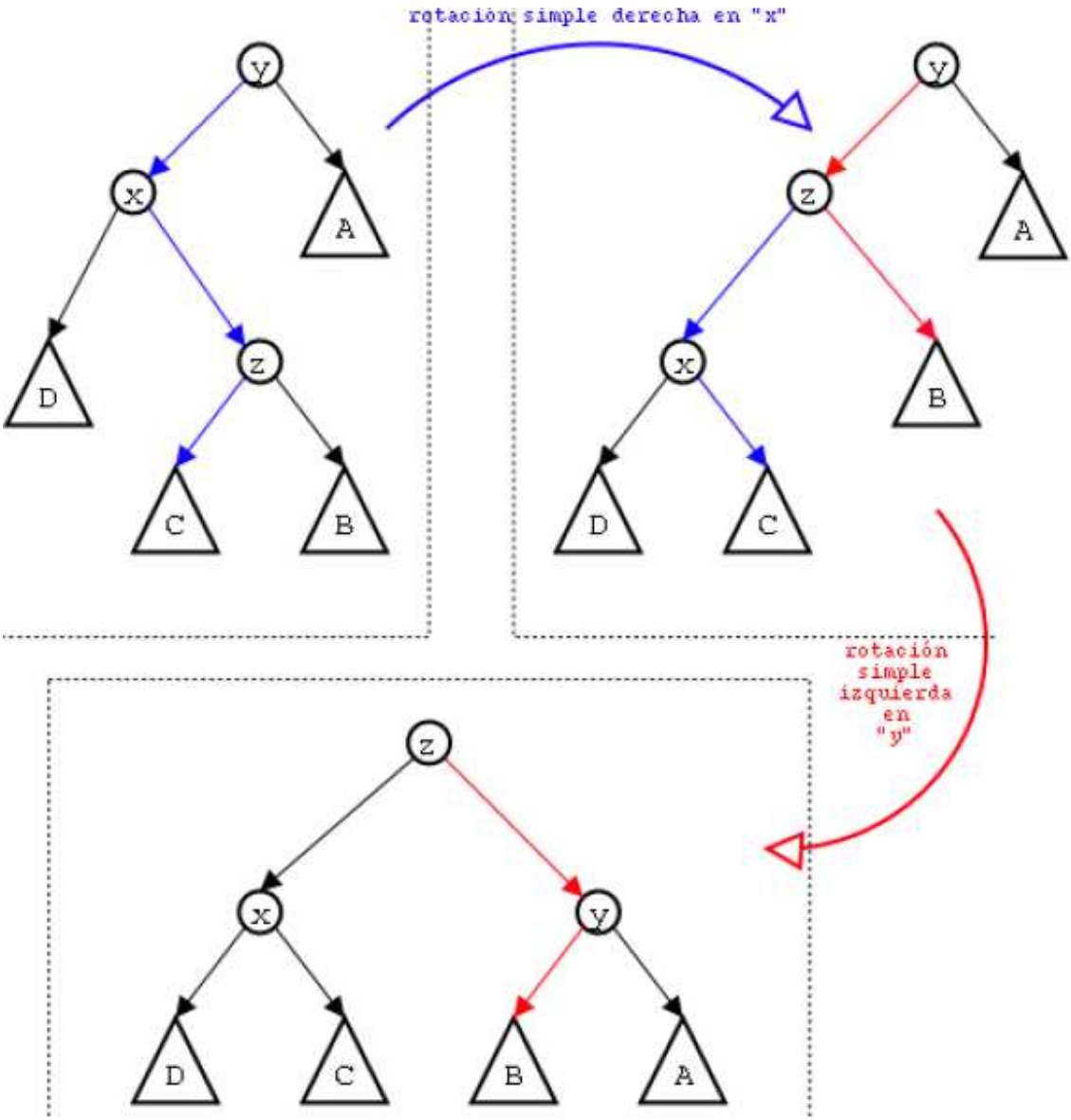
Árbol antes de la rotación



Árbol después de la rotación simple del nodo 6

Hay situación en las que tenemos desequilibrios en zig-zag, en tales casos aplica la *rotación doble* la cual, análogamente a la rotación simple, puede ser izquierda o derecha según el caso. Una rotación doble consta de dos rotaciones simples.

Una rotación doble izquierda consta de una rotación simple derecha, seguida de una rotación simple izquierda, como se ilustra a continuación:



Rotación doble izquierda

Para realizar el balanceo, los procedimientos INSERTA y SUPRIME quedan de la siguiente manera:

```
vacio inserta(TipoElemento x, Arbol* a)  //Arbol* implica que puede modificar “a”
inicio
    si (a = NULO) entonces inicio
        solicitar memoria para el nodo a
        a->elemento:=x
        a->hijo_izq:=NULO
        a->hijo_der:=NULO
        a->altura:=0
    fin
    de lo contrario inicio
        si (x < a->elemento) entonces
            inserta(x, a->hijo_izq)
        de lo contrario si (x > a->elemento) entonces
            inserta(x, a->hijo_der)
        balancear(a)
        actualizarAltura(a)
    fin
    //si x = a->elemento hacer nada; x ya está en el árbol
Fin

vacio suprime(TipoElemento x, Arbol* a)
inicio
    si (a <> NULO) entonces inicio
        si (x < a->elemento) entonces
            suprime(x, a->hijo_izq)
        de lo contrario si (x > a->elemento) entonces
            suprime(x, a->hijo_der)
        de lo contrario inicio
            //si se llega aquí, x es el nodo apuntado por "a"
            si (a->hijo_izq = NULO) Y (a->hijo_der = NULO) entonces
                a:=NULO //suprime la hoja que contiene a x
            de lo contrario si (a->hijo_izq = NULO) entonces
                a:=nodo->hijo_der
            de lo contrario si (a->hijo_der = NULO) entonces
                a:=nodo->hijo_izq;
            de lo contrario //ambos nodos están presentes
                a->elemento:=suprimeMin(a->hijo_der)
            si (a <> NULO) entonces inicio
                balancear(a)
                actualizarAltura(a)
            fin
        fin
    fin
fin
```

Nota: las operaciones para balance del árbol y actualización de altura de nodos se encuentran en la carpeta “Ejemplos EDA\19 TDA Arbol AVL”

Módulo XII. El TDA Grafo

En los problemas originados en ciencias de la computación, matemáticas, ingeniería y muchas otras disciplinas, a menudo es necesario representar relaciones arbitrarias entre objetos de datos. Los grafos dirigidos y los no dirigidos son modelos naturales de tales relaciones.

Un *grafo dirigido G* consiste en un conjunto de vértices *V* y un conjunto de arcos *A*. Los vértices se denominan también *nodos* o *puntos*; lo arcos pueden llamarse *arcos dirigidos* o *líneas dirigidas*. Un arco es un par ordenado de vértices (*v*, *w*); *v* es la cola y *w* la cabeza del arco. El arco (*v*, *w*) se expresa a menudo como $v \rightarrow w$ y se representa como:

Los vértices de un grafo dirigido pueden usarse para representar objetos, y los arcos, relaciones entre los objetos. Por ejemplo, los vértices pueden representar ciudades, y los arcos, vuelos aéreos de una ciudad a otra. En otro ejemplo, un grafo dirigido puede emplearse para representar el flujo de control en un programa de computador. Los vértices representan bloques básicos, y los arcos, posibles transferencia del flujo de control.

Un *camino* en un grafo dirigido es una secuencia de vértices $v_1, v_2, \ldots, v_n$, tal que $v_1 \rightarrow v_2$, $v_2 \rightarrow v_3, \ldots, v_{n-1} \rightarrow v_n$ son arcos. Este camino *va del* vértice v_1 *al* vértice v_n, *pasa por* los vértices $v_2, v_3, \ldots, v_{n-1}$ y termina en el vértice v_n. La *longitud* de un camino es el número de arcos en ese camino, en este caso, *n*-1. Como caso especial, un vértice sencillo, *v*, por sí mismo denota un camino de longitud cero de *v* a *v*. En la figura 6.1, la secuencia 1, 2, 4 es un camino de longitud 2 que va del vértice 1 al vértice 4.

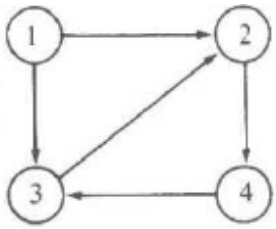


Fig. 6.1. Grafo dirigido

Un camino es *simple* si todos sus vértices, excepto tal vez el primero y el último son distintos. Un *ciclo* simple es un camino simple de longitud por lo menos uno, que empieza y termina en el mismo vértice. En la figura 6.1, el camino 3, 2, 4, 3 es un ciclo de longitud 3.

En muchas aplicaciones es útil asociar información al los vértices y arcos de un grafo dirigido. Para este propósito es posible usar un *grafo dirigido etiquetado*, en el cual cada arco, cada vértice o ambos pueden tener una etiqueta asociada. Una etiqueta puede ser un nombre, un costo o un valor de cualquier tipo de datos dado.

La figura 6.2 muestra un grafo dirigido etiquetado en el que cada arco está etiquetado con una letra que causa una transición de un vértice a otro. En un grafo dirigido etiquetado, un vértice puede tener a la vez un nombre y una etiqueta. A menudo se empleará la etiqueta del vértice como si fuera el nombre. Así, los números de la figura 6.2 pueden interpretarse como nombres o como etiquetas de vértices.

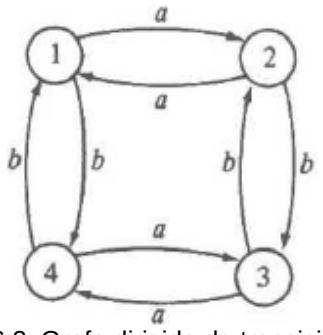


Fig. 6.2. Grafo dirigido de transiciones.

Representación en Matriz de Adyacencia

Para representar un grafo dirigido se pueden emplear varias estructuras de datos; la selección apropiada depende de las operaciones que se aplicarán a los vértices y a los arcos del grafo. Una representación común para un grafo dirigido $G = (V, A)$ es la *matriz de adyacencia*. Supóngase que $V = \{1, 2, \ldots, n\}$. La matriz de adyacencia para G es una matriz A de dimensión n x n, de elementos booleanos, donde $A[i, j]$ es verdadero si, y solo si, existe un arco que vaya del vértice i al j. Con frecuencia se exhibirán matrices de adyacencias con 1 para verdadero y 0 para falso.

En la representación con una matriz de adyacencia, el tiempo de acceso requerido a un elemento es independiente del tamaño de V y A. Así , la representación con matriz de adyacencia es útil en los algoritmos para grafos, en los cuales suele ser necesario saber si un arco dado está presente.

La figura 6.3, muestra la matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2. Aquí, el tipo de la etiqueta es un carácter, y un espacio representa la ausencia de un arco.

	1	2	3	4
1		a		b
2	a		b	
3		b		a
4	b		a	

Fig. 6.3. Matriz de adyacencia etiquetada para el grafo dirigido de la figura 6.2.

La principal desventaja de usar una matriz de adyacencia para representar un grafo dirigido es que requiere un espacio $\Omega(n2)$ aun si el grafo dirigido tiene menos de n^2 arcos. Solo leer o examinar la matriz puede llevar un tiempo $O(n^2)$, lo cual invalidaría los algoritmos $O(n)$ para la manipulación de grafos dirigidos con $O(n)$ arcos.

Operaciones sobre Grafos

Se podría definir un TDA que correspondiera formalmente al grafo dirigido y estudiar las implantaciones de sus operaciones. Las operaciones más comunes en grafos dirigidos incluyen:

1) la lectura de la etiqueta de un vértice o un arco,
2) la inserción o supresión de vértices y arcos y,
3) el recorrido de arcos desde la cola hasta la cabeza.

Las últimas operaciones requieren más cuidado. Con frecuencia, se encuentran proposiciones informales de programas como:

```
Para (cada vértice w adyacente al vértice v) hacer
    //alguna acción sobre w
```

Para obtener esto, es necesaria la noción de un tipo *índice* para el conjunto de vértices adyacentes a algún vértice *v*. Por ejemplo, si las listas de adyacencia se usan para representar el grafo, un índice es, en realidad, una posición en la lista de adyacencia de v. Si se usa una matriz de adyacencia, un índice es un entero que representa un vértice adyacente. Se requieren las tres operaciones siguientes en grafos dirigidos:

1. PRIMERO(*v*), que devuelve el índice del primer vértice adyacente a *v*. Se devuelve un vértice nulo Λ si no existe ningún vértice adyacente a v.
2. SIGUIENTE(*v*, *i*), que devuelve el índice posterior al índice *i* de los vértices adyacentes a *v*. Se devuelve Λ si *i* es el último vértice de los vértices adyacentes a *v*.
3. VERTICE(*v*, *i*), que devuelve el vértice cuyo índice *i* está entre los vértices adyacentes a v.

Un primer acercamiento a la construcción y análisis de un grafo es el algoritmo de Dijkstra, el cual encuentra la ruta más corta hacia un nodo desde un vértice origen hacia cualquier destino.

```
vacio dijkstra()
inicio
    S:={1}  //conjunto de vértices verificados
    para i:=2 hasta n hacer
        D[i]:=C[1,i]  //longitud del camino más corto para vértice i
    para i:=1 hasta n-1 hacer inicio
        //V es el conjunto de todos los vértices del grafo
        elige un vértice w en V-S tal que D[w] sea un mínimo
        agrega w a S
        para cada vértice v en V-S hacer
            D[v]:=min(D[v],D[w]+C[w,v])
    fin
fin
```

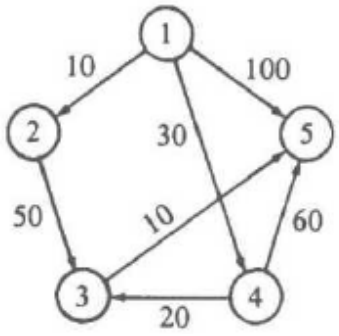


Fig. 6.4. Grafo dirigido con arcos etiquetados

Aplicando el algoritmo sobre el grafo de la figura 6.4 se obtiene la siguiente tabla:

Iteracion	S	w	D[2]	D[3]	D[4]	D[5]
inicial	{1}	-	10	inf	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Representación en Lista de Adyacencia

Para evitar la desventaja en uso de espacio de la matriz de adyacencia, se puede utilizar otra representación común para un grafo dirigido $G = (V, A)$ llamada representación con *lista de adyacencia*. La lista de adyacencia para un vértice i es una lista, en algún orden, de todos los vértices adyacentes a i.

Se puede representar G por medio de un arreglo *CABEZA*, donde *CABEZA*[i] es un apuntador a la lista de adyacencia del vértice i. La representación con lista de adyacencia de un grafo dirigido requiere un espacio proporcional a la suma del número de vértices más el número de arcos; se usa bastante cuando el número de arcos es mucho menor que n^2. Sin

embargo, una desventaja potencial de la representación con lista de adyacencia es que puede llevar un tiempo $O(n)$ determinar si existe un arco del vértice i al vértice j, ya que puede haber $O(n)$ vértices en la lista de adyacencia para el vértice i.

La figura 6.5 muestra una representación con lista de adyacencia para el grafo dirigido de la figura 6.1, donde se usan listas enlazadas sencillas. Si los arcos tienen etiquetas, éstas podrían incluirse en las celdas de la lista ligada.

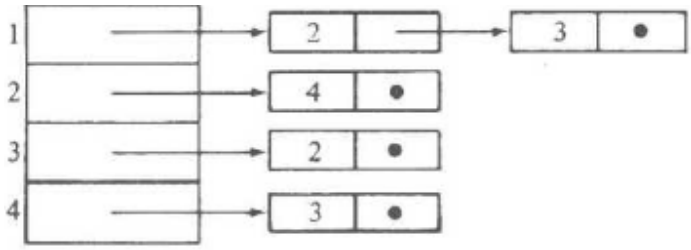


Fig. 6.5. Representación con lista de adyacencia para el grafo dirigido de la figura 6.1

Si hay inserciones y supresiones en la lista de adyacencias, sería preferible tener el arreglo *CABEZA* apuntando a celdas de encabezamiento que no contengan vértices adyacentes (para no tener los problemas de la falta de encabezado).

Por otra parte, si se espera que el grafo permanezca fijo, sin cambios (o con muy pocos) en la listas de adyacencias, sería preferible que *CABEZA*[*i*] fuera un cursor a un arreglo *ADY*, donde *ADY*[*CABEZA*[*i*]], *ADY*[*CABEZA*[*i*]+1], y así sucesivamente, contuvieran los vértices adyacentes al vértice *i*, hasta el ponto en ADY donde se encuentra por primera vez una *POSICION_NULA* (en lenguajes como C un -1, en lenguajes como Pascal un 0), el cual marca el fin de la lista de vértices adyacentes a *i*. Por ejemplo la figura 6.1 puede representarse con la figura 6.6.

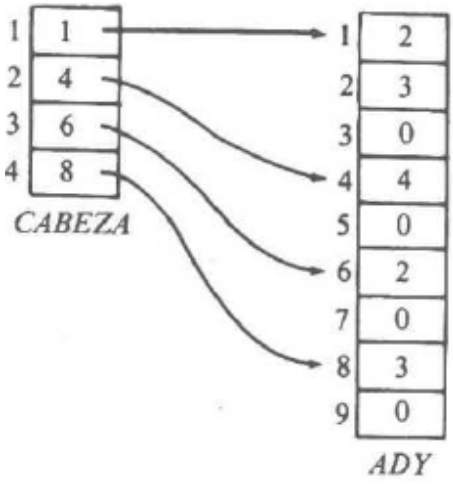


Fig. 6.6. Otra representación con lista de adyacencia.

Recorridos en Grafos

Búsqueda en profundidad:

Supóngase que se tiene un grafo dirigido *G* en el cual todos los vértices están marcados en principio como *no visitados*. La búsqueda en profundidad trabaja seleccionando un vértice *v* de *G* como vértice de partida; *v* se marca como visitado. Después, se recorre cada vértice adyacente a *v* no visitado, usando recursivamente la búsqueda en profundidad. Una vez que se han visitado todos los vértices que se pueden alcanzar desde *v*, la búsqueda de *v* está completa. Si algunos vértices quedan sin visitar, se selecciona alguno de ellos como nuevo vértice de partida, y se repite este proceso hasta que todos los vértices de *G* se hayan visitado.

Esta técnica se conoce como búsqueda en profundidad porque continúa buscando en la dirección hacia adelante (más profunda) mientras sea posible.

```
vacio busquedaProfundidad(Vertice v)
inicio
    Vertice w
    //marca es un arreglo que indica si un vértice ha sido visitado
    marca[v]:=VISITADO
    //L es la lista de vértices adyacentes a un vértice
    para cada vértice w en L[v] hacer
        si marca[w] = NO_VISITADO entonces
            busquedaProfundidad(w)
fin
```

Durante un recorrido en profundidad de un grafo dirigido como el de la figura 6.7, cuando se recorren ciertos arcos, llevan a vértices sin visitar. Los arcos que llevan a vértices nuevos se conocen como *arcos de árbo*l y forman un *bosque abarcador en profundidad* (figura 6.8) para el grafo dirigido dado.

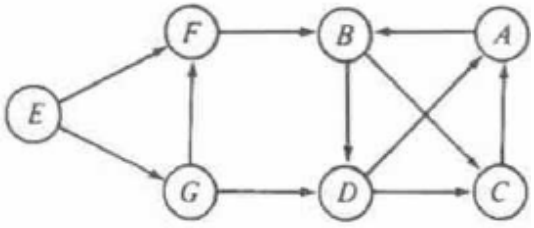


Fig. 6.7 Grafo dirigido

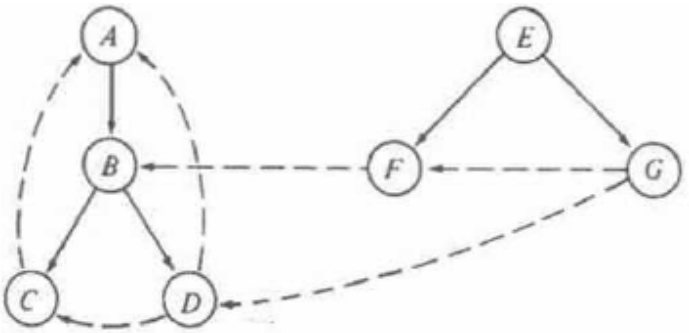


Fig 6.8. Bosque abarcador en profundidad para la figura 6.7

Un *grafo dirigido acíclico*, o *gda*, es un grafo dirigido sin ciclos. Cuantificados en función de las relaciones que representan, los gda son más generales que los árboles, pero menos que los grafos dirigidos arbitrarios. La figura 6.9 muestra un ejemplo de un árbol, un gda, y un grafo dirigido con un ciclo.

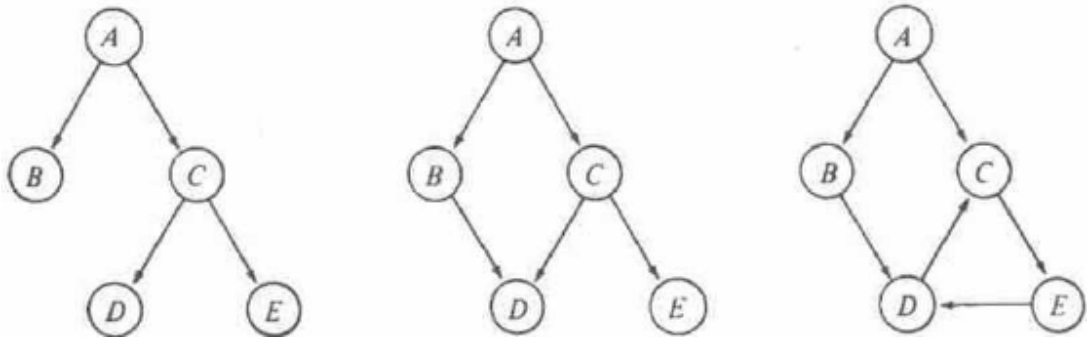


Fig. 6.9 Tres grafos dirigidos.

Un proyecto grande suele dividirse en una colección de tareas más pequeñas, algunas de las cuales se han de realizar en ciertos órdenes específicos, de modo que se pueda culminar el proyecto total. Por ejemplo, una carrera universitaria puede contener cursos que requieran otros como prerrequisitos. Los gda pueden emplearse para modelar de manera natural estas situaciones. Por ejemplo, podría tenerse un arco del curso C al curso D si C fuera un prerrequisito de D.

La figura 6.10 muestra un gda con la estructura de prerrequisitos de cinco cursos. El curso C3, por ejemplo, requiere los cursos C1 y C2 como prerrequisitos.

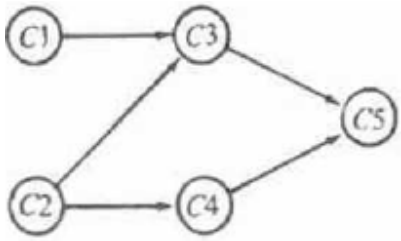


Fig. 6.10. Gda de prerrequisitos

Un grafo no dirigido $G = \{V, A\}$ consta de un conjunto finito de vértices V y de un conjunto de aristas A. Se diferencia de un grafo dirigido en que cada arista en A es un par no ordenado de vértices. Si (v, w) es una arista no dirigida, entonces $(v, w) = (w, v)$.

Los grafos no dirigidos se emplean en distintas disciplinas para modelar relaciones simétricas entre objetos. Los objetos se representan por los vértices del grafo, y dos objetos están conectados por una arista si están relacionados entre sí.

La búsqueda en profundidad también aplica en grafos no dirigidos, esta búsqueda aplicada al grafo de la figura 6.11(a) produce el bosque abarcador de la figura 6.11(b); las aristas de árbol se muestran con líneas de trazo continuo y las otras con líneas de puntos.

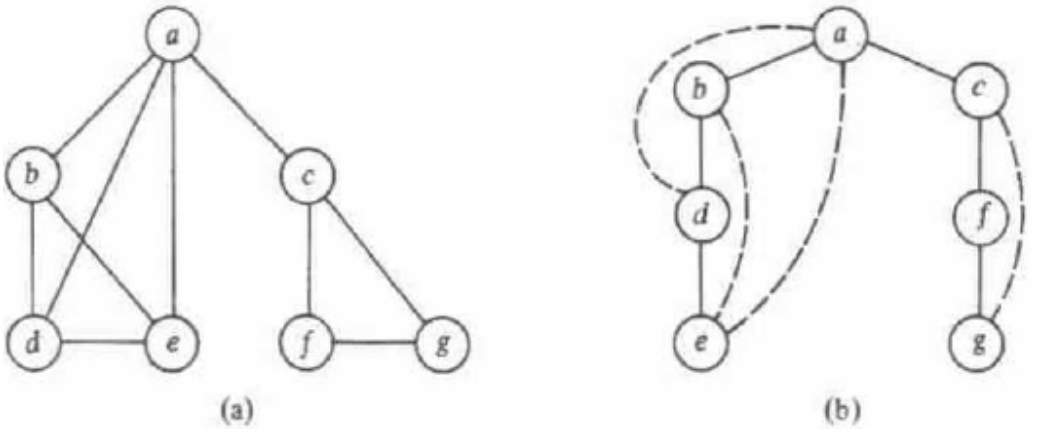


Fig 6.11. Un grafo y su búsqueda en profundidad

Búsqueda en amplitud:

Otra forma sistemática de visitar los vértices se conoce como *búsqueda en amplitud*. Este enfoque se denomina “en amplitud” porque desde cada vértice v que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a v. Esta estrategia de búsqueda se puede aplicar tanto a grafos dirigidos como a no dirigidos

Igual que en la búsqueda en profundidad, al realizar una búsqueda en amplitud se puede construir un bosque abarcador. En este caso, se considera la arista (x, y) como una arista de árbol si el vértice y es el que se visitó primero partiendo del vértice x.

El algoritmo de búsqueda en amplitud se muestra a continuación

```
vacio busquedaAmplitud(Vertice v)
inicio
    ColaDeVertice cola
    Vertice x,y
    marca[v]:=VISITADO
    poneEnCola(v, cola)
    mientras no estaVacia(cola) hacer inicio
        x:=frente(cola)
        quitaDeCola(cola)
        para cada vértice y adyacente a x hacer
            si marca[y] = NO_VISITADO entonces inicio
                marca[y]:=VISITADO
                poneEnCola(y, cola)
                inserta((x, y), cola)
            fin
    fin
fin
```

El árbol abarcador en amplitud del grafo *G* de la figura 6.11(a) se muestra en la figura 6.12. Se supone que la búsqueda empieza en el vértice *a*. Se ha dibujado la raíz del árbol en la parte superior, y los hijos, de izquierda a derecha, de acuerdo con el orden en que fueron visitados.

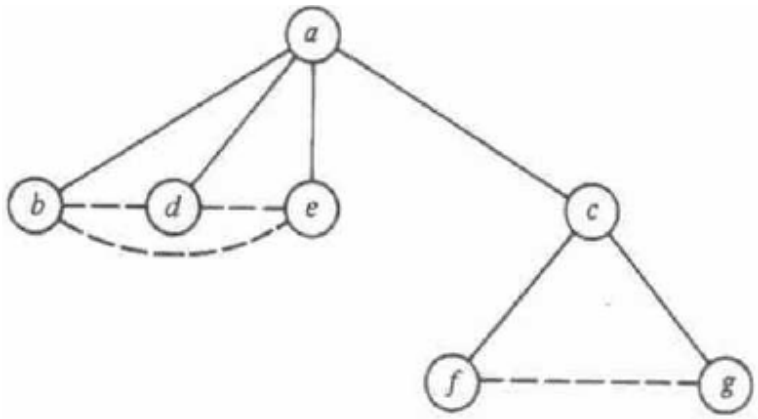


Fig 6.12. Búsqueda en amplitud para *G*.