Estructuras de Arbol

Nivel lógico

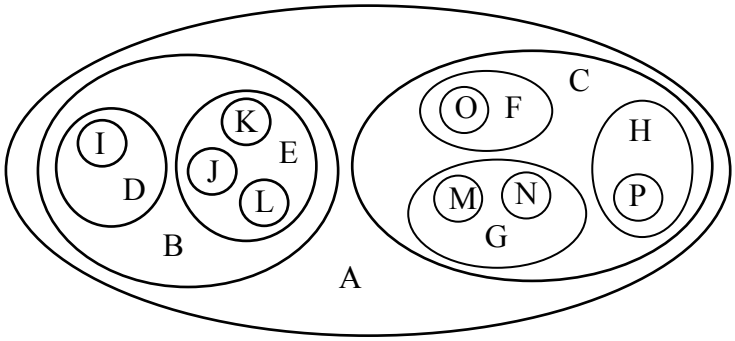
Una estructura de árbol puede definirse como sigue:

Una estructura de árbol con tipo base T es:

1. La estructura vacía.
2. Un nodo de tipo T con un número finito de estructuras árbol disjuntas asociadas de tipo base T, llamadas subárboles conectadas por ramas o aristas.

Hay varias formas de representar una estructura árbol. Por ejemplo, considerando que el tipo base son caracteres, tenemos:

- Conjuntos anidados



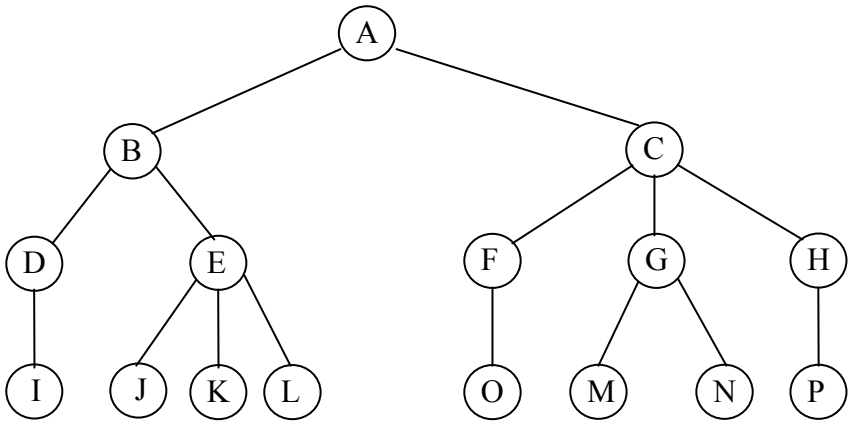
- Paréntesis anidados

  (A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))

- Indentación

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

- Grafo



La representación más usual es la de grafo.

Definiciones previas

Al nodo superior (en el ejemplo contiene la letra A) se le llama *raíz*.

Un nodo que está directamente debajo de otro (conectado por *aristas* o *ramas*) se dice que es un *descendiente* o *hijo* de ese nodo. Si un nodo no tiene descendientes directos o hijos se dice que es una *hoja* o un nodo terminal. Un nodo que no es terminal es un *nodo interior*.

Inversamente, un nodo es antecesor de otro si está en un nivel anterior a él. La raíz de un árbol por definición está en el nivel 1. El máximo nivel de un nodo del árbol caracteriza la *altura* del árbol. Otra definición de la altura de un árbol es la del número de aristas o ramas encontradas desde la raíz hasta el nodo hoja más distante desde éste. La *profundidad* de un nodo es el número de aristas del camino desde la raíz hasta el nodo en cuestión.

El *grado de un nodo* es el número de hijos o descendientes de un nodo interior. El máximo grado de todos los nodos es el *grado del árbol*. El máximo número de nodos en una árbol de un áltura h se alcanza si todos los nodos tienen tienen *d* subárboles, excepto aquellos en el nivel *h*, que no tienen ninguno.

Para un árbol de grado *d*, el máximo número de nodos en un árbol de altura *h* es:

$$N_d(h) = 1 + d + d^2 + \ldots.. + d^{h-1} = \sum_{i=0}^{h-1} d^i$$

Para d = 2, se obtiene: $N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$. La profundidad de un árbol binario con n nodos, por tanto, se calcula como $h = \log_2 n + 1$

Un *árbol ordenado* es aquel en el que las ramas de cada nodo están ordenados. Los árboles ordenados de grado 2 son lamados *árboles binarios*.

Un *árbol binario ordenado* se define como un conjunto finito, que puede estar vacío, de nodos que consisten de un nodo raíz con dos árboles binarios disjuntos llamados subárboles izquierdo y derecho de la raíz.

Implementación de árboles

La representación de un árbol en lenguaje C puede realizarse mediante punteros o arrays. La representación mediante arrays es ideal para árboles binarios completos ya que en éste caso el espacio desperdiciado es nulo. La forma más común es usar punteros. La estructura de datos usada corresponde a:

```
struct arbol_bin {
        int       data;
        struct   arbol_bin *izq;
        struct   arbol_bin *der;
};
struct arbol_bin *raiz = NULL;
```

Recorridos de árboles binarios

Después de construir un árbol es preciso procesar los valores de los datos almacenados en él. Esto quiere decir que es necesario moverse a través del árbol, visitando cada nodo exactamente una vez. La clasificación de éste tipo de algoritmos corresponde a un *recorrido*.

Si se está posicionado en un nodo cualquiera del árbol, una función de recorrido puede

- continuar por la rama izquierda, o
- continuar por la rama derecha, o
- procesar el dato (visitar el nodo).

Por convención (simplificación) se recorre el subárbol izquierdo antes que el derecho. El recorrido de un árbol viene caracterizado entonces por el orden de procesado del contenido del nodo en curso. Las formas posibles corresponden a las siguientes:

- Visitar el nodo antes de dirigirse al subárbol izquierdo.
- Visitar el nodo después de recorrer el subárbol izquierdo pero antes de recorrer el subárbol derecho.
- Visitar el nodo después de recorrer ambos subárboles.

Todos los métodos de recorrido son igualmente importantes, y se hará referencia a ellos por los nombres preorden, inorden y postorden, respectivamente.

Recorrido inorden

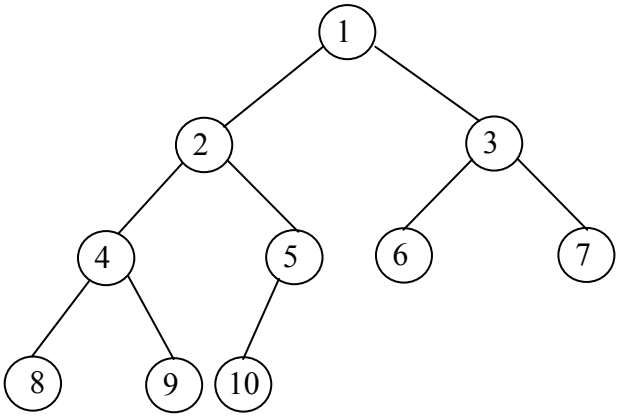
Este método de recorrido (también llamado simétrico) se puede describir informalmente como sigue:

- Moverse hacia el subárbol izquierdo hasta alcanzar la máxima profundidad.
- Visitar el nodo en curso.
- Volver hacia el nodo anterior en el árbol y visitarlo.
- Moverse hacia el subárbol derecho del nodo anteriormente visitado siempre que exista y no haya sido visitado previamente, de otra forma, volver hacia el nodo anterior.
- Repetir los pasos anteriores hasta que todos los nodos hayan sido procesados.

El siguiente procedimiento recursivo muestra una implementación del algoritmo descrito para imprimir el contenido del árbol cuya raíz se pasa como argumento:

```
/* in_orden: imprime el contenido del arbol con raiz p en in-orden */
void in_orden(struct arbol *p)
{
    if (p!=NULL) {
        in_orden(p->izq);
        printf("%4d   ",p->data);
        in_orden(p->der);
    }
}
```

La aplicación de ésta forma de recorrido al siguiente árbol produce el resultado mostrado a la izquierda del mismo



8 4 9 2 10 5 1 6 3 7

Recorrido preorden

En el recorrido en preorden se visita el nodo en curso antes de recorrer el subárbol izquierdo. A continuación se muestra el procedimiento para imprimir el contenido del árbol en preorden.

```
/* pre_orden: imprime el contenido del arbol con raiz p en pre-orden */
void pre_orden(struct arbol *p)
{
    if (p!=NULL) {
        printf("%4d   ",p->data);
        pre_orden(p->izq);
        pre_orden(p->der);
    }
}
```

La aplicación de ésta forma de recorrido al árbol anterior produce el siguiente resultado:

1 2 4 8 9 5 10 3 6 7

Recorrido postorden

El recorrido postorden visita el nodo después de recorrer los subárboles izquierdo y derecho respectivamente. El procedimiento que implementa este tipo de recorrido es el siguiente:

```
/* post_orden: imprime el contenido del arbol con raiz p en post-orden */
void post_orden(struct arbol *p)
{
    if (p!=NULL) {
        post _orden(p->izq);
        post _orden(p->der);
        printf("%4d   ",p->data);
    }
}
```

La aplicación de ésta tipo de recorrido al árbol que sirve de ejemplo produce el siguiente resultado:

8 9 4 10 5 2 6 7 3 1

Recorrido a lo ancho

Los tres métodos de recorrido que se han visto antes son similares en tanto que todos procesan completamente los descendientes de un nodo antes de procesar la información de los nodos que se encuentran en el mismo nivel. Por ello, los métodos vistos se clasifican como búsqueda primero en profundidad. Otra clase de recorrido de un árbol es una búsqueda primero a lo ancho. En una búsqueda primero a lo ancho se procesan los nodos por niveles, de izquierda a derecha en cada nivel.

La implementación de una función que realiza tal recorrido se muestra a continuación:

```
/* imprime_arbol_pa: imprime el contenido del arbol
   primero a lo ancho                                    */
void imprime_arbol_pa(struct arbol *p)
{
    struct arbol *t;
    int i;

    inserta(p);
    while ((t=borra())!=NULL)
    {
        printf("%4d  ",t->item);
        if (t->izq!=NULL)
            inserta(t->izq);
        if (t->der!=NULL)
            inserta(t->der);
    }
}
```

Esta función empieza colocando el nodo raíz en una cola. En cada iteración del ciclo while la función extrae el siguiente nodo de la cola, lo procesa, y a continuación colca en la cola sus hijos (si tiene alguno). El proceso termina cuando la cola se encuentra vacía. La aplicación de esta función al ejemplo usado produce el siguiente resultado:

1 2 3 4 5 6 7 8 9 10

Arbol perfectamente balanceado

Un árbol es perfectamente balanceado si para cada nodo se cumple que el número de nodos en sus subárboles derecho e izquierdo difiere como mucho en uno.

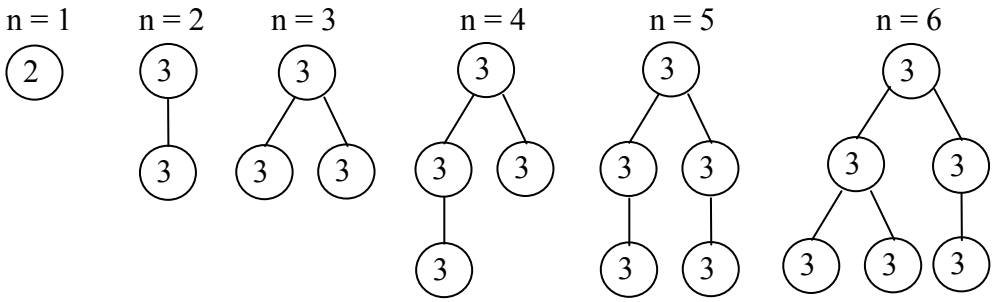
Para construir un árbol perfectamente balanceado o de altura mínima se puede usar el siguiente algoritmo (recursivo) conociendo el número de nodos *n*:

Usar un nodo para la raíz.

Generar el subárbol izquierdo con $nl = n$ div 2 nodos.

Generar el subárbol derecho con $nr = n - nl - 1$ nodos.

Los gráficos de este tipo de árboles para nodos en el intervalo [1, 6] se muestra a continuación:



La implementación del algoritmo anterior en un programa que crea e imprime un árbol balanceado a partir de un fichero de datos sigue a continuación:

```
#include <stdio.h>
#define FICHERO_DATOS "arb_bal.dat"

struct arbol {
    int item;
    struct arbol *izq;
    struct arbol *der;
};

struct arbol *arbol_bal(int n);
void imprime_arbol(struct arbol *p,int tab);

struct arbol *raiz;
FILE *fp;

main()
{
    int n;

    fp=fopen(FICHERO_DATOS,"r");
    if (fp==NULL)
    {
        printf("Error al intentar abrir el fichero %s\n",FICHERO_DATOS);
        exit(-1);
    }
    fscanf(fp,"%d",&n);
    raiz=arbol_bal(n);
    imprime_arbol(raiz,0);
    return(0);
}
```

```
/* Funcion que retorna un arbol balanceado
   cuyos datos se leen desde un fichero   */
struct arbol *arbol_bal(int n)
{
    struct arbol *nuevo_nodo;
    int dato, ni, nd;

    if (n==0)
        return(NULL);
    else
    {
        ni=n/2;
        nd=n-ni-1;
        fscanf(fp,"%d",&dato);
        nuevo_nodo=(struct arbol *) malloc(sizeof(struct arbol));
        nuevo_nodo->item=dato;
        nuevo_nodo->izq=arbol_bal(ni);
        nuevo_nodo->der=arbol_bal(nd);
        return(nuevo_nodo);
    }
}

/* imprime_arbol: imprime p en in-orden con una tabulacion
   correspondiente al nivel de cada nodo                */
void imprime_arbol(struct arbol *p,int tab)
{
    int i;

    if (p!=NULL) {
        imprime_arbol(p->izq,tab+2);
        for(i=0;i<tab;i++)
            printf(" ");
        printf("%4d\n",p->item);
        imprime_arbol(p->der,tab+2);
    }
}
```

Arbol binario de búsqueda

Un árbol binario de búsqueda es un árbol en el que el hijo de la izquierda, si existe, de cualquier nodo contiene un valor más pequeño que el nodo padre, y el hijo de la derecha, si existe, contiene un valor más grande que el nodo padre.

A continuación se da la especificación de esta estructura:

Estructura árbol binario de búsqueda

Función de acceso: El emplazamiento de cada elemento en el árbol satisface la siguiente regla: El valor de la clave de un elemento es mayor que el valor de la clave de

cualquier elemento en su subárbol izquierdo, y menor que el valor de la clave de cualquier elemento en su subárbol derecho.

Operaciones:

Buscar_Arbol(ValorClave)

Función:	Busca en un árbol binario de búsqueda un nodo cuya clave sea ValorClave, retornando un puntero al nodo buscado. Si no se encuentra el valor en ningún nodo del árbol devuelve NULL.
Entrada:	ValorClave
Precondiciones:.	Se tiene un puntero al nodo raíz del árbol binario de búsqueda.
Salida:	Devuelve un puntero al nodo que contiene el ValorClave.
Postcondiciones:	Buscar_Arbol contiene un puntero al nodo buscado sino NULL.

Insertar(InfoNodo)

Función:	Construye un nodo conteniendo InfoNodo y lo inserta en el árbol binario de búsqueda.
Entrada:	InfoNodo
Precondiciones:.	Se tiene un puntero al nodo raíz del árbol binario de búsqueda.
Salida:	Ninguna.
Postcondiciones:	El árbol contiene un nodo adicional con InfoNodo, ubicado de acuerdo con su valor clave y la regla de la función de acceso.

Suprimir(ValorClave)

Función:	Suprime nodo conteniendo Valorclave del árbol binario de búsqueda.
Entrada:	ValorClave
Precondiciones:.	Se tiene un puntero al nodo raíz del árbol binario de búsqueda.
Salida:	Ninguna.
Postcondiciones:	El árbol que contenía ValorClave no está en el árbol conservándose la regla de la función de acceso.

ImprimeArbol(OrdenRecorrido)

Función:	Imprime todos los elementos del árbol binario de búsqueda en el orden indicado por OrdenRecorrido.
Entrada:	OrdenRecorrido (preorden, inorden, postorden).
Precondiciones:.	Se tiene un puntero al nodo raíz del árbol binario de búsqueda.
Salida:	Lista todos los elementos del árbol (pantalla).
Postcondiciones:	El árbol no cambia.

Implementación del árbol binario de búsqueda

Normalmente los programas que emplean los árboles binarios constan de dos fases: en la primera se construye el árbol y en la segunda se recorre para procesar la información de los nodos. Antes se han visto diversos métodos para recorrer árboles. A continuación se muestran los algoritmos para insertar, buscar y borrar nodos.

Inserción

En general hay dos lugares donde se pueden insertar nuevos nodos en una árbol binario: en los nodos terminales (hojas) y los nodos internos. En un árbol binario de búsqueda se restringe la inserción de nuevos nodos a los nodos terminales debido a la condición de ordenamiento entre los nodos.

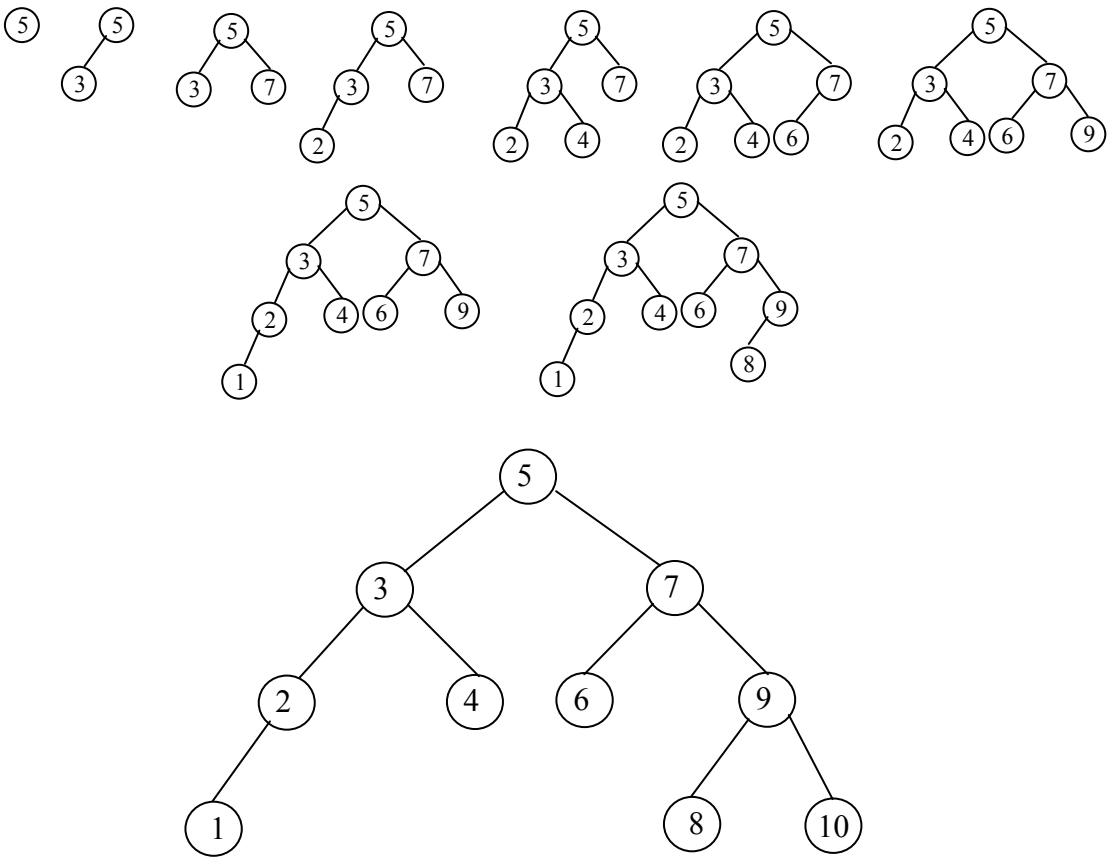
Para ello es necesario recorrer el árbol comparando el nuevo valor con los nodos existentes. La búsqueda se dirige por los descendientes, ya sea por la izquierda o derecha, según el valor añadir es menor o mayor respectivamente que el nodo en curso. Cuando se llega a un nodo terminal se inserta el nuevo valor como un descendiente de este nodo. A continuación se muestra una implementación que usa punteros

```
#include <stdio.h>
#include "arb_bin.h"

struct arb_bin *raiz=NULL;

/* insertar: anade un nodo en el arbol binario */
void inserta(int info)
{
    struct arb_bin *crea_nodo(void);

    struct arb_bin *p, *q;

    if (raiz==NULL) {
        raiz = crea_nodo();
        raiz->info = info;
        return;
    }
    p=raiz;
    while (p!=NULL) {
        q=p;
        if (info<p->info)
            p=p->izq;
        else
            p=p->der;
    }
    p = crea_nodo();
    p->info = info;
    if (info<q->info)
        q->izq=p;
    else
        q->der=p;
    return;
}

/* crea_nodo: construye un nodo del arbol */
struct arb_bin *crea_nodo(void)
{
    struct arb_bin *nuevo_nodo;
```

```
    nuevo_nodo = (struct arb_bin *) malloc(sizeof(struct arb_bin));
    if (nuevo_nodo == NULL) return NULL;
    nuevo_nodo->izq = NULL;
    nuevo_nodo->der = NULL;
    return(nuevo_nodo);
}
```

La función hace uso de una función auxiliar (crea_nodo) que devuelve una posición de memoria para el nuevo nodo que se inserta. Un ejemplo de construcción de un árbol binario de búsqueda para una secuencia de números enteros se muestra a continuación:

5, 3, 7, 2, 4, 6, 9, 1, 8, 10



Búsqueda

Con esta operación se consigue un índice o puntero al elemento buscado dentro del árbol según su clave o NULL si no se encuentra. Una implementación se esta función sigue a continuación:

```
/* buscar: busca un nodo en el arbol binario segun
   el valor de info                          */
struct arb_bin *buscar(int info)
```

```
{
    struct arb_bin *p, *q;

    p=raiz;
    while (p!=NULL) {
        if (p->info==info)
            return (p);
        else
            if (p->info>info)
                p=p->izq;
            else
                p=p->der;
    }
    return(p);
}
```

Borrado

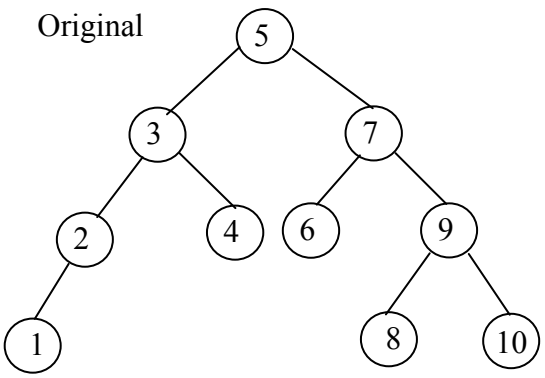
Aun cuando la mayoría de aplicaciones que usan árboles binarios no requieren funciones de borrado, se incluye tal función con objeto de apreciar su manipulación. El borrado de un nodo pasa por analizar su tipo dependiendo de su ubicación dentro del árbol, existiendo tres casos que se describen a continuación:

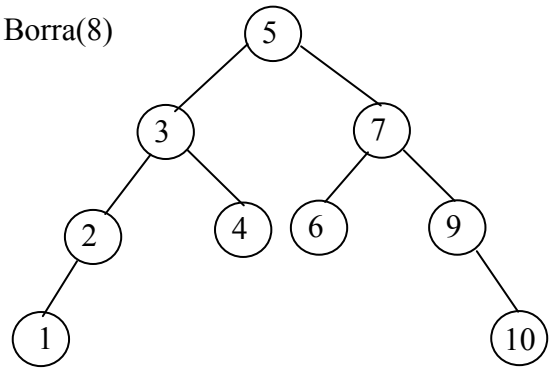
Si un nodo es terminal o no tiene hijos simplemente se elimina.

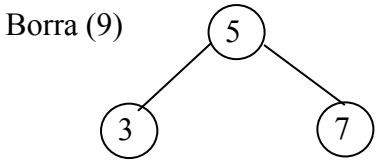
Si el nodo a eliminar tiene un sólo hijo (izquierdo o derecho) el nodo padre de éste nodo se convertirá en antecesor del nodo hijo en una operación parecida a la eliminación de un nodo intermedio en una lista.

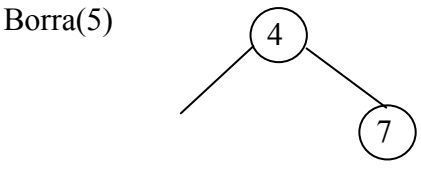
Cuando el nodo tiene dos hijos se puede intercambiar la información contenida en el nodo más a la derecha del subárbol izquierdo o el nodo más a la izquierda del subárbol derecho, con el propósito de conservar el ordenamiento. A continuación, se elimina éste nodo según los métodos anteriores.

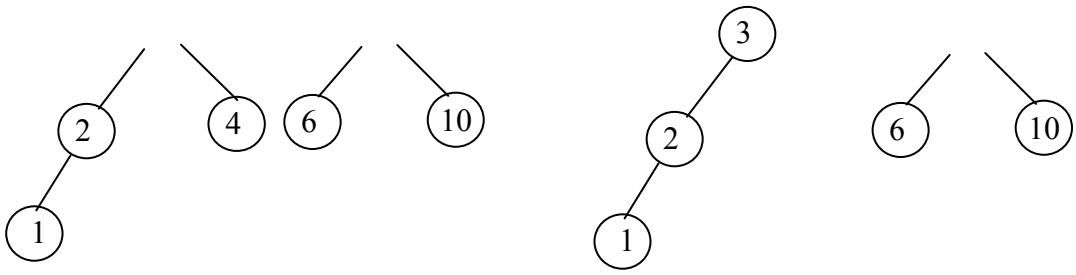
Ejemplos de borrados de nodos con el resultado según las reglas anteriores sigue a continuación:











Una implementación de la función que realiza esta operación sigue a continuación:

```
/* borra: suprime un nodo en el arbol binario segun
   el valor de info                              */
void borra(int info)
{
 void suprime_nodo(struct arb_bin *a,struct arb_bin *p);
 struct arb_bin *p, *ant;

 p=raiz;
 ant=NULL;
 while (p->info!=info){
   ant=p;
   if (p->info>info)
      p=p->izq;
   else
      p=p->der;
 }
if (p==raiz)
   suprime_nodo(raiz,raiz);
 else
   suprime_nodo(ant,p);
}

#define IZQUIERDO 1
#define DERECHO   2
void suprime_nodo(struct arb_bin *prev,struct arb_bin *p)
{
 struct arb_bin *ant, *t, *e;
 int hijo;

 t=p;
 hijo=(prev->izq==p)?IZQUIERDO:DERECHO;
 if (p->der==NULL)
   if (hijo == IZQUIERDO)
      prev->izq=p->izq;
   else
      prev->der=p->izq;
 if (p->izq==NULL)
   if (hijo == IZQUIERDO)
      prev->izq=p->der;
```

```
      else
        prev->der=p->der;
    else
    {
      t=p->izq;
      ant=p;
      while (t->der!=NULL)
      {
       ant=t;
       t=t->der;
      }
      p->info=t->info;
      if (ant==p)
        ant->izq=t->izq;
      else
        ant->der=t->izq;
    }
    free(t);
  }
```

Impresión

La impresión de un árbol binario de búsqueda hace uso de los métodos de recorrido vistos anteriormente. Una función que acepta como argumento el método de recorrido sigue a continuación:

```
  /* imprime_arbol: imprime arbol con raiz p
     en el orden indicado por orden_recorrido */
  void imprime_arbol(int orden_recorrido)
  {
   void imprime_preorden(struct arb_bin *p);
   void imprime_inorden(struct arb_bin *p, int tab);
   void imprime_postorden(struct arb_bin *p);
   switch (orden_recorrido)
   {
    case PRE_ORDEN:
            imprime_preorden(raiz);
            break;
    case IN_ORDEN:
            imprime_inord(raiz);
            break;
    case POST_ORDEN:
            imprime_postorden(raiz);
            break;
   }
  }
```

Ejemplo de Aplicación

Se trata de desarrollar un programa que permitar contar las ocurrencias de todas las palabras de un texto. Como se desconoce la lista de palabras que contiene el texto no puede pensarse en ordenarlas y usar búsqueda binaria. Otra alternativa es usar una búsqueda lineal según se lee cada palabra, pero esto tomaría demasiado tiempo (crece cuadráticamente) cuando el número de palabras aumenta. La mejor solución (por el momento) es tener el conjunto de palabras leídas en cualquier momento ordenadas colocando cada palabra nueva en su lugar. Para ello usaremos un árbol binario de búsqueda.

El árbol contendrá en cada nodo la siguiente información:

Un puntero al texto de la palabra,

Un contador del número de ocurrencias,

Punteros a los hijos izquierdo y derecho respectivamente.

A continuación se muestra el código que implementa el programa solicitado:

```
#include <stdio.h>
#define MAX_CAR 100

struct arb_bin {
  char *palabra;
  int cont;
  struct arb_bin *izq;
  struct arb_bin *der;
};

struct arb_bin *inserta(struct arb_bin *, char *);
void imprime(struct arb_bin *);
int leepalabra(char *, int);

/* Programa que cuenta la frecuencia de palabras de un texto */
main()
{
 struct arb_bin *raiz;
 char palabra[MAX_CAR];

 raiz=NULL;
 while(leepalabra(palabra,MAX_CAR)!=EOF)
   if (isalpha(palabra[0]))
      raiz=inserta(raiz,palabra);
 imprime(raiz);
 return 0;
}

struct arb_bin *talloc(void);
char *strdup(char *);

/* inserta: anade un nodo con w, en o bajo p */
```

```
struct arb_bin *inserta(struct arb_bin *p, char *w)
{
 int cond;

 if (p==NULL) {
    p = talloc();
    p->palabra = strdup(w);
    p->cont = 1;
    p->izq = p->der = NULL;
 } else if ((cond=strcmp(w,p->palabra))==0)
    p->cont++;
 else if (cond<0)
    p->izq= inserta(p->izq,w);
 else
    p->der = inserta(p->der,w);
 return p;
}

/* imprime: imprime p en in-orden */
void imprime(struct arb_bin *p)
{
 if (p!=NULL) {
   imprime(p->izq);
   printf("%4d  %s\n",p->cont,p->palabra);
   imprime(p->der);
 }
}

/* talloc: construye un nodo del arbol binario */
struct arb_bin *talloc(void)
{
 return (struct arb_bin *) malloc(sizeof(struct arb_bin));
}

/* devuelve un puntero a una copia de una cadena de caracteres */
char *strdup(char *s)
{
 char *p;

 p=(char *) malloc(strlen(s)+1);
 if (p!=NULL)
   strcpy(p,s);
 return p;
}

/* leepalabra: lee sigte. palabra o caracter de la entrada */
int leepalabra(char *palabra, int lim)
{
 int c, leech(void);
```

```
        void devuelvech(int);
        char *w = palabra;

        while (isspace(c=leech()))
            ;
        if (c!=EOF)
           *w++=c;
        if(!isalpha(c)) {
           *w='\0';
           return c;
        }
        for ( ;--lim>0; w++)
          if(!isalnum(*w=leech())) {
             devuelvech(*w);
             break;
          }
        *w='\0';
        return palabra[0];
       }

       #define DIMBUFFER 100
       char buffer[DIMBUFFER];
       int bufp=0;

       int leech(void)
       {
        return (bufp>0)?buffer[--bufp]:getchar();
       }

       void devuelvech(int c)
       {
        if (bufp>DIMBUFFER)
           printf("devuelvechar: demasiados caracteres\n");
        else
           buffer[bufp++] = c;
       }
```

La ejecución de este programa sobre el fichero fuente del mismo produce el siguiente resultado: (el comando usado fue: cuenta_p.exe < cuenta_p.c > salida)

```
 3  CAR          14  bin          3  cond          1  devuelve
 3  DIMBUFFER     1  binario      1  construye     3  devuelvech
 2  EOF           1  break        4  cont          1  devuelvechar
 3  MAX           3  buffer       1  copia         4  else
 5  NULL          5  bufp         1  cuenta        2  en
 1  Programa      8  c            1  d             1  entrada
 1  a             1  cadena       5  de            1  for
 1  anade         1  caracter      2  define        1  frecuencia
14  arb           2  caracteres    1  del           1  getchar
 1  arbol        14  char         1  demasiados    1  h
 1  bajo          1  con          5  der          10  if
```

7 imprime	4 leech	1 palabras	1 strcpy
1 in	4 leepalabra	2 printf	3 strdup
1 include	2 lim	1 puntero	1 strlen
6 inserta	1 main	1 que	14 struct
11 int	2 malloc	5 raiz	4 talloc
1 isalnum	2 n	7 return	1 texto
2 isalpha	2 nodo	4 s	4 un
1 isspace	2 o	1 sigte	2 una
5 izq	1 orden	1 sizeof	8 void
2 la	27 p	1 stdio	13 w
1 lee	12 palabra	1 strcmp	2 while