Estructura de Datos y Algoritmos

Arboles

1. Introducción

Los árboles imponen una estructura jerarquica en una colección de elementos. Ejemplos de árboles son los árboles genealogicos y los diagramas de organizaciones. Es usual utilizar árboles para organizar información en sistemas de bases de datos.

1.1. Terminología básica

Un árbol es una colección de elementos llamados nodos, uno de los cuales es distinguido y llamado **raiz** junto con una relación *ser padre* que impone una estructura jerárquica en los nodos.

Un nodo, del mismo modo que un elemento en una lista, puede ser del tipo que deseemos. Puede ser un caracter, un string o un número. Formalmente, un árbol se puede definir recursivamente de la siguiente forma:

- Un nodo es en si mismo un árbol. Es el nodo raiz.
- Supongamos n es un nodo y $T_1, T_2, \ldots, T_k$ son árboles con raices $n_1, n_2, \ldots, n_k$ respectivamente. Podemos construir un nuevo árbol haciendo que n sea padre de los nodos $n_1, n_2, \ldots, n_k$. En ese árbol, n es la raiz y $T_1, T_2, \ldots, T_k$ son los subárboles de la raiz. Los nodos $n_1, n_2, \ldots, n_k$ son llamados hijos de la raiz.

Algunas veces puede ser conveniente incluir entre los árboles el árbol vacio, el cual no tiene nodos, que representamos con NULL.

Ejemplo:

Consideremos la tabla de contenidos de un libro. Ésta es un árbol. La raiz, llamada "libro"tiene subárboles correspondientes a los capítulos. Los capítulos tienen subárboles correspondientes a las secciones y éstas subárboles correspondientes a las subsecciones.

Definición: Cáminos, descendientes, ancestros y subárboles.

Si $v_1, v_2, \ldots, v_k$ es una secuencia de nodos en un árbol tal que v_i es padre de v_{i+1} para $1 \leq i < k$, llamamos a ésta secuencia camino de v_1 a v_k. El largo del camino es uno menos que la cantidad de vértices. Observar que un nodo es en si mismo un camino de largo 0.

Si hay camino del nodo a al nodo b, decimos a es ancestro de b y b es descendiente de a. En el ejemplo del libro los capítulos son ancestros de las secciones

y subsecciones y estos últimos son descendientes de los capítulos. Observar que un nodo es ancestro y descendiente de si mismo.

Un ancestro o descendiente de un nodo, distinto de el mismo es llamado ascentro propio o descendiente propio. En un árbol, la raiz es el único nodo sin ancestros propios. Un nodo sin descendiente propio se llama **hoja**. Un subárbol de un árbol es un nodo junto con sus descendientes.

Definición: Altura, profundidad.

La altura de un nodo es el largo del camino más largo del nodo a una hoja. La altura de un árbol es la altura de la raiz. La profundidad de un nodo es el largo del camino unico de la raiz a dicho nodo.

1.2. El orden de los nodos

Los hijos de un nodo se ordenan usualmente de izquierda a derecha. Dos árboles en los cuales los hijos de un nodo aparecen en un orden distinto que en el otro son distintos.

Si queremos ignorar el orden de los hijos hablamos de un **árbol no ordenado**.

El ordenamiento de izquierda a derecha de los hermanos (hijos de un mismo nodo) se puede extender para comparar cualesquiera dos nodos que no estén en la relación ancestro-descendiente. La regla es que 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.

Una regla simple, dado un nodo n distinto de la raiz, para encontrar aquellos nodos en su izquierda y en su derecha es dibujar el camino de la raiz a n. Todos los nodos a la izquierda de ese camino y los descendientes de estos nodos están a la izquierda de n. Todos los nodos y descendientes a la derecha del camino están a la derecha de n.

1.3. Preorden, Postorden y Inorden

Hay distintas formas en las cuales podemos ordenar todos los nodos de un árbol. Los ordenamientos más importantes son preorden, postorden e inorden. Estos se definen recursivamente como sigue:

- El ordenamiento de los nodos da una lista de nodos.
- Si el árbol es vacio, cualquiera de los tres ordenamientos son la lista vacia.
- Si T es un nodo simple este nodo en si mismo es la lista (en cualquiera de los tres ordenamientos).

En otro caso, sea T un árbol con raiz n y subárboles $T_1, T_2, \ldots, T_k$

1. El listado en preorden de los nodos de T son la raiz n seguido por el listado en preorden de T_1, el listado en preorden de T_2 y asi sucesivamente hasta el listado en preorden de T_k.

2. El listado en inorden de los nodos de T consiste en el listado en inorden de los nodos de T_1 seguido de n, seguido del listado en inorden de los nodos de T_2, y asi sucesivamente hasta el listado en inorden de los nodos de T_k.

3. El listado en postorden de los nodos de T consiste en el listado de los nodos de T_1 en postorden, el listado de los nodos de T_2 en postorden y asi sucesivamente hasta el listado de T_k en postorden y luego n.

2. Árboles binarios de Búsqueda

Un árbol binario es un árbol vacio o un árbol en el cual cada nodo tiene ninguno, uno o dos hijos. Llamamos a los hijos hijo izquierdo e hijo derecho.

Una aplicación importante de árboles binarios es su uso en búsquedas. Asumamos a cada nodo del árbol se asocia un valor (asumimos un entero, podriamos utilizar valores más complejos). Supondremos no hay valores duplicados.

La propiedad que hace de un árbol binario un árbol binario de búsqueda es que para todo nodo en el árbol los valores de los nodos en el subárbol izquierdo son menores y los valores de los nodos en el subárbol derecho son mayores.

Veamos que operaciones realizamos en árboles binarios de búsqueda. Es usual escribir dichas operaciones recursivamente.

2.1. Operaciones

Tenemos las siguientes operaciones:

1. Creo_vacio : retorna el árbol vacio (NULL).
2. Es_vacio(T) : retorna true si T es NULL, false en otro caso.
3. Valor(T) : retorna el valor de la raiz de T.
4. Hijo_izq(T) : retorna la dirección del subarbol izquierdo de T. Precondición : T no vacio.
5. Hijo_der(T) : retorna la dirección del subarbol derecho de T. Precondición : T no vacio.
6. Buscar(T,x) : retorna la dirección del nodo con valor x.
7. Buscar_minimo(T) : retorna la dirección del nodo con menor valor en el árbol.
8. Buscar_maximo(T) : retorna la dirección del nodo con mayor valor en el árbol.
9. Insertar(T,x) : inserta un nodo con valor x en el árbol T. La inserción respeta la propiedad de que el árbol es de búsqueda.
10. Borrar(T,x) : borra el nodo con valor x del árbol T.
11. Inorden(T) : lista los valores de los nodos de T cuando recorremos el árbol en inorden.

12. Preorden(T) : lista los valores de los nodos de T cuando recorremos en árbol en preorden.

13. Postorden(T) : lista los valores de los nodos de T cuando recorremos el árbol en postorden.

A continuación implementaremos las operaciones para las representaciones con arreglo y con punteros.

2.2. Implementación con arreglos

```
#define maxnodos 20

struct nodo{
        int padre;
        int izquierdo;
        int derecho;
        int valor;
        bool borrado;
        };

struct arbol{
        nodo abb[maxnodos];
        int tope;
        int nro_borrado;
        };

// tope indica el indice del ultimo nodo creado.

// funcion Creo_vacio: crea un arbol sin nodos que lo indicamos con tope=-1.
arbol Creo_vacio()
{       arbol T;
        T.tope=-1;
        T.nro_borrado=0;
        return T;
}
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Es_vacio: devuelve true si el arbol no tiene nodos.
bool Es_vacio(arbol T)
{      return (T.tope==-1); }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Valor: dado el indice de un nodo devuelve su valor.
int Valor(arbol T,int i)
{    return T.abb[i].valor; }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Es_hoja: es true si tanto el arbol izquierdo como el derecho
// son vacios.
bool Es_hoja(arbol T,int i)
{     return (T.abb[i].izquierdo==-1) && (T.abb[i].derecho==-1); }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Padre: dado el indice de un nodo, devuelve el indice de su padre.
// Precondicion: el nodo existe.
int Padre(arbol T, int i)
{ return T.abb[i].padre; }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Hijo_izq: dado el indice de un nodo, devuelve el indice de su
// subarbol izquierdo.
// Precondicion: el nodo existe.
int Hijo_izq(arbol T,int i)
{ return T.abb[i].izquierdo; }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Hijo_der: dado el indice de un nodo, devuelve el indice de su
// subarbol derecho.
// Precondicion: el nodo existe.
int Hijo_der(arbol T,int i)
{ return T.abb[i].derecho; }
```

Es $O(1)$ y $\Omega(1)$.

```
// funcion Buscar: busco el nodo recursivamente en el subarbol izquierdo
// o derecho dependiendo de su valor. Se pasan como argumento el valor que
// busco y el indice del nodo que estoy considerando (contra el que estoy
// comparando). Se retorna el indice del nodo con el valor o -1 si el valor
// no existe en el arbol.
int Buscar(arbol T,int x,int i)
{   if (i==-1) return -1;
    else if (Valor(T,i)==x) return i;
         else if (Valor(T,i)>x) return Buscar(T,x,Hijo_izq(T,i));
              else return Buscar(T,x,Hijo_der(T,i));
}
```

En el peor caso se recorre el camino de la raiz a una hoja.
Supongamos todos los largos son iguales (misma cantidad de nodos
en el subarbol izquierdo que en el derecho para todo nodo).
Llamemos n a la cantidad de nodos. Consideramos la cantidad de nodos que recorremos.

$T(n) = T(n/2) + 1$
$T(n/2) = T(n/4) + 1$
$\vdots$
$T(2) = T(n/2^k) + 1$
$T(1) = 1$

sumando obtenemos $T(n) = k + 1$ donde $k + 1$ es la cantidad de ecuaciones.
Entonces $2^k = n$ luego $k = log_2(n)$.
De donde la operacion es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Buscar_minimo: se retorna el valor del nodo que esta mas a la
// izquierda.
int Buscar_minimo(arbol T,int i)
{   int j;
    if (Es_vacio(T)) return -1;
    if (Hijo_izq(T,i)==-1) return Valor(T,i);
    else return Buscar_minimo(T,Hijo_izq(T,i));
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Buscar_maximo: se retorna el valor del nodo que esta mas a la
// derecha.
int Buscar_maximo(arbol T,int i)
{   if (Es_vacio(T)) return -1;
    if (Hijo_der(T,i)==-1) return Valor(T,i);
    else return Buscar_maximo(T,Hijo_der(T,i));
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$

```
// funcion ins : funcion auxiliar utilizada por Insertar.
// Se invoca cuando el arbol en el cual voy a insertar es no vacio.
// Si el valor a insertar es menor que la raiz, y no hay subarbol
// izquierdo lo inserto ahi, si hay subarbol izquierdo, invoco a la
// funcion recursivamente. Si el valor a insertar es mayor que la raiz,
// y no hay subarbol derecho lo inserto ahi, si hay subarbol derecho
// invoco a la funcion recursivamente. El caso restante es que el valor
// sea igual al valor de la raiz en cuyo caso se cancela la insercion.
void ins(arbol &T,int x,int i)
{
      if (x < Valor(T,i))
        if (Hijo_izq(T,i)==-1) { T.tope++;
```

```
                                    T.abb[i].izquierdo=T.tope;
                                    T.abb[T.tope].padre=i;
                                    T.abb[T.tope].valor=x;
                                    T.abb[T.tope].borrado=false;
                                    T.abb[T.tope].izquierdo=-1;
                                    T.abb[T.tope].derecho=-1;}
        else ins(T,x,Hijo_izq(T,i));
    else if (x > Valor(T,i))
          if (Hijo_der(T,i)==-1) { T.tope++;
                                    T.abb[i].derecho=T.tope;
                                    T.abb[T.tope].padre=i;
                                    T.abb[T.tope].valor=x;
                                    T.abb[T.tope].borrado=false;
                                    T.abb[T.tope].izquierdo=-1;
                                    T.abb[T.tope].derecho=-1;}
          else ins(T,x,Hijo_der(T,i));
        else cout << "Valor ya existente"<< endl; }
```

es $O(log_2(n))$ y $\Omega(log_2(n))$

```
// funcion Insertar: si el arbol es vacio crea un nuevo nodo con la
// informacion (nodo raiz). Sino llama a la funcion ins, que realiza
// la insercion.
    void Insertar(arbol &T,int x)
{
    if (Es_vacio(T))
      { T.abb[0].valor=x;
        T.abb[0].izquierdo=-1;
        T.abb[0].derecho=-1;
        T.abb[0].padre=-1;
        T.abb[0].borrado=false;
        T.tope=0;
        return;
      }
    else ins(T,x,0);
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$

```
// funcion Borrar: Si el arbol es vacio no se modifica. Sino busco el
// nodo a borrar en el subarbol izquierdo o derecho dependiendo de su
// valor. Cuando lo encuentro, si no tiene padre (es la raiz) y es hoja,
// borro la raiz o sea retorno el arbol vacio. Si no es la raiz tiene padre.
// Supongamos es hoja y tiene padre. Si su valor es menor que el valor del
// padre es un hijo izquierdo, borro el hijo izquierdo del padre. Si su valor
// es mayor que el valor del padre es un hijo derecho, borro el hijo derecho
// del padre. Si no es hoja y no tiene hijo izquierdo y su valor es mayor que
// el de su padre coloco su hijo derecho como hijo derecho del padre. Modifico
```

```
// tambien el padre del hijo derecho. Si no es hoja y no tiene hijo izquierdo
// y su valor es menor que el de su padre coloco su hijo derecho como hijo
// izquierdo del padre. Modifico tambien el padre del hijo derecho.
// De igual forma si no es hoja y no tiene hijo derecho. Si no es hoja y tiene
// los dos hijos, coloco el minimo del subarbol derecho en el lugar del nodo
// que quiero borrar y borro este ultimo.
void Borrar(arbol &T,int x,int i)
{     int a,j,k;
      if (i==-1) return;
      if (x < Valor(T,i)) Borrar(T,x,Hijo_izq(T,i));
      else if (x > Valor(T,i)) Borrar(T,x,Hijo_der(T,i));
           else
                { j=i;
                  if (Es_hoja(T,j))
                     { if (Padre(T,j)==-1) { T.abb[0].borrado=true;
                                             T.nro_borrado++;
                                             T.tope=-1;
                                             return;}
                       if (Valor(T,j) < Valor(T,Padre(T,j)))
                          { T.abb[Padre(T,j)].izquierdo=-1;
                          }
                       else if (Valor(T,j) > Valor(T,Padre(T,j)))
                               { T.abb[Padre(T,j)].derecho=-1;
                               }
                       T.abb[j].borrado=true;
                       T.nro_borrado++;
                       return;
                     }
                  else
                      if (Hijo_izq(T,j)==-1)
                        { if (Valor(T,j) > Valor(T,Padre(T,j)))
                             {k=T.abb[j].derecho;
                             T.abb[Padre(T,j)].derecho=T.abb[j].derecho;
                             T.abb[k].padre=Padre(T,j);}
                          else
                             {k=T.abb[j].derecho;
                             T.abb[Padre(T,j)].izquierdo=T.abb[j].derecho;
                             T.abb[k].padre=Padre(T,j);}
                          T.abb[j].borrado=true;
                          T.nro_borrado++;}
                      else if (Hijo_der(T,j)==-1)
                             { if (Valor(T,j) > Valor(T,Padre(T,j)))
                                  {k=T.abb[j].izquierdo;
                                  T.abb[Padre(T,j)].derecho=T.abb[j].izquierdo;
                                  T.abb[k].padre=Padre(T,j);}
                               else
                                  {k=T.abb[j].izquierdo;
                                  T.abb[Padre(T,j)].izquierdo=T.abb[j].izquierdo;
                                  T.abb[k].padre=Padre(T,j);}
                               T.abb[j].borrado=true;
```

```
                                T.nro_borrado++;}
                            else
                                {a=Buscar_minimo(T,Hijo_der(T,j));
                                Borrar(T,a,j);
                                T.abb[j].valor=a;
                                }
                    }
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$

```
// funcion Reorganizar: crea un arbol con los nodos no borrados.
arbol Reorganizar(arbol T)
{       arbol p=Creo_vacio();
        for (int i=0;i¡=T.tope;i++)
            if (!T.abb[i].borrado) Insertar(p,T.abb[i].valor);
        return p;
}
```

es $O(n)$ y $\Omega(n)$.

```
// funcion Inorden: lista los nodos del arbol en recorrida inorden.
// Para imprimir todos los nodos del arbol se invoca con i=0.
void Inorden(arbol T,int i)
{       if (Es_vacio(T)) return;
        if (i==-1) return;
        Inorden(T,Hijo_izq(T,i));
        cout << Valor(T,i) << endl;
        Inorden(T,Hijo_der(T,i));
}
```

es $O(n)$ y $\Omega(n)$.

```
// funcion Preorden: lista los nodos del arbol en recorrida preorden.
void Preorden(arbol T,int i)
{       if (Es_vacio(T)) return;
        if (i==-1) return;
        cout << Valor(T,i) << endl;
        Preorden(T,Hijo_izq(T,i));
        Preorden(T,Hijo_der(T,i));
}
```

es $O(n)$ y $\Omega(n)$.

```
// funcion Postorden: lista los nodos del arbol en recorrida postorden.
void Postorden(arbol T,int i)
{       if (Es_vacio(T)) return ;
        if (i==-1) return;
        Postorden(T,Hijo_izq(T,i));
        Postorden(T,Hijo_der(T,i));
        cout << Valor(T,i) << endl;
}
```

es $O(n)$ y $\Omega(n)$.

2.3. Implementación con punteros

La representación con punteros es como sigue:

```
struct nodo{
        nodo *izquierdo;
        nodo *derecho;
        int valor;
        }

typedef nodo *arbol;


// funcion Creo_vacio: crea un arbol sin nodos (valor NULL).
arbol Creo_vacio()
{       return NULL; }
```

es $O(1)$ y $\Omega(1)$.

```
// funcion Es_vacio: devuelve true si el arbol no tiene nodos.
bool Es_vacio(arbol T)
{     return (T==NULL); }
```

es $O(1)$ y $\Omega(1)$.

```
// funcion Valor: dada la direccion de un nodo devuelve su valor.
int Valor(arbol T)
{   return T->valor; }
```

es $O(1)$ y $\Omega(1)$.

```
// funcion Hijo_izq: dado un arbol, devuelve la direccion de su hijo
// izquierdo. Precondicion : el arbol es no vacio.
arbol Hijo_izq(arbol T)
{       return T->izquierdo; }
```

es $O(1)$ y $\Omega(1)$.

```
// funcion Hijo_der: dado un arbol, devuelve la direccion de su hijo
// derecho. Precondicion : el arbol es no vacio.
arbol Hijo_der(arbol T)
// precondicion T no vacio
{       return T->derecho; }
```

es $O(1)$ y $\Omega(1)$.

```
// funcion Buscar: dado un valor, lo busco en el arbol. Se utiliza la
// propiedad de que el arbol binario es de busqueda, luego si el valor
```

```
// es mayor que la raiz busco en el subarbol derecho mientras que si es
// menor busco en el subarbol izquierdo, esto lo hago recursivamente.
arbol Buscar(arbol T,int x)
{       if (Es_vacio(T)) return NULL;
        else if (Valor(T)==x) return T;
             else if (Valor(T)>x) return Buscar(Hijo_izq(T),x);
                  else return Buscar(Hijo_der(T),x);
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion buscar iterativa.
arbol buscar(arbol T,int x)
{
        while (T!=NULL && Valor(T)!=x)
        {       if (Valor(T)>x) T=Hijo_izq(T);
                else T=Hijo_der(T);
        }
        return T;
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Buscar_minimo: retorno la direccion del nodo con menor
// valor. Este es el nodo de mas a la izquierda en el arbol.
arbol Buscar_minimo(arbol T)
{       if (Es_vacio(T)) return NULL;
        while (!Es_vacio(Hijo_izq(T))) T=Hijo_izq(T);
        return T;
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Buscar_maximo: retorno la direccion del nodo con mayor
// valor. Este es el nodo de mas a la derecha en el arbol.
arbol Buscar_maximo(arbol T)
{       if (Es_vacio(T)) return NULL;
        while (!Es_vacio(Hijo_der(T))) T=Hijo_der(T);
        return T;
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// Funcion Insertar: agrego un nodo a un arbol con un valor dado.
// Si el valor es menor que la raiz del arbol lo inserto en el
// subarbol izquierdo mientras que si es mayor lo agrego en el
// subarbol derecho. El nodo que se inserta sera una hoja, o
// sea que se crea cuando el arbol que se pasa como parametro
// es vacio.
void Insertar(arbol &T,int x)
```

```
{     arbol q;
      if (Es_vacio(T))
      {q = new nodo;
       q->valor = x;
       q->izquierdo=NULL;
       q->derecho=NULL;
       T=q;
      }
      else
          if (x < Valor(T))
            Insertar(T->izquierdo,x);
          else if (x > Valor(T))
               Insertar(T->derecho,x);
               else cout << "Valor ya existe en el arbol"<< endl;
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// Funcion insertar iterativa.
arbol insertar(arbol T,int x)
{     arbol p,q,r=T;
      bool b;
      if (T==NULL)
      { q=new nodo;
      q->valor=x;
      q->izquierdo=NULL;
      q->derecho=NULL;
      return q;
      }
      while (T!=NULL)
      {     if (x<Valor(T)) {b=true;p=T;T=Hijo_izq(T);}
            else if (x>Valor(T)) {b=false;p=T;T=Hijo_der(T);}
                 else {cout << "Valor ya existe en el arbol "; return r;}
      }
      q=new nodo;
      q->valor=x;
      q->izquierdo=NULL;
      q->derecho=NULL;
      if (b) p->izquierdo=q;
      else p->derecho=q;
      return r;
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Borrar: borro en el subarbol izquierdo o derecho dependiendo
// del valor. Cuando encuentro el nodo con el valor dado, lo borro de
// la siguiente manera: si su hijo izquierdo es vacio, coloco en su lugar
// su hijo derecho, y si su hijo derecho es vacio, coloco en su lugar
// su hijo izquierdo. Si tiene los dos hijos, coloco en su lugar el
```

```
// valor minimo de su subarbol derecho (podria haber colocado el maximo
// de su subarbol izquierdo). Borro recursivamente el nodo correspondiente
// (tiene la propiedad de no tener hijo izquierdo).
void Borrar(arbol &T,int x)
{     arbol p,q;
      if (Es_vacio(T)) return;
      if (x < Valor(T)) Borrar(T->izquierdo,x);
      else if (x > Valor(T)) Borrar(T->derecho,x);
           else
               { q=T;
               if (Es_vacio(Hijo_izq(T)))
                  {T=Hijo_der(T);
                  delete q;}
               else if (Es_vacio(Hijo_der(T)))
                       { T=Hijo_izq(T);
                       delete q;}
                    else
                        {p=Buscar_minimo(Hijo_der(T));
                        T->valor=p->valor;
                        Borrar(T->derecho,p->valor);}
               }
}
```

es $O(log_2(n))$ y $\Omega(log_2(n))$.

```
// funcion Inorden: lista los nodos del arbol en una recorrida en inorden.
void Inorden(arbol T)
{     if (Es_vacio(T)) return;
      Inorden(Hijo_izq(T));
      cout << Valor(T) << endl;
      Inorden(Hijo_der(T));
}
```

es $O(n)$ y $\Omega(n)$.

```
// funcion Preorden: lista los nodos del arbol en una recorrida en preorden.
void Preorden(arbol T)
{     if (Es_vacio(T)) return;
      cout << Valor(T) << endl;
      Preorden(Hijo_izq(T));
      Preorden(Hijo_der(T));
}
```

es $O(n)$ y $\Omega(n)$.

```
// funcion Postorden: lista los nodos del arbol en una recorrida en postorden.
void Postorden(arbol T)
{     if (Es_vacio(T)) return;
      Postorden(Hijo_izq(T));
      Postorden(Hijo_der(T));
```

```
    cout << Valor(T) << endl;
}
```
es $O(n)$ y $\Omega(n)$.

2.4. Versión iterativa de inorden

Utilizamos un stack en el cual guardaremos punteros y enteros (utilizamos un tipo union con datos de estos tipos).

```
// nodo del arbol
struct nodo{
        nodo *izquierdo;
        nodo *derecho;
        int valor;
        };

typedef nodo *arbol;

// tipo union
union n_stack {
        int x;
        arbol t;
        };

// elemento del stack. b=true indica ns corresponde a un puntero
// b=false indica ns corresponde a un integer.

struct nodo_stack {
        n_stack ns;
        nodo_stack *sig;
        bool b;};

// elemento con datos del stack sin puntero a siguiente
struct datos_stack {
        n_stack ns;
        bool b;};

typedef nodo_stack *stack;
```

Las operaciones sobre el stack para el tipo de datos union anterior serían:

```
datos_stack top (stack l)
{               datos_stack d;
                d.ns=l->ns;
                d.b=l->b;
                return d; }


void push (stack &l,datos_stack a)
```

```
{     stack q;
      q = new nodo_stack;
      q->ns=a.ns;
      q->b=a.b;
      q->sig=l;
      l=q;
}
```

La operación pop no cambia.

Comenzamos colocando el puntero a la raiz del árbol en el stack. Luego, mientras tengamos elementos en el stack, si es un puntero y es no NULL, coloco en el stack su subarbol derecho, la raiz y luego el subarbol izquierdo. Si es un entero lo imprimo. El programa queda como sigue:

```
void inorden(arbol T)
{     stack s;
      datos_stack ns1,ns2;
      datos_stack d;
      s=creo_vacio();
      ns1.b=true;
      ns1.ns.t=T;
      push(s,ns1);
      while (!es_vacio(s))
      {       d=top(s);
              pop(s);
              if (d.b)
              { if (d.ns.t!=NULL)
                    { ns2.b=true;
                      ns2.ns.t=d.ns.t->derecho;
                      push(s,ns2);
                      ns2.b=false;
                      ns2.ns.x=d.ns.t->valor;
                      push(s,ns2);
                      ns2.b=true;
                      ns2.ns.t=d.ns.t->izquierdo;
                      push(s,ns2);
                    }
              }
              else cout << d.ns.x << endl;
      }
}
```

3. El TAD Árbol m-ario

Podemos ver los árboles tanto como un TAD como una estructura de datos.

Veremos operaciones útiles en árboles y veremos como algoritmos en árboles se pueden diseñar en términos de éstas operaciones.

Operaciones:

1. Padre(n,T) : retorna (la dirección de) el padre del nodo n en el árbol T. Si n es la raiz se retorna NULL.
2. Es_raiz(n,T) : retorna true si n corresponde a la raiz.
3. Es_hoja(n,T) : retorna true si n corresponde a un nodo sin hijos.
4. Hijo_mas_izquierdo(n,T) : retorna (la dirección de) el hijo más izquierdo del nodo n en el árbol T. Si n es una hoja retorna NULL.
5. Hermano_derecho(n,T) : retorna (la dirección de) el hermano derecho de n en T. Éste es el nodo hijo del mismo padre que está inmediatamente a la derecha de n.
6. Valor(n,T): retorna la información correspondiente al nodo n en el árbol T.
7. Crear_raiz(v) : crea un árbol con un unico nodo con valor v.
8. Agregar_hijo(T,v,p) : agrega un nodo como hijo del nodo con dirección p. Se ubicará como nodo más a la derecha entre los hijos.
9. Raiz(T) : retorna la información del nodo raiz del árbol T y true o false si T es el árbol vacio.
10. Buscar(T,v) : retorna (la dirección de) el nodo con valor v.
11. Borrar(T,i) : borra el nodo con dirección i del árbol.
12. Inorden(T) : lista los nodos de T en inorden.
13. Preorden(T) : lista los nodos de T en preorden.
14. Postorden(T) : lista los nodos de T en postorden.

4. Implementaciones de árboles

4.1. Representación de árboles con arreglos de hijos

Una forma importante y útil de representar árboles es tener para cada nodo una lista de sus hijos. Podemos representar estas listas por arreglos si conocemos la cantidad máxima de hijos que puede tener un nodo.

Para esta primer representación supongamos tenemos además de una cantidad maxima de hijos por nodo una cantidad acotada de nodos en el árbol (maxnodos).

La representación es:

```
struct nodo{
        int padre;
        int valor;
        int hijos[maxhijos];
        int cantidad_hijos;
        };

struct arbol{
        int raiz;
        nodo info[maxnodos];
        int cantidad_nodos;
        };
```

En la estructura, raiz corresponde al indice de la raiz que será 0, los nodos tienen asociado un indice que es el indice en el arreglo info, los indices de los nodos están numerados consecutivamente de 0 a cantidad_nodos-1. En los nodos padre es el indice del padre del nodo, el arreglo de hijos contiene los indices de los hijos del nodo y los hijos de un nodo van del 0 a cantidad_hijos-1.

En las distintas operaciones tenemos opciones respecto a que argumentos pasar. Podemos pasar un entero n que representa la posición en el array del nodo o podemos pasar un entero m que representa el valor del elemento en el nodo.

En general vamos a seguir la primer opción, pero no tenemos que preocuparnos porque estas opciones son equivalentes en el sentido de que dada la posición del elemento en el arreglo podemos retornar el valor y dado el valor podemos retornar el indice del elemento (suponiendo no hay valores repetidos).

```
// Funcion Buscar: retorna el indice en el array del elemento
// con dato valor. Si dicho valor no se encuentra retorna -1.
int Buscar(arbol T, int valor)
{   int i;
    for (i=0;i<T.cantidad_nodos && T.info[i].valor!=valor;i++) ;
    if (T.info[i].valor==valor) return i;
    else return -1;
}


// Funcion Es_raiz: retorna true si i==0, esto es, i corresponde
// al indice de la raiz. Retorna false en otro caso.
bool Es_raiz(arbol T,int i)
{     return (i==0); }


// Funcion Es_hoja: retorna true si el nodo con indice i no tiene
// hijos. Retorna false en otro caso.
// Precondicion: i corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
bool Es_hoja(arbol T,int i)
```

```
{     return T.info[i].cantidad_hijos==0; }


// Funcion Padre: Esta funcion retorna el indice correspondiente al padre
// del nodo con indice i. Se retorna -1 si el nodo es la raiz (no tiene
// padre), en otro caso el nodo tiene padre y se retorna su indice.
// Precondicion: i corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
int Padre(arbol T, int i)
{
    if (Es_raiz(T,i)) return -1;
    else return T.info[i].padre;
}


// Funcion Hijo_mas_izquierdo: Los nodos hijos se insertan siguiendo
// el orden de izquierda a derecha. Luego el Hijo_mas_izquierdo
// corresponde al indice 0 en el array de hijos. Si el nodo fuera
// una hoja no tiene hijos, se retorna -1.
// Precondicion: i corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
int Hijo_mas_izquierdo(arbol T, int n)
{   if (Es_hoja(T,n)) return -1;
    else return T.info[n].hijos[0]; }


// Funcion siguiente: es una funcion auxiliar utilizada por la funcion
// Hermano_derecho. Recibe como argumento el indice n de un elemento
// y el indice p de su padre. Retornara el indice del hermano
// derecho del nodo con indice n. A continuacion busca en el arreglo de
// hijos de p el elemento de indice n. Si este es el ultimo nodo en
// el arreglo de hijos o si no esta en el arreglo de hijos,
// no tiene hermano derecho retorna -1. En caso contrario el nodo con
// indice n tiene hermano derecho, estando el indice de este en el array de hijos
// de p, en la posicion siguiente a la posicion del nodo n.
// Observar que como siguiente es invocado con el padre de n,
// n estara siempre como hijo.
int siguiente(arbol T, int n, int p)
{   int i;
    for(i=0;T.info[p].hijos[i]!=n && i<T.info[p].cantidad_hijos;i++) ;
    if (i>=T.info[p].cantidad_hijos) return -1;
    else return T.info[p].hijos[i+1];
}


// Funcion Hermano_derecho: recibe el indice de un elemento y retorna
// el indice de su hermano derecho si existe. Utiliza la funcion siguiente.
// Si no hay hermano derecho retorna -1. Si el nodo de indice i es la raiz
// retorna -2 (luego podemos diferenciar los distintos casos en que no
// hay hermano derecho).
```

```
// Precondicion: i corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
int Hermano_derecho(arbol T, int i)
{   if (Es_raiz(T,i)) return -2;
    else return siguiente(T,i,T.info[i].padre);
}


// Funcion Valor: devuelve el valor correspondiente al nodo de indice n.
// Precondicion: i corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
int Valor(arbol T, int n)
{ return T.info[n].valor; }


// Funcion Crear_raiz: Devuelve un arbol con un unico nodo que corresponde
// con la raiz y cuyo valor es el dado como argumento.
arbol Crear_raiz(int valor)
{       arbol T;
        T.raiz=0;
        T.info[0].padre=-1;
        T.info[0].cantidad_hijos=0;
        T.info[0].valor=valor;
        T.cantidad_nodos=1;
        return T;
}


// Funcion Raiz: Retorna el valor del nodo raiz en un arbol.
int Raiz(arbol T)
{   return T.info[T.raiz].valor; }


// Funcion Agregar_hijo: coloca la informacion de un nuevo hijo
// en la primer posicion libre del arreglo de nodos. Recibe como
// argumentos el indice de su padre, y el valor del nuevo hijo.
// Su informacion sera: 0 hijos, el padre dado, el valor dado.
// Ademas se coloca como hijo del padre (en el arreglo de hijos).
// Se incrementan cantidad de hijos del padre y cantidad de nodos.
// Precondicion: padre corresponde a un nodo, es decir es mayor o igual que 0
// y menor que cantidad_nodos.
void Agregar_hijo(arbol &T, int valor_hijo, int padre)
{       int j=T.cantidad_nodos;
        T.info[j].cantidad_hijos=0;
        T.info[j].padre=padre;
        T.info[j].valor=valor_hijo;
        T.cantidad_nodos++;
        T.info[padre].hijos[T.info[padre].cantidad_hijos]=j;
        T.info[padre].cantidad_hijos++;
}
```

```
// Funcion copiar_hijos: es una funcion auxiliar utilizada
// por la funcion copiar_nodo. Copia un array de hijos en otro.
void copiar_hijos(int a1[],int a2[],int cant)
{     int j;
      for (j=0;j<cant;j++)
          a1[j]=a2[j];
}


// Funcion copiar_nodo: es una funcion auxiliar utilizada por
// borrar. Copia la informacion de un nodo en otro.
void copiar_nodo(nodo &n1,nodo &n2)
{
      n1.padre=n2.padre;
      n1.valor=n2.valor;
      copiar_hijos(n1.hijos,n2.hijos,n2.cantidad_hijos);
      n1.cantidad_hijos=n2.cantidad_hijos;
}


// Funcion renumerar: es una funcion auxiliar utilizada por borrar.
// Cuando borro un nodo con indice i, debo cambiar el indice de los
// nodos con indice mayor que i (corriendolos a la posicion con
// indice uno menor). Debo hacer esto en los arreglos de hijos
// y en el arreglo de nodos. renumerar modifica los arreglos de hijos.
// modifico los hijos con indice mayor que i.
// Observar que los hijos de un nodo se numeran de izquierda a derecha
// en forma ascendente. void renumerar(arbol &T, int padre, int hijo)
{                                  int i,j;
                                   if (Es_hoja(T,padre)) return;
                                   for(i=0;i<T.info[padre].cantidad_hijos && T.info[padre].hijos[i]<hijo;i++);
                                   if (i==T.info[padre].cantidad_hijos) return;
                                   if (T.info[padre].hijos[i]==hijo)
                                     { for (j=i+1;j<T.info[padre].cantidad_hijos;j++)
                                           T.info[padre].hijos[j-1]=T.info[padre].hijos[j] - 1;
                                     T.info[padre].cantidad_hijos - -;
                                     }
                                   else return;
}


// Funcion Borrar: utiliza renumerar. Primero invoco renumerar para todos
// los nodos, con lo que actualizo los arreglos de hijos de todos los nodos.
// Luego borro la informacion del nodo con indice i, corriendo los nodos
// con indice mayor una posicion hacia adelante. Para esto ultimo utiliza
// copiar_nodo.
// Precondicion: el nodo con indice i no tiene hijos y el indice i
// corresponde a un nodo.
```

```
void Borrar(arbol &T, int i)
{
      int j;
      for (j=0;j<T.cantidad_nodos;j++)
          renumerar(T,j,i);
      for (j=i;j<T.cantidad_nodos-1;j++)
          copiar_nodo(T.info[j],T.info[j+1]);
      T.cantidad_nodos - -;
}


// Funcion inorden : lista los nodos del arbol bajo este recorrido.
void inorden(arbol T,int n)
{     int i;
      if (Es_hoja(T,n)) {cout << Valor(T,n) << endl; return;}
      inorden(T,Hijo_mas_izquierdo(T,n));
      cout << Valor(T,n) << endl;
      for (i=1;i<T.info[n].cantidad_hijos;i++)
          inorden(T,T.info[n].hijos[i]);
}


// Funcion preorden : lista los nodos del arbol bajo este recorrido.
void preorden(arbol T,int n)
{     int i;
      if (Es_hoja(T,n)) {cout << Valor(T,n) << endl; return;}
      cout << Valor(T,n) << endl;
      for (i=0;i<T.info[n].cantidad_hijos;i++)
          preorden(T,T.info[n].hijos[i]);
}


// Funcion postorden : lista los nodos del arbol bajo este recorrido.
void postorden(arbol T,int n)
{     int i;
      if (Es_hoja(T,n)) {cout << Valor(T,n) << endl; return;}
      for (i=0;i<T.info[n].cantidad_hijos;i++)
          postorden(T,T.info[n].hijos[i]);
      cout << Valor(T,n) << endl;
}
```

4.2. Una representación con punteros

Una posible representación es con punteros donde cada nodo apunta a su primer hijo y a su hermano derecho. Tambien se mantiene la direccion del padre.

Veremos las operaciones para esta última representación.

```
struct nodo{
       nodo *padre;
```

```
        int valor;
        nodo *hijos;
        nodo *hermano;
        };

typedef nodo *arbol;


arbol Crear_vacio()
{ return NULL; }


bool Es_vacio(Arbol T)
{ return T==NULL; }


// Funcion Es_raiz: retorna true si el nodo no tiene padre y falso
// en caso contrario.
bool Es_raiz(arbol T)
{ return T->padre==NULL; }


// Funcion Es_hoja: retorna true si el nodo no tiene hijos y falso
// en caso contrario.
bool Es_hoja(arbol T)
{ return T->hijos==NULL; }


// Funcion Crear_raiz: recibe el valor correspondiente a la raiz
// y retorna un arbol de un solo nodo con ese valor como dato.
arbol Crear_raiz(int valor)
{       arbol T;
        T = new nodo;
        T->padre=NULL;
        T->valor=valor;
        T->hijos=NULL;
        T->hermano=NULL;
        return T;
}


// Funcion Hijo_mas_izquierdo: retorna NULL si no hay hijos
// y la direccion del hijo mas izquierdo si lo hay.
arbol Hijo_mas_izquierdo(arbol T)
{       if (Es_hoja(T)) return NULL;
        return T->hijos;
}


// Funcion Hermano_derecho: retorna NULL si el nodo es la raiz
```

```
// y la direccion del hermano si no es la raiz. Esta ultima
// direccion sera NULL si no hay hermano.
arbol Hermano_derecho(arbol T)
{       if (Es_raiz(T)) return NULL;
        return T->hermano;
}


// Funcion Buscar: retorna un puntero al nodo con valor dado.
// Si no hay retorna NULL. La estrategia utilizada para buscar
// el nodo es una recorrida en preorden. Luego si la direccion
// dada corresponde al valor devuelvo esta direccion. Si el
// nodo es una hoja termina la recorrida, sino busco primero
// en el hijo mas izquierdo. Si encuentro el valor devuelvo
// el puntero, sino recorro los hermanos. Si en la recorrida
// encuentro el valor retorno la direccion, sino retorno NULL.
arbol Buscar(arbol T,int valor)
{       arbol p,q,l;
        if (Es_hoja(T))
          if ((T->valor)==valor) return T;
          else return NULL;}
        if ((T->valor)==valor) return T;
        q=Hijo_mas_izquierdo(T);
        p=q;
        l=Buscar(q,valor);
        if (l!=NULL) return l;
        while (Hermano_derecho(p)!=NULL)
        {
                q=Hermano_derecho(p);
                l=Buscar(q,valor);
                if (l!=NULL) return l;
                p=q;
        }
        return NULL;
}


// Funcion Padre: retorna la direccion del nodo padre del nodo con direccion
// dada. Retorna NULL si es la raiz.
// Precondicion: la direccion p existe en el arbol.
arbol Padre(arbol T, arbol p)
{
        if (Es_raiz(p)) return NULL;}
        return p->padre;
}


// Funcion Valor: retorna el valor del nodo con direccion dada.
// Precondicion: la direccion p existe en el arbol.
int Valor(arbol p)
```

```
{   return p->valor; }


// Funcion Agregar_hijo: recibe el valor del nuevo hijo y la direccion
// del padre. Crea un nuevo nodo y carga la informacion. Si el padre
// es una hoja es su primer hijo, luego coloca su direccion en los
// hijos del padre. Si el padre tiene hijos lo coloca como ultimo
// hermano del primer hijo.
// Precondicion: la direccion del padre existe en el arbol.
void Agregar_hijo(int valor_hijo, arbol padre)
{     arbol q,p;
      q = new nodo;
      q->padre=padre;
      q->valor=valor_hijo;
      q->hijos=NULL;
      q->hermano=NULL;
      if (Es_hoja(padre)) {padre->hijos=q; return;}
      p=padre->hijos;
      while (p-¿hermano!=NULL) p=p-¿hermano;


// Funcion Borrar: borra el nodo con valor dado. Invoca Buscar para hallar
// su direccion. Si el nodo no existe no modifica el arbol. Si es la raiz
// y no tiene hijos devuelve NULL. Si tiene hijos no se borra (es una
// precondicion no tener hijos). En otro caso, es una hoja. Para borrarlo
// considero los hijos del padre, si el valor corresponde al primer hijo,
// pongo el hermano derecho de este como primer hijo del padre, sino
// recorro la lista de hermanos buscando el nodo con el valor dado. Cuando
// lo encuentro lo quito de la lista de hermanos.
arbol Borrar(arbol T, int valor)
{
       arbol p,q,l;
       p=Buscar(T,valor);
       if (p==NULL) return T;
       if (!Es_hoja(p)) return T;
       q=p->padre;
       l=q->hijos;
       if (l->valor == valor) {q->hijos=l->hermano; delete l;return T;}
       while (l->hermano!=NULL && l->hermano->valor!=valor) l=l->hermano;
       if (l->hermano!=NULL)
         if (l->hermano->hijos!=NULL)
         { cout << "Nodo con hijos, no puede borrarse";
           return T;}
         else { l->hermano=l->hermano->hermano;
                delete l->hermano;
                return T;}
}


// Funcion inorden: imprime los datos de los nodos del arbol en recorrida inorden.
```

```
void inorden(arbol T)
{     arbol p,q;
      if (Es_hoja(T)) {cout << Valor(T) << endl; return;}
      q=Hijo_mas_izquierdo(T);
      p=q;
      inorden(q);
      cout << Valor(T) << endl;
      while (Hermano_derecho(p)!=NULL)
      {       q=Hermano_derecho(p);
              inorden(q);
              p=q;
      }
}


// Funcion preorden: imprime los datos de los nodos del arbol en recorrida preorden.
void preorden(arbol T)
{     arbol p,q;
      if (Es_hoja(T)) {cout << Valor(T) << endl; return;}
      cout << Valor(T) << endl;
      q=Hijo_mas_izquierdo(T);
      p=q;
      preorden(q);
      while (Hermano_derecho(p)!=NULL)
      {       q=Hermano_derecho(p);
              preorden(q);
              p=q;
      }
}


// Funcion postorden: imprime los datos de los nodos del arbol en recorrida postorden.
void postorden(arbol T)
{     arbol p,q;
      if (Es_hoja(T)) {cout << Valor(T) << endl; return;}
      q=Hijo_mas_izquierdo(T);
      p=q;
      postorden(q);
      while (Hermano_derecho(p)!=NULL)
      {       q=Hermano_derecho(p);
              postorden(q);
              p=q;
      }
      cout << Valor(T) << endl;
}
```