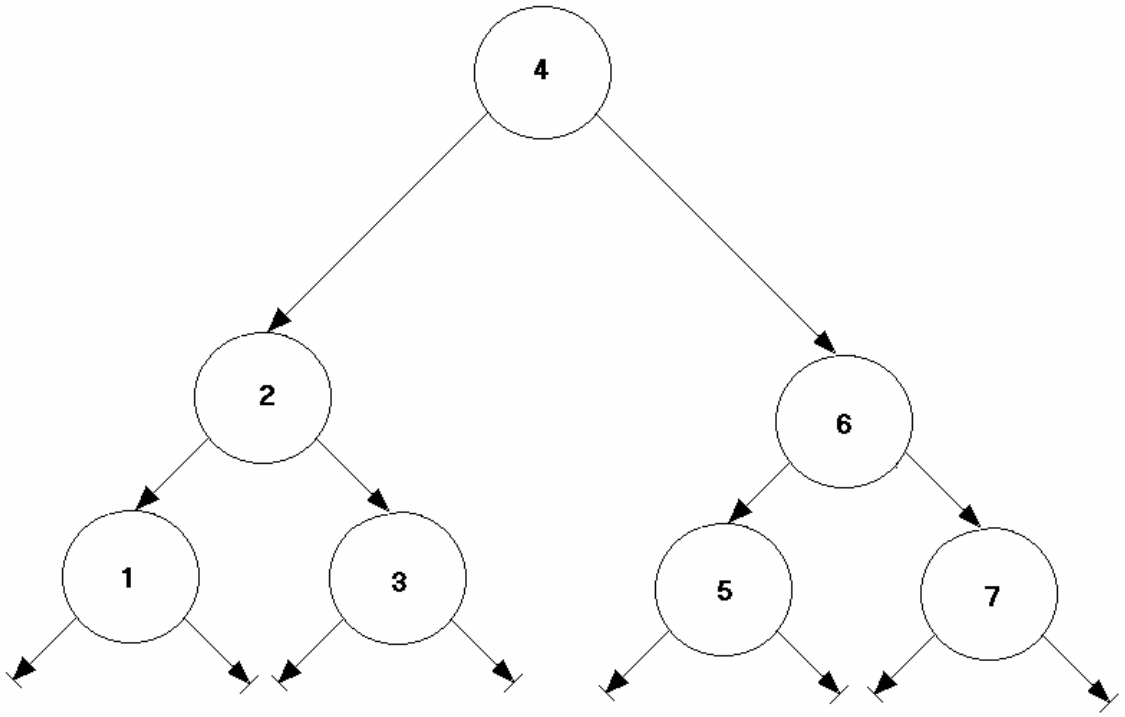
Apuntes elaborados por: Eduardo Quevedo, Aaron Asencio y Raquel López
Revisado por: Javier Miranda el ????

Tema 7: Árbol Binario

En el árbol binario se combina lo mejor del array (acceso rápido a elementos, ej. Búsqueda binaria) y lo mejor de la lista dinámica (no hay que fijar de antemano el tamaño máximo, no hay que abrir hueco para insertar un elemento y no hay que cerrar el hueco al borrar un elemento).

Si se tuvieran los elementos a insertar sería ideal y sencillo dado que se podrían colocar en su posición perfecta, pero no es así, no se tienen los datos desde el principio, así que se sigue el criterio de tener los menores a la izquierda y los mayores a la derecha, así que si el primer elemento es muy pequeño o muy grande puede ocurrir que tengamos una lista en vez de un árbol, esto se soluciona si se vuelca el contenido en un array y se vuelve a insertar, por lo que tendríamos un árbol bien organizado como el siguiente:



Los árboles se implementan fundamentalmente con listas simplemente enlazadas, ya que no hay interés de tener una doble.

Se pueden tener árboles balanceados que se van reajustando si se sobrepasa el límite, esto no se implementará en la asignatura.

No hay que hacer métodos de ordenación lógicamente.

Terminología de árboles binarios:

Hoja:
Nodo sin hijos, en estos nodos es en los que vamos a realizar la inserción.

Subárbol:
Subconjunto del árbol.

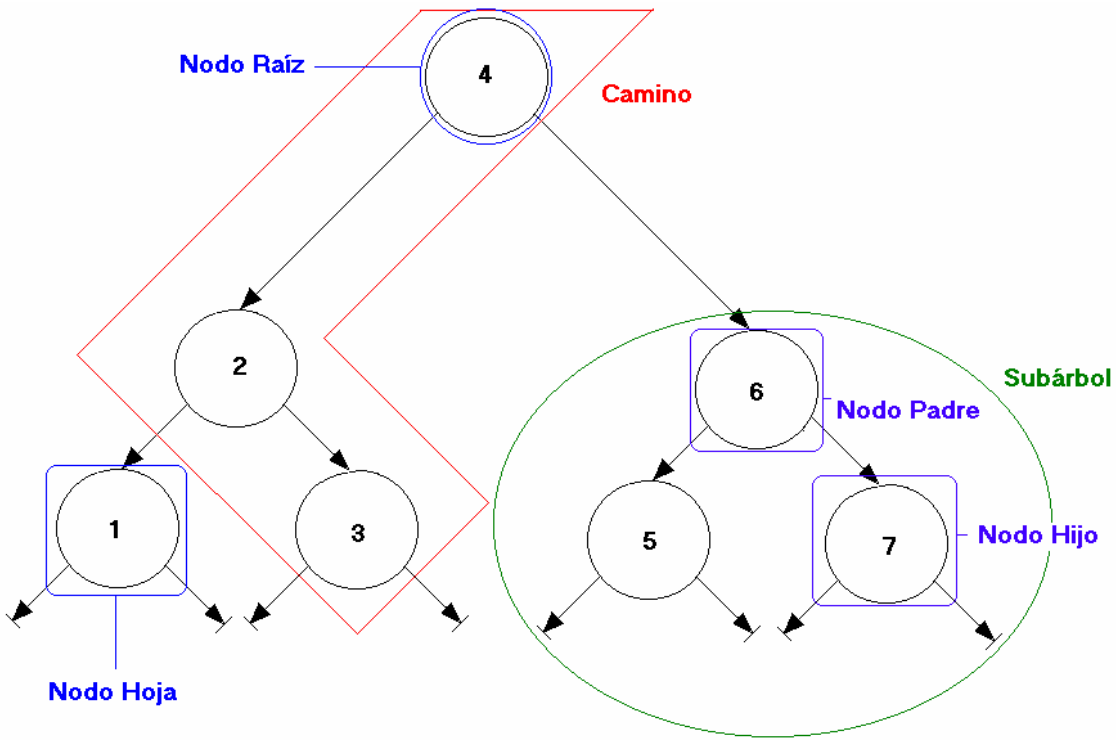
Raíz:
El padre de todos los nodos.

Niveles:
Dos nodos se dice que están al mismo nivel si están a la misma profundidad en el árbol.

Camino:
Recorrido de un nodo a otro.

Padre:
Nodo anterior a un nodo.

Hijo:
Nodo al que otro nodo está apuntando (su padre).



Estructuras de datos

Las declaraciones que necesitamos para declarar un árbol son similares a las que hemos utilizado para listas, se vuelve a hacer limited private y en lugar de tener para el nodo un Siguiente se tienen dos campos: Izquierda y Derecha, para el árbol se designa raíz que será el primer nodo del árbol:

```
package Arbol is
        type T_Arbol is  limited private;
private
        type T_Nodo
        type T_Nodo_Ptr is access T_Nodo;
        type T_Nodo is
          record
            Info       : T_Info;
            Izquierda : T_Nodo_Ptr := null;
            Derecha  : T_Nodo_Ptr := null;
            - -  Ahora se les llama Izquierda y Derecha a los punteros a
            - -  nodos
          end record;
        type T_Arbol is
          record
            Raiz : T_Nodo_Ptr := null;
          end record;
end Arbol;
```

Funcion “Existe”

Comencemos viendo una posible implementación de la función *Existe* que comprueba si existe un elemento en un árbol, se considera que Info es un entero.

```
        function Existe (Arbol : in T_Arbol;
                         Info    : in Integer) return Boolean is
          Actual : T_Nodo_Ptr := Arbol.Raiz;
        begin
          while Actual /= null loop
            - -  Se considerará que si es que si el valor es menor que el
            - -  actual se va a la izquierda y si no a la derecha
            if Info = Actual.Info then
              return True;
            elsif Info < Actual.Info then
              Actual := Actual.Izquierda;
            else
              Actual := Actual.Derecha;
            end if;
          end loop;
          return False;
        end Existe;
```

Procedimiento Insertar

Cabe destacar que se tendrá en cuenta el caso de duplicados.

```
procedure Insertar
  (Arbol : in out T_Arbol;
   Valor : in       Integer)
is
   Actual    : T_Nodo_Ptr := Arbol.Raiz;
   Anterior : T_Nodo_Ptr := null;
   Nuevo    : T_Nodo_Ptr := null;
begin

   --  Buscamos la posición donde hay que insertar el nuevo valor. Para ello
hacemos un
   --  bucle mientras no lleguemos al final de una rama y mientras el valor del
nodo en el
   --  que estamos no coincida con el valor que queremos insertar:
   while Actual /= null and then Actual.Info /= Valor loop
      Anterior := Actual;
      if Valor < Actual.Info then
         Actual := Actual.Izquierda;
      else
         Actual := Actual.Derecha;
      end if;
   end loop;

   --  Si al salir de este bucle "Actual" no es null significa que se encontró un
nodo cuyo
   --  valor coincide con el valor que queríamos insertar. Puesto que no
permitimos
   --  elementos duplicados, elevamos la correspondiente excepción:
   if Actual /= null then

      raise Duplicado;

   --  En otro caso podemos insertar sin problemas:
   else

      --  Pedimos memoria y guardamos la información en el nuevo nodo:
      Nuevo       := new T_Nodo;
      Nuevo.Info := Valor;

      --  Si "Anterior" es null, es decir, si el árbol está vacío:
      if Anterior = null then
         Arbol.Raiz := Nuevo;
      else
```

```
        -- Comprobamos si el valor a insertar es mayor o menor que el valor del
nodo
        -- apuntado por "Anterior", para saber si hay que insertar a la izquierda o
a la
        -- derecha:
        if Valor < Anterior.Info then
          Anterior.Izquierda := Nuevo;
        else
          Anterior.Derecha  := Nuevo;
        end if;

        -- En lugar de usar el nodo "Anterior" para preguntar en este if, se podía
haber
        -- usado una variable de tipo Boolean o de tipo T_Dirección
        --                  type T_Dirección is (Izquierda, Derecha);
        --  que nos vaya diciendo, a medida que hacemos el recorrido, en qué
dirección
        -- nos movemos.

     end if;
   end if;
end Insertar;
```

Procedimiento Borrar

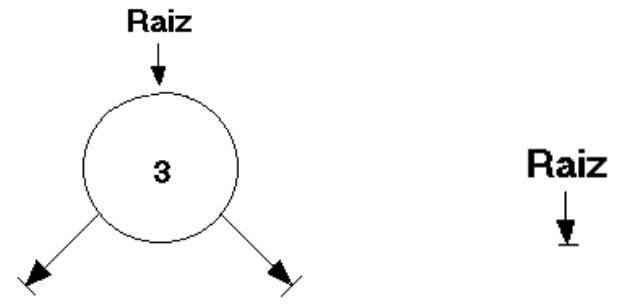
Este es posiblemente el procedimiento más complicado de la asignatura, hay una gran cantidad de casos particulares y el caso general es bastante complicado, sería bueno que los casos particulares fueran tratados en procedimientos aparte a la hora de hacer la práctica. Se distinguirán los casos particulares y el caso general.

Para implementar este procedimiento hay que tener en cuenta, a grandes rasgos, cinco casos:

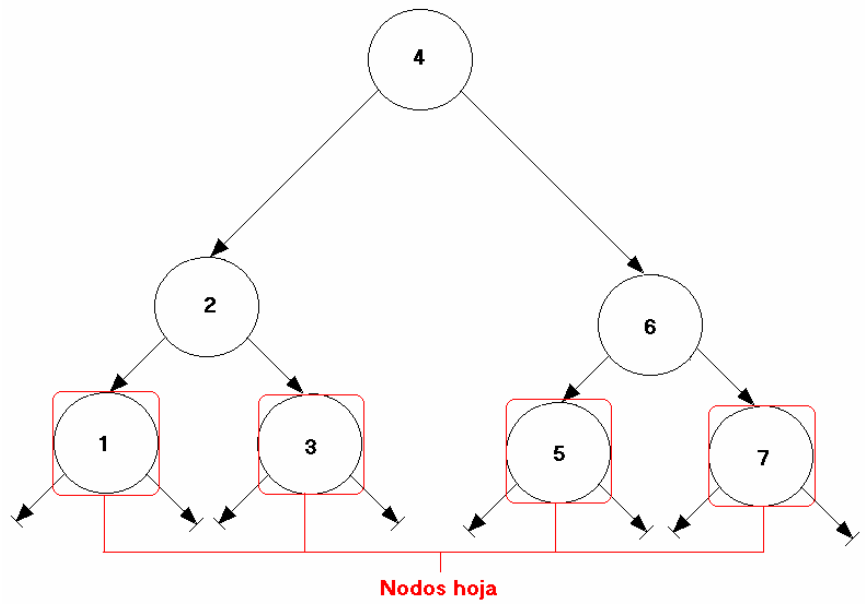
- Árbol vacío.
- Árbol con un solo elemento.
- El nodo a borrar es un nodo hoja.
- El nodo a borrar tiene un único hijo, en suyo caso éste ocupará el lugar del nodo que vamos a borrar.
- El nodo a borrar tiene dos hijos. En este caso hay que buscar un nodo que pueda ocupar el lugar del que vamos a borrar de forma que se respete la estructura del árbol: para ello, el nodo sustituto debe ser el mayor de los menores (el situado más a la derecha del hijo izquierdo de la raíz del árbol), o bien, el menor de los mayores (el situado más a la izquierda del hijo derecho de la raíz del árbol)

Caso 1: Borrar la raíz

Es el más sencillo, tan solo hay que hacer free del nodo y poner Raiz a null:

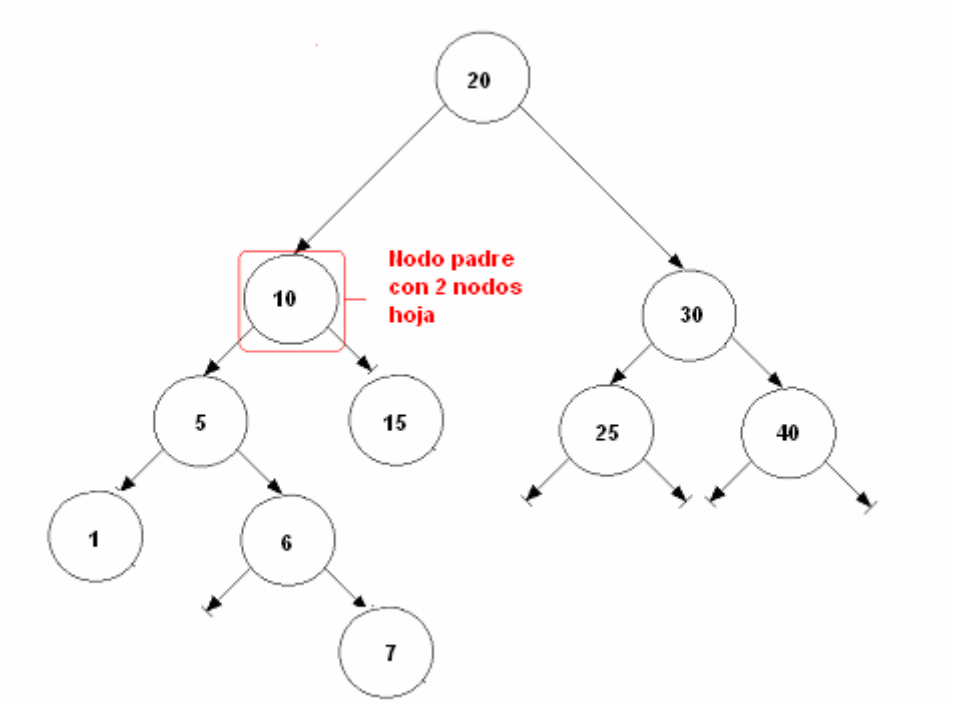


Caso 2: Borrar un nodo hoja



Los nodos hoja se borran también fácilmente, basta con hacer free y poner el anterior a null.

Caso 3: Borrar un nodo padre de dos nodos hoja

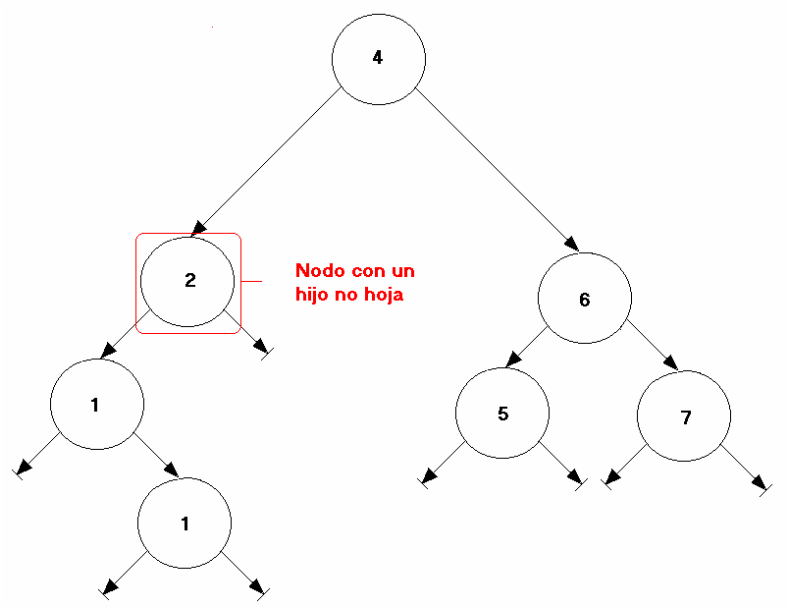


En este caso debemos encontrar un sustituto ideal para reemplazar al padre con dos hijos. Hay dos posibilidades:

1) El sustituto ideal puede ser el mayor del sub-arbol izquierdo del padre. (en nuestro ejemplo el 7)
2) El sustituto ideal puede ser el menor del sub-arbol derecho del padre. (en nuestro ejemplo el 15)

Para el árbol de la figura anterior sería más conveniente la primera opción porque el árbol resultante estaría más “equilibrado” que si resolvemos utilizando la opción 2. Para elegir una opción u otra podríamos hacer una función que nos calcule cual de los sub-arboles del padre tiene mayor número de elementos.

Caso 4: Borrar un nodo padre con un solo hijo que no es hoja

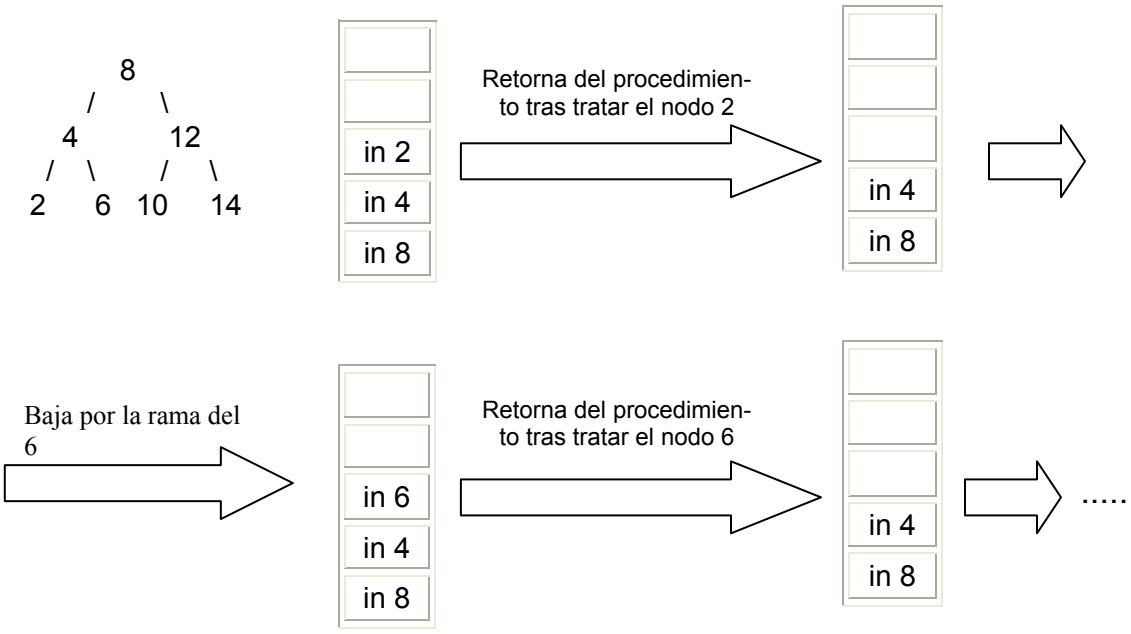


Para borrar este nodo su sustituto ideal es el hijo izquierdo. En caso de tener sólo el hijo derecho, el sustituto ideal es el hijo derecho.

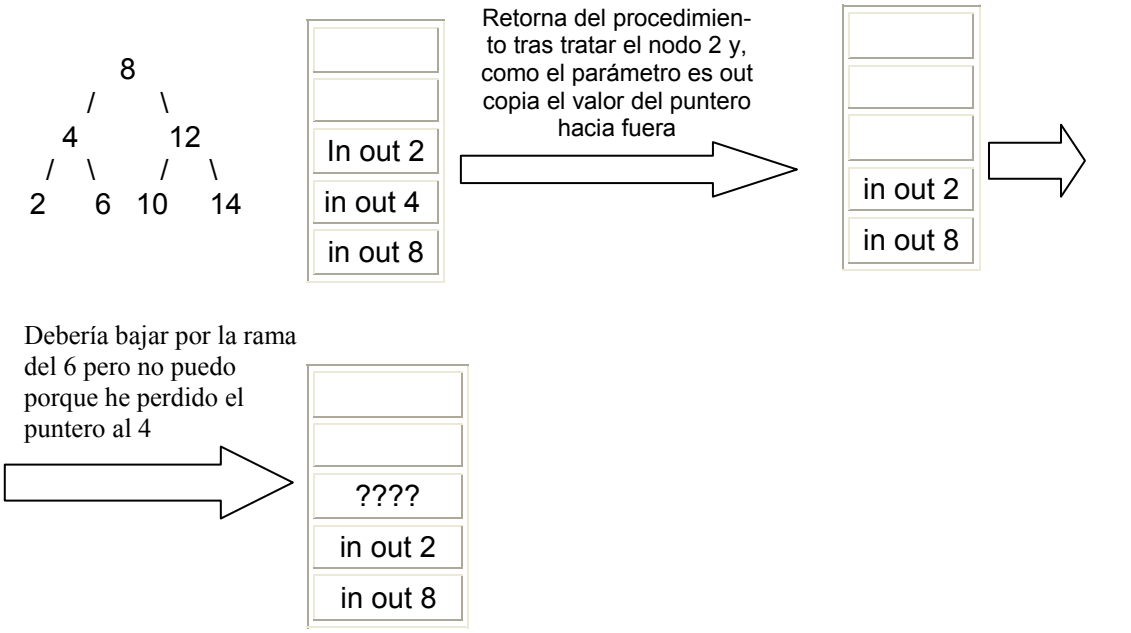
TIPOS DE RECORRIDO

La forma más sencilla de recorrer el árbol es utilizando *recursividad*, pues tiene la ventaja de que si el parámetro del procedimiento (el puntero al nodo actual) se pasa en modo in, la dirección del nodo anterior se guarda en la pila que usa el compilador, con lo que no perdemos ninguna dirección. Si no podemos utilizar la recursividad habría que hacer una solución iterativa que utilizase una pila de punteros.

> ***NOTA:*** En los procedimientos recursivos en los que vayamos a recorrer el árbol, el parámetro de tipo T_Nodo_Ptr que usamos como índice debe ser de entrada exclusivamente ya que si fuese de entrada-salida al llegar al final de una rama no podríamos volver a subir en el árbol. Esto es así porque si se pone como parámetro de modo out al salir del procedimiento se copia el valor final del puntero para devolverlo al punto de la llamada (con lo que al retornar ya no tenemos el valor anterior del puntero, que era el puntero al nodo anterior)..
> Lo vemos mejor en el siguiente ejemplo:



Sin embargo si el modo del parámetro fuese “in out” veamos que el recorrido falla:



Existen 3 recorridos básicos de un árbol:

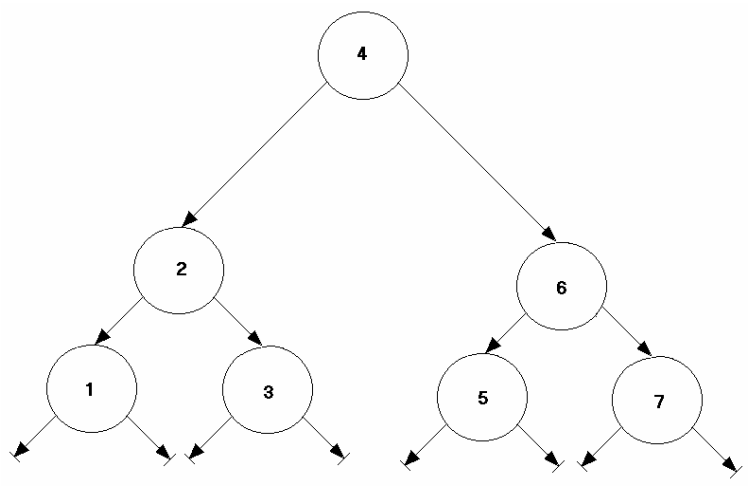
- ***Pre-order*** **:** Primero se procesan los nodos "padre" y después los nodos "hijo". Este tipo de recorrido se usa, por ejemplo, para *guardar un árbol en un fichero* y reconstruirlo posteriormente exactamente cómo estaba. Cuando queramos leer de fichero para reconstruir el árbol lo único que hay que hacer es ir insertando los elementos en el árbol según el orden en que están dispuestos en el fichero. Así conseguimos construir un árbol idéntico al que guardamos.
- ***In-order*** **:** Se recorre el árbol en orden. Este tipo de recorrido se usa para recorrer todos los nodos de menor a mayor o de mayor a menor. Una posible aplicación seria *equilibrar las ramas de un árbol*; la forma de hacer este procedimiento es: recorremos el árbol en in-order y vamos guardando los elementos en un array; como este array estará ordenado, hacemos un recorrido binario del mismo y vamos reinsertando los elementos en un nuevo árbol que finalmente estará equilibrado.
- ***Post-order*** **:** Primero se procesan los hijos y después el padre. Este tipo de recorrido se puede utilizar, por ejemplo, para *borrar todos los nodos de un árbol*. De esta forma evitamos perder nodos ya que si usásemos otro recorrido, como por ejemplo el recorrido en pre-order, estaríamos borrando primero el nodo padre, con lo que habríamos perdido la dirección de los nodos que vamos a borrar.

Las implementaciones recursivas serían las siguientes:

```
-- Pre_Order

procedure Recorrer_Pre_Order (Nodo : in T_Nodo_Ptr) is
begin
   if Nodo = null then
      return;
   end if;
   Mostrar_Contenido (Nodo.Info);
   Recorrer_Pre_Order (Nodo.Izquierda);
   Recorrer_Pre_Order (Nodo.Derecha);
end Recorrer_Pre_Order;

-- In_Order

procedure Recorrer_In_Order (Nodo : in T_Nodo_Ptr) is
begin
   if Nodo = null then
      return;
   end if;
   Recorrer_In_Order (Nodo.Izquierda);
   Mostrar_Contenido (Nodo.Info);
   Recorrer_In_Order (Nodo.Derecha);
end Recorrer_In_Order;
```

```
-- Post_Order


procedure Recorrer_Post_Order (Nodo : in T_Nodo_Ptr) is
begin
   if Nodo = null then
      return;
   end if;
   Recorrer_Post_Order (Nodo.Izquierda);
   Recorrer_Post_Order (Nodo.Derecha);
   Mostrar_Contenido (Nodo.Info);
end Recorrer_Post_Order;
```

Para el siguiente árbol:



Los resultados serían los siguientes:

Pre_Order: 4-2-1-3-6-5-7
In_Order: 1-2-3-4-5-6-7
Post_Order: 1-3-2-5-7-6-4

Los recorridos del árbol explicados anteriormente, se pueden implementar con la ayuda de una pila o de una pila y un array en determinados casos.

Ejercicio de Profundidad:

Se hizo un ejercicio que recursivamente halla la profundidad de un árbol, para ello se usa la fórmula de

1 + Máximo (Profundidad (Izquierda), Profundidad (Derecha))

Si la raíz es nula devuelve directamente 0, si no usa la fórmula.

La función usa otra auxiliar para usar punteros a nodo.

```
function Profundidad (Arbol : in T_Arbol) return Natural
  is
    function Maximo
       (Izquierda : in Natural;
       Derecha   : in Natural)
       return Natural
    is
    begin
      if Izquierda <= Derecha then
        return Derecha;
      else
        return Izquierda;
      end if;
    end Maximo;
    function Profundidad_Auxiliar (Nodo : in T_Nodo_Ptr) return Natural
    is
    begin
      if Nodo = null then
        return 0;
      else
        return 1 + Maximo (Profundidad_Auxiliar (Nodo.Izquierda),
                           Profundidad_Auxiliar (Nodo.Derecha));
      end if;
    end Profundidad_Auxiliar;
  begin
    return Profundidad_Auxiliar (Arbol.Raiz);
  end Profundidad;
```

Ejercicio de Vaciado de un árbol:

Es un procedimiento que vacía el árbol de forma recursiva. Para ello realizamos un recorrido en post-order.

```
procedure Vaciar (Arbol : in out T_Arbol) is
  procedure Vaciar (p : in out T_Nodo_Ptr) is
  begin
    if p = null then
      return;
```

```
      end if;
      Vaciar (p.Izq);
      Vaciar (p.Der);
      Free (p);
   end Vaciar;
begin
   Vaciar (Arbol.Raiz);
end Vaciar;
```

Ejercicios propuestos de árboles:

1º) Función Existe que cuando encuentre un elemento escriba el pantalla todo el camino que ha seguido (sólo si lo encuentra).
2º) Espejo de árbol (Dado un árbol poner su inverso).
3º) Escribir en pantalla todos los valores por niveles.
(*nota para este ejercicio es necesario la implementación de una cola)