Apunte 3 - Estructuras Jerárquicas

Índice general

Índice de figuras

Algoritmos

Índice de ejercicios

Apunte 3

Estructuras jerárquicas

Actualizado: 9 de mayo de 2024

En esta unidad trabajaremos sobre la siguiente pregunta:

> *¿Cuáles son las estructuras de datos apropiadas para modelar relaciones de jerarquía entre datos y cómo se pueden implementar?*

A lo largo de la unidad se describen dos tipos de estructuras de datos (árboles genéricos y binarios); de las cuales se presentan los conceptos básicos, aplicaciones y operaciones más importantes de cada una, considerando la perspectiva de Tipo de Dato Abstracto (TDA). También se trabaja sobre su implementación, haciendo en cada caso el correspondiente análisis de eficiencia respecto al uso de memoria y el tiempo de ejecución de las distintas operaciones.

3.1. Definición de árboles

Los árboles son contenedores que permiten organizar un conjunto de datos en forma jerárquica. Se los usa comúnmente para:

- Organizar relaciones genealógicas de un ser vivo con respecto a sus ancestros y sus descendientes.
- Mostrar relaciones evolutivas entre las especies.
- Diagramar la organización de empresas (organigramas).
- Mantener la estructura de archivos de una computadora (jerarquía de carpetas o directorios).
- Representar fórmulas o expresiones matemáticas.

Un árbol se puede definir como un conjunto finito de elementos, llamados *nodos*, donde se distingue un nodo especial llamado *raíz*, que provee el acceso a la estructura. A diferencia de las estructuras lineales vistas en el capítulo anterior, en un árbol los nodos poseen una *relación de paternidad* que impone la estructura jerárquica, donde cada nodo (excepto la raíz) está relacionado con un nodo padre y cero o más nodos hijos. Los nodos que no poseen nodos hijos se denominan *hojas*. Cada nodo del árbol tiene un único nodo *padre* o *predecesor*.

Un árbol también se puede definir de manera recursiva de la siguiente manera:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es la raíz del árbol y también es una hoja.
2. Dado un nodo n y sean A1, A2, ..., Ak árboles con raíces n1, n2, ..., nk respectivamente, se puede construir un nuevo árbol con raíz n y n1, n2, ..., n3 como sus nodos hijos. El árbol tiene a n como raíz y A1, A2, ..., Ak como subárboles de la raíz. Los nodos n1, n2, ..., nk reciben el nombre de hijos del nodo n.

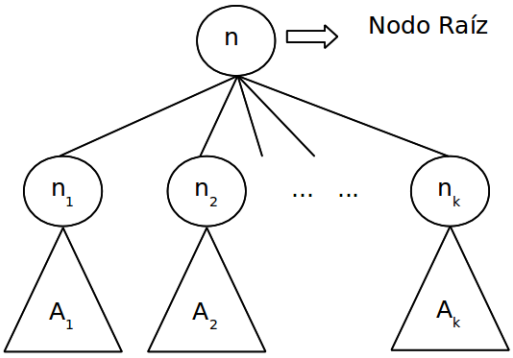

Figura 3.1: Definición recursiva de árbol

Existe una terminología especial que se utilizará en adelante para tratar con las estructuras de tipo árbol. En la Figura 3.2 se muestra un árbol y la aplicación de cada término, tal como se explica a continuación:

- Cada nodo contiene un elemento. Ejemplo: el nodo de la raíz almacena el elemento A.
- Cada nodo tiene un grado, que es el número de hijos (subárboles) que posee. Ejemplos: El nodo C tiene grado 2, el nodo D tiene grado 1 y el nodo E tiene grado 0.
- Un nodo de grado cero se lo denomina nodo *hoja* o *terminal*. Ejemplo: G es una hoja.
- Un nodo que no es hoja ni raíz se denomina nodo *interior*. Ejemplo: B y C son nodos interiores.
- Los nodos hijos de un mismo padre se llaman (entre sí) *hermanos*. Ejemplo: D y E son hermanos. B es padre de D y E.
- Cada nodo tiene asignado un número de *nivel*: La raíz está en el nivel 0. El nivel de cada nodo es uno más que el nivel de su padre. Ejemplo: Los nodos H, I y J están en el nivel 3.
- Un *camino* de un nodo M a un nodo L es el conjunto de nodos por los que hay que pasar para llegar de M a L. Ejemplos: Camino de A a J: {A, C, F, J}, Camino de A a D: {A, B, D}, Camino de C a J: {C, F, J}.
- Un *camino* sólo se forma bajando de padres a hijos, por lo tanto no es posible encontrar un camino entre elementos en subárboles distintos. Ejemplos: no existe camino entre E y F, o entre D y E.
- Un nodo L es *descendiente* de un nodo M si existe un camino de M a L. Ejemplos: D es descendiente de B y de A J es descendiente de A, C y de F C es descendiente de A.
- Si un nodo L es *descendiente* de un nodo M, entonces M es un *ancestro* de L. Ejemplos: B es ancestro de D, E, H A es ancestro de todos los nodos del árbol.
- La longitud de un camino es la cantidad de nodos del camino menos uno (o la cantidad de enlaces por la que hay que pasar). Ejemplos: longitud del cambio desde A a J = 3; longitud del camino desde A a D = 2; longitud del camino desde F a J = 1.
- La *altura de un árbol* es la longitud del camino más largo de la raíz a una hoja. Ejemplo: La altura del árbol en la Figura 3.2 es 3.
- La *altura de un nodo* es la longitud del camino más largo desde el nodo a una hoja. Ejemplos: Altura del nodo D: 1, Altura del nodo H: 0, Altura del nodo B: 2.
- La *profundidad* de un nodo es la longitud del camino que va desde la raíz del árbol hasta ese nodo. Ejemplos: Profundidad del nodo E: 2, Profundidad del nodo F: 2, Profundidad del nodo B: 1.

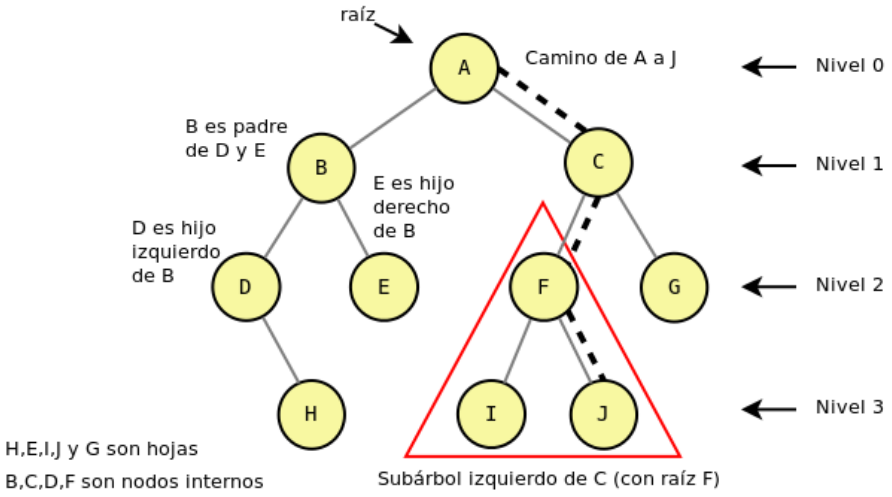

Figura 3.2: Terminología de árboles

3.2. Árboles binarios

Un árbol binario tiene la particularidad de que cada nodo puede tener como máximo dos hijos, que se distinguen como hijo *izquierdo* e hijo *derecho*.

Dos árboles pueden compararse entre sí, de acuerdo a la siguiente definición:

- *Distintos*: dos árboles binarios son distintos cuando tienen estructura diferente. Por ejemplo, en la Figura 3.3, el árbol A es distinto de todos los otros pues su estructura es diferente.

- *Similares*: dos árboles binarios son similares cuando tienen idéntica estructura pero la información contenida en sus nodos difiere entre sí. Por ejemplo, en la Figura 3.3, el árbol B es similar a C (misma estructura, distinto contenido).

- *Equivalentes*: dos árboles binarios son equivalentes si son similares y además coinciden en la información contenida en sus nodos. Por ejemplo, en la Figura 3.3, los árboles B y D son equivalentes (misma estructura y mismo contenido).

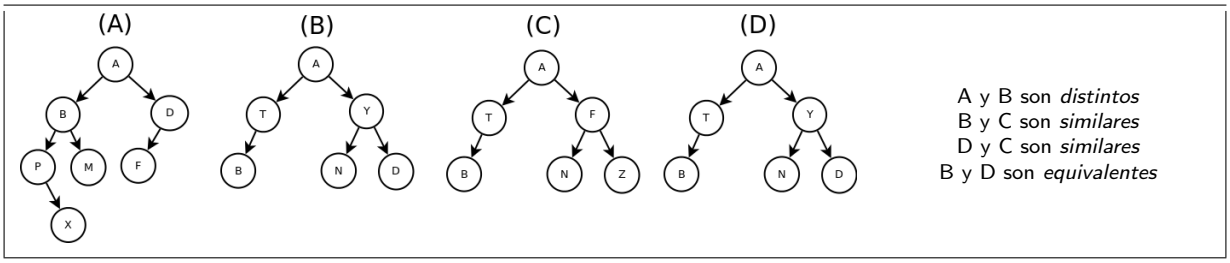

Figura 3.3: Comparación de árboles binarios (similares, equivalentes y distintos)

- *Árbol Completo*: es un árbol en el que todos los nodos tienen 2 hijos, excepto los del último nivel que son todos hojas. En los árboles completos el número de nodos se calcula de la siguiente manera: $2^{niv} - 1$, donde *niv* es la cantidad de niveles del árbol. Por ejemplo, sea un árbol de 4 niveles (como el de la Figura 3.4), aplicamos la fórmula $2^{niv} - 1 = 2^4 - 1 = 16 - 1 = 15$ nodos. Otra propiedad del árbol completo es que la cantidad de nodos por nivel es 2^i, donde i es el nivel dado. Por ejemplo, como se puede ver en la Figura 3.4, en el nivel 0 hay $2^0 = 1$ nodo y en el nivel 3 la cantidad de nodos es $2^3 = 8$.

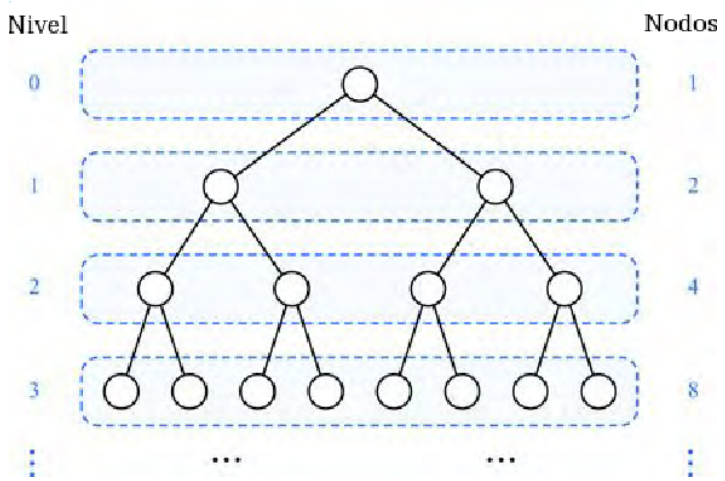

Figura 3.4: Árbol binario completo: Relación entre niveles y cantidad de nodos

3.2.1. Recorridos de árboles binarios

Un recorrido en un árbol A es una forma sistemática de acceder o *visitar* todos los nodos de A una sola vez cada uno.

- *Recorrido en Preorden*: En un recorrido en preorden de un árbol A, la raíz de A es visitada primero y luego los subárboles izquierdo y derecho son recorridos recursivamente con el mismo criterio: se lista la raíz y luego sus subárboles, hasta llegar a un subárbol vacío. Podemos sintetizar el recorrido en preorden de la siguiente manera:

Algoritmo 3.1 Algoritmo de recorrido en preorden de árbol binario

```
Algoritmo preorden(nodo)
 1. visitar la raíz del subárbol (nodo)
 2. recorrer hijo_izquierdo(nodo) en Preorden
 3. recorrer hijo_derecho(nodo) en Preorden
Fin algoritmo
```

- *Recorrido en Inorden*: En un recorrido en inorden de un árbol A, primero se visitan los elementos del subárbol izquierdo de la raíz, luego se visita la raíz de A y por último los elementos del subárbol derecho. La forma de listar cada subárbol es en inorden, es decir que en cada nodo se respeta el orden descripto anteriormente. Podemos sintetizar el recorrido en inorden de la siguiente manera:

Algoritmo 3.2 Algoritmo de recorrido en inorden de árbol binario

```
Algoritmo inorden(nodo)
 1. recorrer hijo_izquierdo(nodo) en inorden
 2. visitar la raíz del subárbol (nodo)
 3. recorrer hijo_derecho(nodo) en inorden
Fin algoritmo
```

- *Recorrido en Posorden*: En un recorrido en posorden de un árbol A, primero se visitan los elementos del subárbol izquierdo y derecho de la raíz y luego la raíz de A. La forma de listar cada subárbol es en posorden, es decir que en cada nodo se respeta el orden descripto anteriormente, hasta llegar a un subárbol vacío. Podemos sintetizar el recorrido en posorden de la siguiente manera:

Algoritmo 3.3 Algoritmo de recorrido en posorden de árbol binario

```
Algoritmo posorden(nodo)
 1. recorrer hijo_izquierdo(nodo) en posorden
 2. recorrer hijo_derecho(nodo) en posorden
 3. visitar la raíz del subárbol (nodo N)
Fin algoritmo
```

- *Recorrido por niveles*: En el recorrido por niveles el orden de visita de los nodos es de acuerdo al nivel del mismo, comenzando con los nodos del nivel cero, incrementando el nivel en uno, hasta llegar al nivel máximo del árbol. Los nodos del mismo nivel se listan de izquierda a derecha. A diferencia de los recorridos en preorden, inorden y posorden que son recursivos, el recorrido por niveles es iterativo y utiliza una cola (TDA Cola visto en la unidad anterior) de un espacio proporcional al número máximo de nodos en una profundidad dada. A continuación se presenta el algoritmo:

Algoritmo 3.4 Algoritmo de recorrido por nivel de árbol binario

```
Algoritmo PorNivel()
 1. Q = nueva cola
 2. poner el nodo raíz en Q
 3. Mientras Q no esté vacía hacer
    • nodoActual= obtener el frente de Q
    • sacar el frente de Q
    • visitar(nodoActual)
    • si hijo_izquierdo(nodoActual) no vacío entonces
      Q.poner(hijo_izquierdo(nodoActual))
    • si hijo_derecho(nodoActual) no vacío entonces
      Q.poner(hijo_derecho(nodoActual))
   Fin mientras
Fin algoritmo
```

En la Figura 3.5 se muestra el resultado de los distintos recorrido vistos anteriormente en un mismo árbol binario.

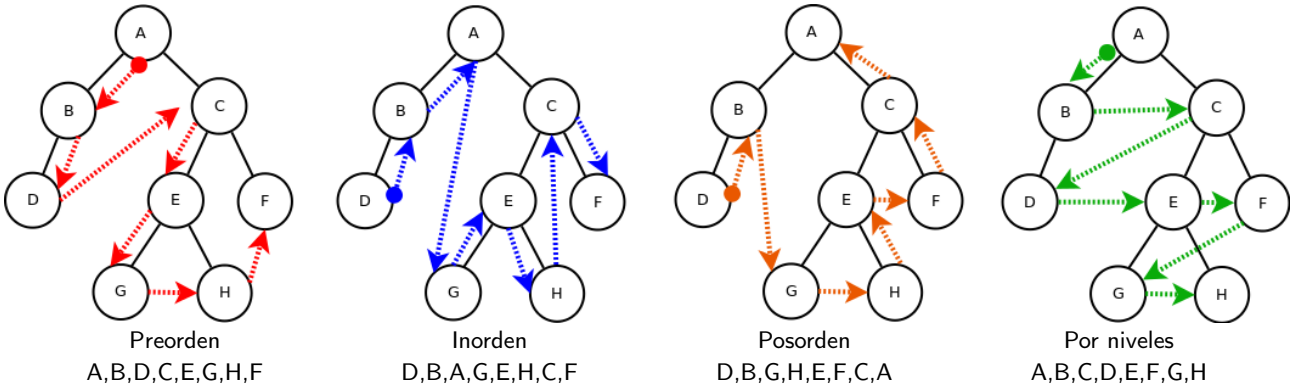

Figura 3.5: Ejemplo de recorridos en un árbol binario

3.2.2. Operaciones del TDA árbol binario

Las operaciones básicas de esta estructura son:

- *constructor vacío*:
  // Crea un árbol binario vacío

- *insertar*(elemNuevo, elemPadre, posHijo) → boolean
  // Dado un elemento *elemNuevo* y un elemento *elemPadre*, inserta elemNuevo como hijo izquierdo o

derecho de la primer aparición de *elemPadre*, según lo indique el parámetro *posHijo*. Para que la operación termine con éxito debe existir un nodo en el árbol con elemento = elemPadre y ese nodo debe tener libre su hijo *posHijo*. Si puede realizar la inserción devuelve *verdadero*, en caso contrario devuelve *falso*.

- *insertarPorPosicion*(elemNuevo, posPadre, posHijo): boolean
  // Dado un elemento *elemNuevo* y la posición numérica de su padre en el árbol en preorden, agrega *elemNuevo* como hijo *izquierdo o derecho* del elemento cuya posición en preorden dentro del árbol sea la dada, y según lo indique el parámetro *posHijo*. Para que la operación termine con éxito debe existir un nodo en el árbol cuya posición en preorden sea válida y ese nodo debe tener libre su hijo *posHijo*. Esta operación devuelve *verdadero* cuando se pudo agregar *elemNuevo* a la estructura y *falso* en caso contrario.

- *esVacio*(): boolean
  // Devuelve *falso* si hay al menos un elemento cargado en el árbol y *verdadero* en caso contrario.

- *altura*(): int
  // Devuelve la altura del árbol, es decir la longitud del camino más largo desde la raíz hasta una hoja (Nota: un árbol vacío tiene altura -1 y una hoja tiene altura 0).

- *nivel*(elemento): int
  // Devuelve el nivel de un *elemento* en el árbol. Si el elemento no existe en el árbol devuelve -1.

- *padre*(elemento): elemPadre
  // Dado un elemento devuelve el valor almacenado en su nodo padre (busca la primera aparición de *elemento*).

- *listarPreorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol binario en el recorrido en preorden

- *listarInorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol binario en el recorrido en inorden

- *listarPosorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol binario en el recorrido en posorden

- *listarNiveles*(): lista de elementos
  // Devuelve una lista con los elementos del árbol binario en el recorrido por niveles

- *clone*(): ArbolBinario
  // Genera y devuelve un árbol binario que es equivalente (igual estructura y contenido de los nodos) que el árbol original.

- *vaciar*() : void
  // Quita todos los elementos de la estructura. *El manejo de memoria es similar al explicado anteriormente para estructuras lineales dinámicas.*

- *toString*(): String
  // Genera y devuelve una cadena de caracteres que indica cuál es la raíz del árbol y quienes son los hijos de cada nodo.

3.2.3. Implementaciones

Implementación estática

Se puede implementar un árbol binario de forma estática utilizando arreglos. Para ello, cada celda del arreglo almacenará un nodo con la siguiente información:

- El elemento almacenado en el nodo,
- Dos índices que indican las posiciones del arreglo en donde se encuentran su hijo izquierdo y derecho,
- Un valor booleano que indica si esa celda del arreglo está en uso o no.

El árbol estará formado por el índice donde se encuentra la raíz y el arreglo de nodos o celdas. Si el árbol está vacío, la posición de la raíz será un valor negativo, como hemos hecho en las implementaciones estáticas de Pila y Cola.

En la Figura 3.6 se muestra un ejemplo de la representación del árbol de la izquierda de manera estática y en la Figura 3.7 se presenta el diagrama de clases UML para esta implementación.

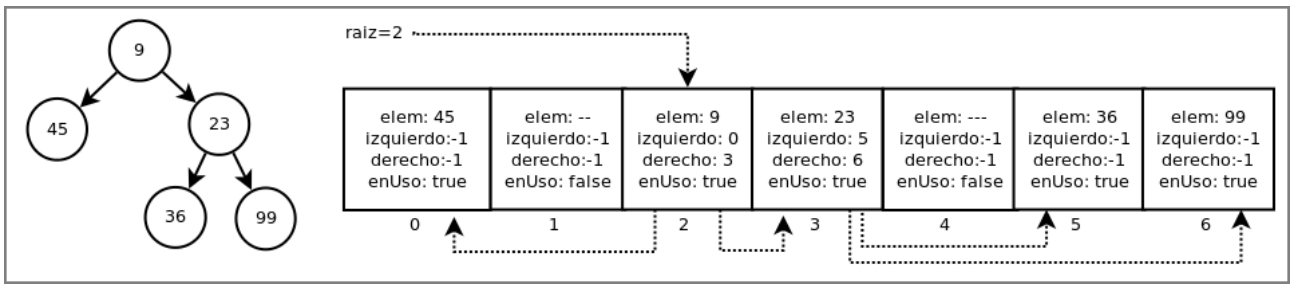

Figura 3.6: Ejemplo de un árbol binario implementado de manera estática

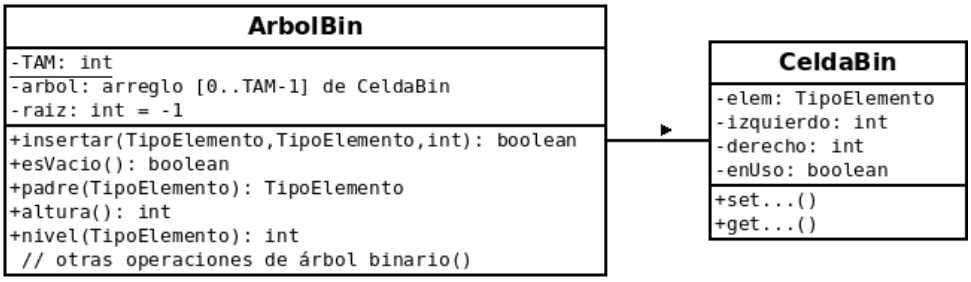

Figura 3.7: Diagrama UML de clases para implementar árbol binario en forma estática

A continuación se muestra el código Java de la implementación del TDA árbol binario para almacenar elementos de tipo entero, de acuerdo a la definición del UML anterior.

Algoritmo 3.5 Código Java de clase ArbolBin y su constructor (implementación estática)

```
public class ArbolBin {

    // atributos
    private static final int TAM = 20;
    private CeldaBin[] arbol;
    private int raiz;

    // constructor vacio
    public ArbolBin(){
        this.arbol = new CeldaBin[TAM];
        for (int i=0; i<TAM; i++){
            this.arbol[i] = new CeldaBin();
        }
        this.raiz = -1;
    }


}
```

Como se explicó para las implementaciones estáticas de Pila y Cola, la implementación estática no es la más recomendada porque el espacio asignado es fijo y predefinido. Además, para evitar los errores de estructura llena, se suele definir un arreglo de mayor dimensión que la necesaria, y eso genera que se malgaste espacio de memoria. La implementación dinámica mejora ambos aspectos, como se describe a continuación.

Implementación dinámica

Los árboles binarios también pueden implementarse de forma dinámica definiendo nodos individuales, donde cada nodo almacena:

- El elemento
- Una referencia a cada hijo (izquierdo - derecho)

El arbol está formado por la referencia al nodo raíz. Un árbol vacío es aquel cuya *raíz* está seteada a `null`. En la Figura 3.8 se muestra el diagrama de clases para implementar árbol binario de manera dinámica. Notar que la clase NodoArbol, de manera similiar a Nodo en las estructuras lineales, utiliza las referencias a los objetos para implementar los enlaces entre el nodo padre con sus hijos izquierdo y derecho.

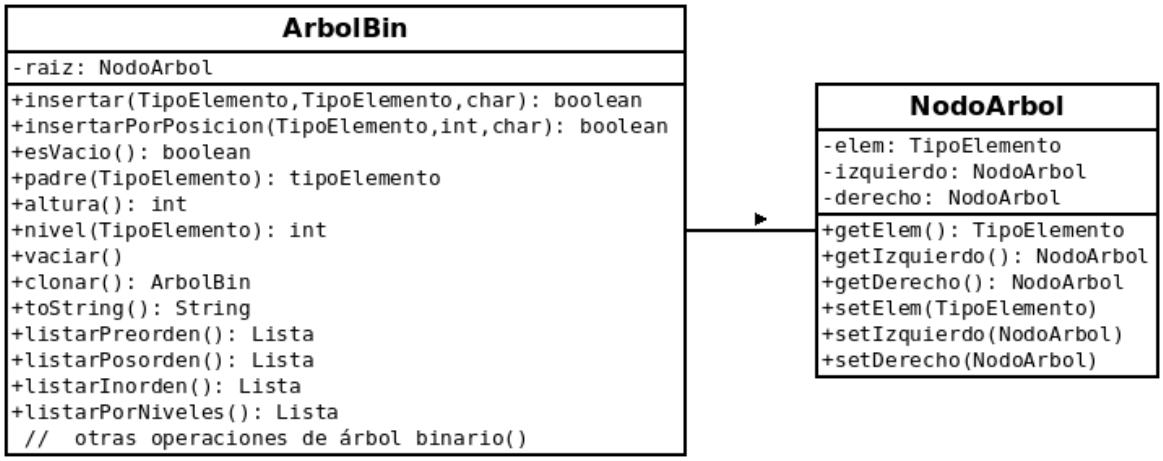

Figura 3.8: Diagrama UML de clases para implementar árbol binario en forma dinámica

A continuación se muestra la implementación del método constructor para árbol binario para almacenar elementos de tipo Object, de acuerdo a la especificación UML anterior.

Algoritmo 3.6 Código Java de clase ArbolBin y su constructor (implementación dinámica)

```
public class ArbolBin {

    // atributos
    private NodoArbol raiz;

    // constructor
    public ArbolBin() {
        this.raiz = null;
    }
```

A continuación se muestra la implementación del método que devuelve los elementos del árbol binario en preorden. El tipo de visita que realiza el método es poner el elemento en la lista. Notar que el método público sólo se encarga de llamar al método interno o auxiliar *listarPreordenAux* que es el que realiza el recorrido recursivo. Este método se define "private" porque necesita recibir por parámetro un nodo que es de tipo NodoArbol, y esto es propio de la implementación dinámica, por lo tanto al ser privado se mantiene oculta la implementación a los usuarios de la clase.

Algoritmo 3.7 Código Java de árbol binario para listarlo en preorden

```
public Lista listarPreorden() {
    // retorna una lista con los elementos del árbol en PREORDEN
    Lista lis = new Lista();
    listarPreordenAux(this.raiz, lis);
    return lis;
}

private void listarPreordenAux(NodoArbol nodo, Lista lis) {
    // método recursivo PRIVADO porque su parametro es de tipo NodoArbol

    if (nodo != null) {
        // visita el elemento en el nodo
        lis.insertar(nodo.getElem(),lis.longitud()+1); // (1)

        // recorre a sus hijos en preorden
        listarPreordenAux(nodo.getIzquierdo(),lis);   // (2)
        listarPreordenAux(nodo.getDerecho(),lis);     // (3)
    }
}
```

Otro método que analizaremos a continuación es *insertar*, que debe agregar un nuevo elemento indicándole su *padre* y si quiere que sea su hijo izquierdo o derecho. Para indicar dicho *lugar* se ha elegido un char que se debe tener el valor 'I' o 'D'. Para facilitar la programación, se ha implementado un método auxiliar *obtenerNodo* que es el que realiza la búsqueda de manera recursiva y retorna el nodo donde se ha encontrado el elemento. En caso que no lo encuentre devuelve null. Este método *debe ser siempre privado*, porque tiene como resultado el acceso directo a un nodo, es decir, a la estructura interna del árbol binario.

Algoritmo 3.8 Código Java de método *insertar* en árbol binario (implementación dinámica)

```
/**
 * Inserta elemNuevo como hijo del primer nodo encontrado en preorden igual
 * a elemPadre, como hijo izquierdo (I) o derecho (D), segun lo indique el
 * parametro lugar
 */
public boolean insertar(Object elemNuevo, Object elemPadre, char lugar) {

    boolean exito = true;

    if (this.raiz == null) {
        // si el arbol esta vacio, pone elem nuevo en la raiz
        this.raiz = new NodoArbol(elemNuevo, null, null);
    } else {
        // si arbol no esta vacio, busca al padre
        NodoArbol nPadre = obtenerNodo(this.raiz, elemPadre);

        // si padre existe y lugar no está ocupado lo pone, sino da error
        if (nPadre != null) {
            if (lugar == 'I' && nPadre.getIzquierdo() == null) {
                nPadre.setIzquierdo(new NodoArbol(elemNuevo, null, null));
            } else if (lugar == 'D' && nPadre.getDerecho() == null) {
                nPadre.setDerecho(new NodoArbol(elemNuevo, null, null));
            } else {
                exito = false;
            }
        } else {
            exito = false;
        }
    }
    return exito;
}
```

Algoritmo 3.9 Código Java del método privado *obtenerNodo* en árbol binario (implementación dinámica)

```
private NodoArbol obtenerNodo(NodoArbol n, Object buscado) {
    // metodo PRIVADO que busca un elemento y devuelve el nodo que
    // lo contiene. Si no se encuentra buscado devuelve null

    NodoArbol resultado = null;
    if (n != null) {
        if (n.getElem().equals(buscado)) {
            // si el buscado es n, lo devuelve
            resultado = n;
        } else {
            // no es el buscado: busca primero en el HI
            resultado = obtenerNodo(n.getIzquierdo(), buscado);
            // si no lo encontro en el HI, busca en HD
            if (resultado == null) {
                resultado = obtenerNodo(n.getDerecho(), buscado);
            }
        }
    }
    return resultado;
}
```

3.2.4. Análisis de eficiencia

Los elementos del árbol binario no poseen orden entre ellos, por lo que al momento de ubicar un determinado elemento se debe recorrer toda la estructura hasta encontrar al elemento buscado o hasta que no queden más elementos por recorrer. Por este motivo, la mayoría de las operaciones que nos interesan de árbol binario son de O(n), pues en el peor de los casos se deberá recorrer todos los elementos de la estructura para llevar a cabo la operación. Por ejemplo, en la operación *insertar* el elemento que necesitamos ubicar para operar es el padre, y en la operación *nivel* es el elemento del cual se desea conocer dicha información.

Ejercicio 3.1: Implementación del TDA Árbol Binario de manera dinámica

1. Crear un paquete *jerarquicas* e implementar la clase ArbolBin de manera dinámica, incluyendo todas las operaciones vistas para el TDA árbol binario.
2. Agregar al árbol binario la operación *frontera*() que devuelve una lista con todos los elementos almacenados en las hojas del árbol listadas de izquierda a derecha.
3. Agregar al árbol binario la operación *obtenerAncestros*(elem) que devuelve en una lista todos los ancestros del elemento pasado por parámetro (si el elemento no está, devuelve la lista vacía)
4. Agregar al árbol binario la operación *obtenerDescendientes*(elem) que devuelve en una lista todos los descendientes del elemento pasado por parámetro (si el elemento no está, devuelve la lista vacía)
5. Crear un paquete *tests.jerarquicas* e implementar la clase TestArbolBin que permita probar todas las operaciones de la clase ArbolBin anterior.

3.2.5. Ejemplos de uso de árboles binarios

Árbol genealógico

Un uso posible de árbol binario es para representar un árbol genealógico, donde la raíz del árbol contiene los datos de la persona de la cual se desea armar su árbol genealógico y sus descendientes indican la relación con sus padres, abuelos, bisabuelos, etc. En la Figura 3.9 se muestra un ejemplo de un árbol genealógico para una persona, de la manera que se suele representar visualmente. A la derecha se observa cómo se almacenarían los elementos en el árbol binario.

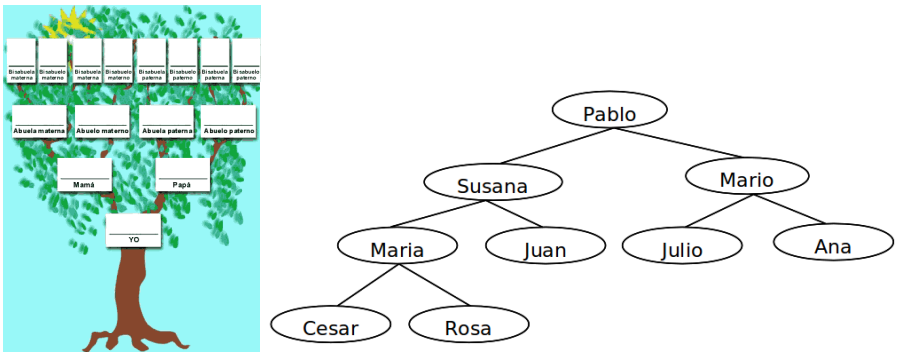

Figura 3.9: Uso de árbol binario para representar un árbol genealógico

Árbol de decisión

Otro uso de los árboles binarios es para implementar árboles de decisión, que son un modelo de predicción utilizado en el ámbito de la inteligencia artificial. Dada una base de datos se crean diagramas de construcciones lógicas que sirven para representar y categorizar una serie de condiciones que ocurren de forma sucesiva, para la resolución de un problema. Por ejemplo, en la Figura 3.10 se muestra un árbol de decisión para otorgar un crédito a un cliente de un banco.

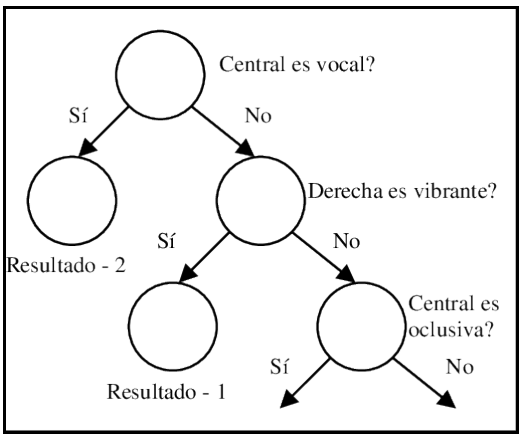

Figura 3.10: Uso de árbol binario para crear árboles de decisión

Árbol de Huffman

Un uso muy interesante de árbol binario es para la creación de los códigos de Huffman, que es un algoritmo para compresión de datos. El algoritmo consiste en la creación de un árbol binario que tiene símbolos en sus hojas, y está construido de tal forma que, bajando desde la raíz hacia una hoja, se obtiene el código Huffman asociado al símbolo almacenado en dicha hoja. Con este árbol se puede conocer el código asociado a un símbolo, así como obtener el símbolo asociado a un determinado código. El árbol se arma a partir de una tabla que ha sido rellenada de una manera específica, basándose en la probabilidad estimada de aparición de cada símbolo. Con esto se logra que los símbolos que aparecen más veces tengan una cadena de representación más corta, logrando buen rendimiento en la compresión. Para la codificación se considera que el hijo izquierdo siempre representa al valor 0 (cero) y el hijo derecho al 1 (uno). En el ejemplo de la Figura 3.11 se puede ver que el código asignado al símbolo B (que es el más probable, con 0,30) es el código más corto "00", luego los códigos asignado a E y A son "011" y "010" que son los que le siguen en probabilidad, etc. Algo muy importante es que, al construir los códigos utilizando el árbol binario, ningún código es prefijo de otro, por lo que no es necesario usar separadores de símbolos. Por ejemplo, a partir del árbol de la figura 3.11 se puede descifrar que la cadena 1110010110010 significa DAGA.

Símbolo	*Frecuencia*
A	0,15
B	0,30
C	0,20
D	0,05
E	0,15
F	0,05
G	0,10

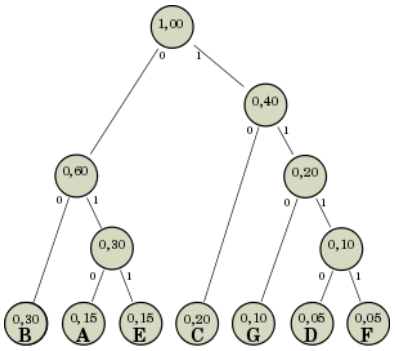

Figura 3.11: Uso de árboles binarios para crear códigos para compresión de datos (Código Huffman)

Árbol de Expresión

Los árboles binarios son también útiles en la implementación de compiladores para lenguajes de programación. Estos árboles, llamados *árboles de expresión* almacenan ecuaciones que pueden ser evaluadas de manera más rápida que si estuvieran almacenados como una cadena y sin necesidad de almacenar los paréntesis. En el árbol de expresión cada nodo interno es un operador binario matemático o lógico (+,-,*,/, %,>,=,etc) y cada hoja es un operando (una variable o un valor literal). Por ejemplo, en el árbol de la Figura 3.12 está representada la ecuación (((10 + 1) * 3) - 20) + (3 * 2).

Algo importante es que los recorridos recursivos vistos para árboles binarios permiten obtener rápidamente distintos tipos de notación de la misma ecuación. Por ejemplo, el recorrido en inorden devuelve la notación infija, que es la que los humanos comprendemos rápidamente, pero que necesita los paréntesis para que sea correcta. Por otro lado, el recorrido posorden devuelve la notación posfija, que es ideal para evaluar la expresión en un sólo recorrido, teniendo los hijos calculados antes de operar con el padre. El recorrido en preorden devuelve la notación prefija.

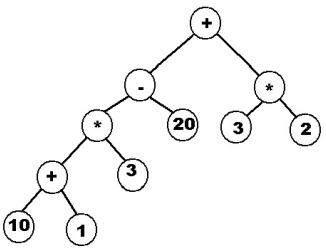

Figura 3.12: Uso de árboles binarios para crear árboles de expresión

3.3. Árboles genéricos

Como se explicó en la sección anterior, en un árbol binario cada nodo puede tener como máximo 2 hijos. Además, cada uno de los hijos se puede reconocer como hijo izquierdo o derecho de su padre. Sin embargo, en la práctica son comunes los ejemplos donde no se sabe a priori la cantidad de hijos que un nodo puede tener o se desea que pueda tener más de 2 hijos.

Un uso extendido de los árboles genéricos es para programar juegos (área de inteligencia artificial). En estas aplicaciones, cada estado del juego se modela como un nodo del árbol y los estados alcanzables a partir de él en un único movimiento se guardan como sus hijos (es decir, un nodo tiene un hijo por cada estado siguiente posible). De esta manera, un camino desde la raíz hasta una hoja del árbol representa una posible partida del juego.

En el ejemplo de la Figura 3.13 se presenta un árbol para el juego del "3 en raya" (o TA-TE-TI) donde el estado inicial es el tablero vacío. Luego, la ficha "X" que utiliza el jugador 1 se puede ubicar en cualquiera de las 9 celdas, por lo tanto el estado inicial (raíz) tiene 9 hijos o *estados siguientes*. A continuación, el jugador 2 (que usa la ficha "O") puede ubicarla en una de las 8 celdas desocupadas, y así sucesivamente.

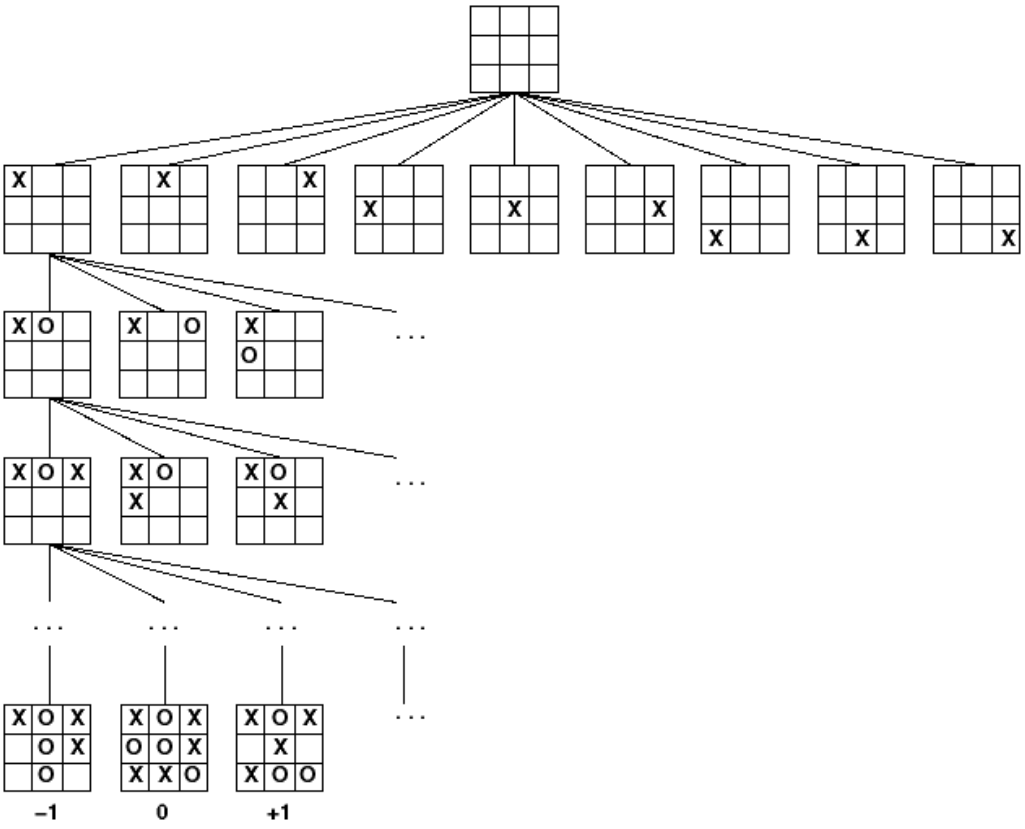

Figura 3.13: Ejemplo de árbol genérico utilizado en Teoría de Juegos

En el ejemplo del árbol del "3 en raya", en cada nivel hay una cantidad máxima distinta de hijos. Aquellos nodos donde alguno de los jugadores gana o cuando las fichas quedan bloqueadas y no se puede seguir jugando (empate), son los nodos terminales u hojas del árbol. En el ejemplo, se utilizan los valores -1, 0 y 1 para indicar el resultado del juego (1: gana el jugador 1; 0: empate; -1: gana el jugador 2).

Este tipo de árbol, llamado *árbol genérico* o *n-ario*, respeta la definición presentada en la Sección 3.1, y permite que cada nodo pueda tener N nodos hijos.

A continuación se repasa la terminología de árboles presentada en la Sección 3.1, en un ejemplo mostrado en la Figura 3.14.

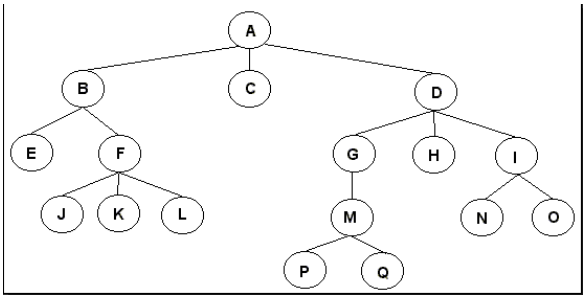

Figura 3.14: Ejemplo de árbol genérico

- Raíz del árbol: Es el nodo que almacena el elemento A.
- Grado del nodo (cantidad de hijos del nodo). Grado de D es 3.
- Nodos hojas: E, J, K, L, C, P, Q, H, N, O.
- Nodos interiores: B, F, D, G, M, I.
- Nodos hermanos: B, C y D son hermanos con padre A; E y F son hermanos con padre B.
- Nivel: los nodos P y Q están en el nivel 4.
- F es descendiente de A y de B.
- B es ancestro de L.
- Un *camino* se forma bajando de padres a hijos. Ejemplo: camino de B a L:{B, F, L}. No existe camino entre B y M.
- Longitud del camino de A a M = 3.
- Altura del árbol = altura de A = 4. Altura de B = 2; Altura de C = 0
- Profundidad de F = 2. Profundidad de P = 4.

3.3.1. Recorridos de árboles genéricos

- *Recorrido en Preorden*: En un recorrido en preorden de un subárbol T, la raíz de T es visitada primero y luego se recorren los subárboles de sus nodos hijos de izquierda a derecha, en forma recursiva y con el mismo criterio: se lista la raíz y luego sus subárboles, hasta llegar a un subárbol vacío. Podemos sintetizar el recorrido en preorden de la siguiente manera:

Algoritmo 3.10 Algoritmo de recorrido en preorden de árbol genérico

```
Algoritmo preorden(nodo)
  ▪ visitar la raíz del subárbol (nodo)
  ▪ Para cada hijo i de nodo
      • recorrer hijo i en Preorden
Fin algoritmo
```

- *Recorrido en Inorden*: En un recorrido en inorden de un subárbol T, primero se visita en inorden el subárbol izquierdo de la raíz (el hijo más izquierdo o primer hijo de la raíz), luego se visita la raíz de T y por último se recorren en inorden los restantes hijos de T (los que se encuentran a la derecha del primer hijo). La forma de listar cada subárbol es en inorden, es decir que en cada nodo se respeta el orden descripto anteriormente. Se puede sintetizar el recorrido en inorden de la siguiente manera:

Algoritmo 3.11 Algoritmo de recorrido en inorden de árbol genérico

```
Algoritmo inorden(nodo)
  ▪ recorrer en inorden el primer hijo de nodo
  ▪ visitar la raíz del subárbol (nodo)
  ▪ Para cada hijo i de nodo (donde i >= 2)
      • recorrer hijo i en inorden
Fin algoritmo
```

- *Recorrido en posorden*: En un recorrido en posorden de un árbol T, primero se listan en posorden los subárboles de la raíz y luego se visita la raíz de T. Los subárboles se recorren de izquierda a derecha y cada uno de ellos se lista en posorden, es decir que en cada nodo se respeta el orden descripto anteriormente, hasta llegar a un subárbol vacío. Se puede sintetizar el recorrido en posorden de la siguiente manera:

Algoritmo 3.12 Algoritmo de recorrido en posorden de árbol genérico

```
Algoritmo posorden(nodo)
  ▪ Para cada hijo i de nodo
      • recorrer hijo i en posorden
  ▪ visitar la raíz del subárbol (nodo)
Fin algoritmo
```

- *Recorrido por niveles*: En el recorrido por niveles el orden de visita de los nodos es de acuerdo al nivel del mismo, comenzando con los nodos del nivel cero, incrementando el nivel en uno, hasta llegar al nivel máximo del árbol. Los nodos del mismo nivel se listan de izquierda a derecha. A diferencia de los recorridos en preorden, inorden y posorden que son recursivos, el recorrido por niveles es iterativo y utiliza una cola (TDA Cola visto en la unidad anterior) de un espacio proporcional al número máximo de nodos en una profundidad dada.

  IMPORTANTE: El tipo de elemento que se guarda en la Cola es el nodo del árbol (no sólo su elemento) así es posible saber quienes son los hijos del nodo que se recupera de la Cola. A continuación se presenta el algoritmo:

Algoritmo 3.13 Algoritmo de recorrido por nivel de árbol genérico

```
Algoritmo PorNivel()
 1. Cola Q = nueva cola
 2. poner en Q el nodo raíz
 3. mientras not cola.vacía hacer
      ▪ nodo = obtener el frente de Q
      ▪ sacar el frente de Q
      ▪ visitar(nodo)
      ▪ Para cada hijo i de nodo
          • poner el hijo i de nodo en la cola Q
    fin mientras
Fin algoritmo
```

3.3.2. Operaciones de TDA árbol genérico

Las operaciones básicas de esta estructura son:

- *constructor vacío*:
  // Crea un árbol genérico vacío

- *insertar*(elemNuevo, elemPadre): boolean
  // Dado un elemento *elemNuevo* y un elemento *elemPadre*, agrega *elemNuevo* como hijo de la primer aparición de *elemPadre*. Para que la operación termine con éxito debe existir un nodo en el árbol con elemento = *elemPadre*. No se establece ninguna preferencia respecto a la posición del hijo respecto a sus posibles hermanos. Esta operación devuelve *verdadero* cuando se pudo agregar *elemNuevo* a la estructura y *falso* en caso contrario.

- *insertarPorPosicion*(elemNuevo, posPadre): boolean
  // Dado un elemento *elemNuevo* y la posición de su padre en el árbol en preorden, agrega *elemNuevo* como hijo del elemento cuya posición en preorden dentro del árbol sea la dada. Para que la operación termine con éxito debe existir un nodo en el árbol cuya posición en preorden sea válida. No se establece ninguna preferencia respecto a la posición del hijo respecto a sus posibles hermanos. Esta operación devuelve *verdadero* cuando se pudo agregar *elemNuevo* a la estructura y *falso* en caso contrario.

- *pertenece*(elemento): boolean
  // Devuelve *verdadero* si el elemento pasado por parámetro está en el árbol, y *falso* en caso contrario.

- *ancestros*(elemento): lista de elementos
  // Si el elemento se encuentra en el árbol, devuelve una lista con el camino desde la raíz hasta dicho elemento (es decir, con los ancestros del elemento). Si el elemento no está en el árbol devuelve la lista vacía.

- *esVacio*(): boolean
  // Devuelve *falso* si hay al menos un elemento cargado en el árbol y *verdadero* en caso contrario.

- *altura*(): int
  // Devuelve la altura del árbol, es decir la longitud del camino más largo desde la raíz hasta una hoja (Nota: un árbol vacío tiene altura -1 y una hoja tiene altura 0).

- *nivel*(elemento): int
  // Devuelve el nivel de un *elemento* en el árbol. Si el elemento no existe en el árbol devuelve -1.

- *padre*(elemento): elemPadre
  // Dado un elemento devuelve el valor almacenado en su nodo padre (busca la primera aparición de *elemento*).

- *listarPreorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol en el recorrido en preorden

- *listarInorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol en el recorrido en inorden

- *listarPosorden*(): lista de elementos
  // Devuelve una lista con los elementos del árbol en el recorrido en posorden

- *listarNiveles*(): lista de elementos
  // Devuelve una lista con los elementos del árbol en el recorrido por niveles

- *clone*(): ArbolGenerico
  // Genera y devuelve un árbol genérico que es equivalente (igual estructura y contenido de los nodos) que el árbol original.

- *vaciar*() : void
  // Quita todos los elementos de la estructura. *El manejo de memoria es similar al explicado anteriormente para estructuras lineales dinámicas.*

- *toString*(): String
  // Genera y devuelve una cadena de caracteres que indica cuál es la raíz del árbol y quienes son los hijos de cada nodo.

3.3.3. Implementación

Los árboles genéricos pueden implementarse de forma dinámica de manera similar a lo explicado árbol binario. La diferencia es que, para no limitar la cantidad de hijos de cada nodo, el padre se enlaza al primer hijo, el cual será referenciado como el "hijo extremo izquierdo" (o HEI) y los otros hijos se almacenan enlazados

unos a otros (enlace al hermano derecho o HD). En la Figura 3.15 se muestra un ejemplo de árbol genérico y la manera que los nodos quedan enlazados en la implementación dinámica HEI-HD.

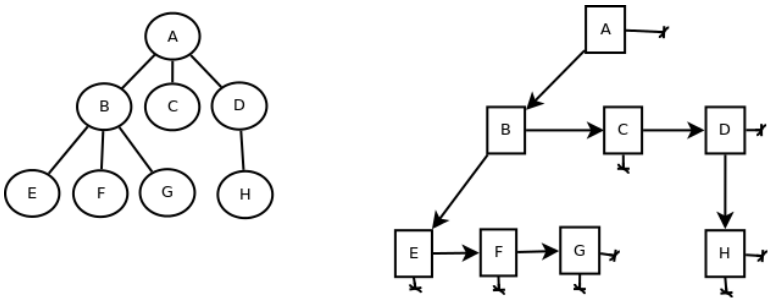

Figura 3.15: Representación de un árbol genérico en implementación HEI-HD

Esta implementación se puede realizar tanto de manera estática como dinámica. A continuación se presenta el diagrama de clases UML para la forma dinámica, mediante una clase que representa al árbol y que almacena la referencia al nodo raíz. Un árbol vacío es aquel cuya *raíz* está seteada a `null`. Luego, se define la clase nodo que almacena el elemento, el enlace al primer hijo o hijo extremo izquierdo y un segundo enlace al hermano derecho del nodo. En la Figura 3.16 se muestra el diagrama de clases para esta implementación.

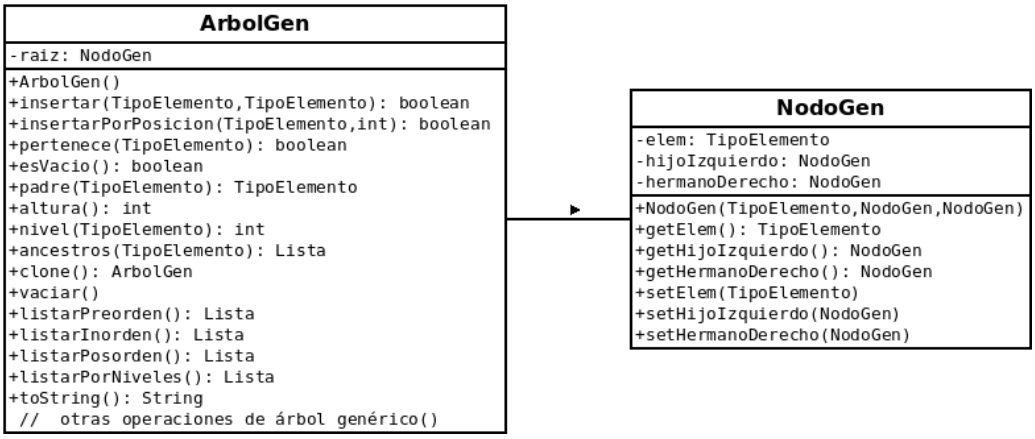

Figura 3.16: Diagrama UML de clases para implementar árbol genérico en forma dinámica

A continuación se muestra la implementación del método constructor para árbol genérico para almacenar elementos de tipo Object (superclase de todas las clases Java), de acuerdo a la especificación UML anterior.

Algoritmo 3.14 Código Java de clase ArbolGen y constructor (implementación dinámica)

```
public class ArbolGen {

    // atributos
    private NodoGen raiz;

    // constructor
    public ArbolGen() {
        this.raiz = null;
    }
```

A continuación se muestra la implementación de un método que pone en una lista los elementos del árbol genérico en preorden, siguiendo el algoritmo 3.10 presentado en la Sección 3.3.1. El tipo de visita que realiza el método es insertar el elemento en la lista (para ello, la lista debe estar definida para elementos de tipo Object, como la vista en la materia en la primera unidad). Notar que el método público *listarPreorden* llama a un método interno *listarPreordenAux* que es el que realiza el recorrido recursivo. Este método se define "private" porque necesita recibir por parámetro un nodo que es de tipo NodoGen para mantener oculta la implementación a los usuarios de la clase ArbolGen. Después de la visita se realiza el llamado en forma recursiva para cada hijo del nodo n.

Algoritmo 3.15 Código Java del método listarPreorden de árbol genérico

```
public Lista listarPreorden() {
    Lista salida = new Lista();
    listarPreordenAux(this.raiz, salida);
    return salida;
}


private void listarPreordenAux(NodoGen n, Lista ls) {
    if (n != null) {
        // visita del nodo n
        ls.insertar(n.getElem(), ls.longitud() + 1);

        // llamados recursivos con los hijos de n
        NodoGen hijo = n.getHijoIzquierdo();
        while (hijo != null) {
            listarPreordenAux(hijo, ls);
            hijo = hijo.getHermanoDerecho();
        }
    }

}
```

A continuación se muestra la implementación de un método que genera un String que permite ver la estructura del árbol indicando quién es hijo de quién. Este método es útil para verificar el funcionamiento de la operación insertar, visualizando cómo quedó cargada la estructura. Notar que en el método privado *toStringAux*, el algoritmo usado es el mismo algoritmo de recorrido en preorden que se usó en el método (listarPreordenAux, pero que en este caso la visita que realiza es el armado de una línea de texto que incluye el toString del elemento almacenado en el nodo que ingresa por parámetro (n) y a continuación pone una flecha >ƨ luego recorre a todos los hijos de n, agregando el toString de cada uno de ellos al String de salida. Una vez realizada esta visita, vuelve a recorrer a los hijos, pero esta vez para realizar la llamada recursiva con cada uno.

Algoritmo 3.16 Código Java del método toString de árbol genérico (usando recorrido en preorden)

```
public String toString() {
    return toStringAux(this.raiz);
}

private String toStringAux(NodoGen n) {
    String s = "";
    if (n != null) {
        // visita del nodo n
        s += n.getElem().toString() + " -> ";
        NodoGen hijo = n.getHijoIzquierdo();
        while (hijo != null) {
            s += hijo.getElem().toString() + ", ";
            hijo = hijo.getHermanoDerecho();
        }

        // comienza recorrido de los hijos de n llamando recursivamente
        // para que cada hijo agregue su subcadena a la general
        hijo = n.getHijoIzquierdo();
        while (hijo != null) {
            s += "\n" + toStringAux(hijo);
            hijo = hijo.getHermanoDerecho();
        }
    }
    return s;
}
```

Este método toString permite visualizar la estructura completa y ver cómo están los elementos almacenados (es decir, hijo de quien es cada nodo, y quién es la raíz, que está listada en la primera posición, dado que el recorrido usado es en preorden). Por ejemplo, el String obtenido para el árbol de la Figura 3.15 sería el siguiente:

A ->B, C, D
B ->E, F, G
E ->
F ->

G ->
C ->
D ->H
H ->

A continuación se muestra la implementación de un método que lista los elementos del árbol genérico en inorden. El algoritmo aplicado es el 3.11, presentado en la Sección 3.3.1. En este caso el tipo de visita que realiza es insertar el elemento en una lista. Obsérvese que el recorrido en inorden primero recorre en inorden el hijo extremo izquierdo de n, luego realiza la visita del nodo n y por último realiza el llamado recursivo para el resto de los hijos de n.

Algoritmo 3.17 Código Java método listarInorden de árbol genérico

```
public Lista listarInorden() {
    Lista salida = new Lista();
    listarInordenAux(this.raiz, salida);
    return salida;
}

private void listarInordenAux(NodoGen n, Lista ls) {
    if (n != null) {
        // llamado recursivo con primer hijo de n
        if (n.getHijoIzquierdo() != null) {
            listarInordenAux(n.getHijoIzquierdo(), ls);
        }

        // visita del nodo n
        ls.insertar(n.getElem(), ls.longitud() + 1);

        // llamados recursivos con los otros hijos de n
        if (n.getHijoIzquierdo() != null) {
            NodoGen hijo = n.getHijoIzquierdo().getHermanoDerecho();
            while (hijo != null) {
                listarInordenAux(hijo, ls);
                hijo = hijo.getHermanoDerecho();
            }
        }
    }
}
```

En la implementación de estos métodos es MUY IMPORTANTE no confundir el recorrido sobre la lista de hijos del nodo n con recorrer la lista de sus hermanos: verificar que en todos los algoritmos propuestos la variable auxiliar *hijo* se ubica sobre el hijo izquierdo de n y luego se mueve sobre la lista de hermanos de dicho nodo; es decir que la sentencia iterativa (while) recorre a los nodos del nivel inferior del nodo n (sus hijos) y no a sus hermanos, que estarían en el mismo nivel que n.

Ejercicio 3.2: Implementación del TDA árbol genérico de manera dinámica (HEI-HD)

1. Agregar al paquete *jerarquicas* la clase NodoGen para elementos de tipo Object (Java).
2. Agregar al paquete *jerarquicas* la clase ArbolGen, incluyendo todas las operaciones del TDA árbol genérico para elementos de tipo Object.
3. En el paquete *tests.jerarquicas* agregar la clase TestArbolGen que permita probar todas las operaciones de la clase ArbolGen implementada en el inciso anterior. Para el test, suponga que el árbol almacena el sistema de directorios o carpetas de un sistema operativo.

3.3.4. Ejemplos de uso de árboles genéricos

Los árboles genéricos son estructuras muy útiles en el modelado de casos reales. En particular, los problemas que esta estructura facilita son aquellos relacionados a encontrar los ancestros o descendientes de un elemento. Un uso habitual es en el área de sistemas operativos, para modelar los sistemas de archivos o carpetas (Figura

3.17). Otra utilización frecuente es para modelar organigramas y, en herramientas de edición de textos, para mantener los índices de las publicaciones (Figura 3.18). Otros ejemplos son las jerarquías de clases en los lenguajes orientados a objetos, o la jerarquía de países, provincias, departamentos y municipios que organiza al poder político de una república, etc.

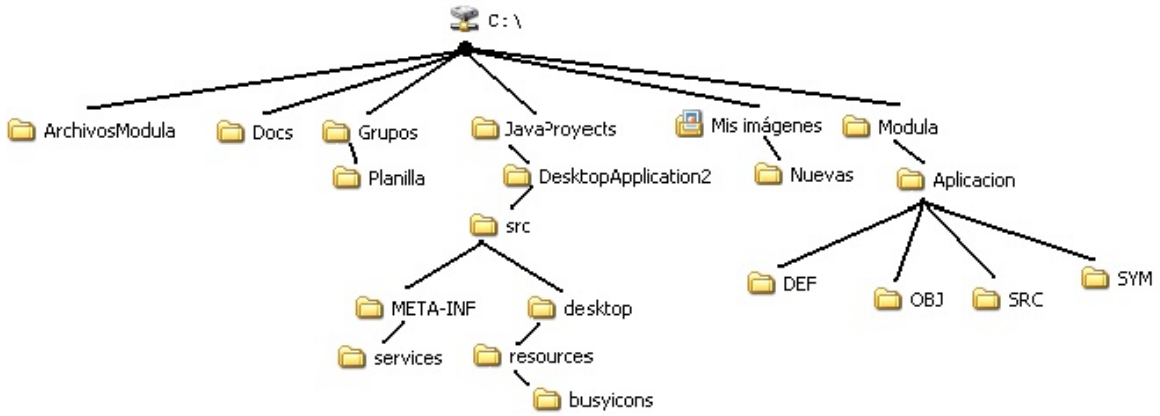

Figura 3.17: Uso de árbol genérico para mantener sistemas de archivos

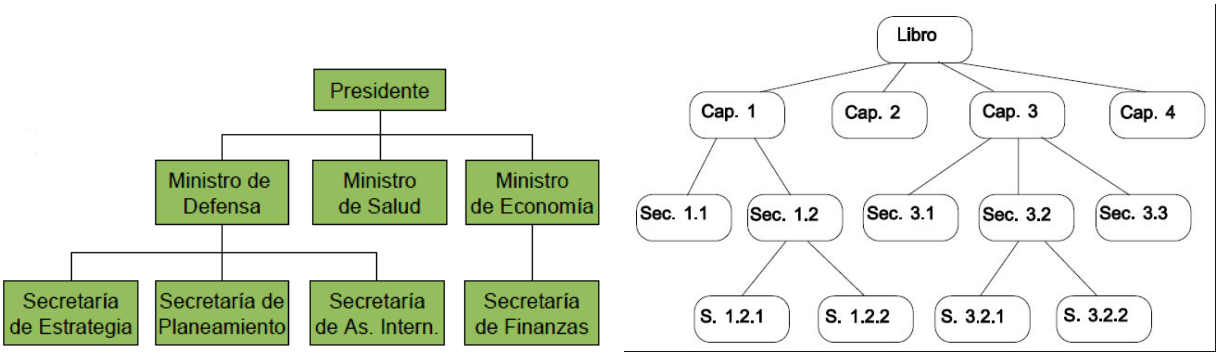

Figura 3.18: Uso de árbol genérico para mantener organigramas e índices de publicaciones

---

Ejercicio 3.3: Más ejercicios de estructuras jerárquicas

1. Modificar el TDA ArbolBinario (clase ArbolBin) para agregar las siguientes operaciones:

   a) *verificarPatron(lisPatron)* que recibe por parámetro una lista *lisPatron* y determine si dicha lista coincide exactamente con al menos un camino del árbol que comience en la raíz y termine en una hoja. El método debe retornar un valor boolean. Debe ser eficiente, es decir, recorrer el árbol lo estrictamente necesario.

   b) *listaQueJustificaLaAltura()* que devuelve una lista de elementos que es el camino que comienza en la raíz y termina en la hoja más lejana.

2. Modificar el TDA ArbolGenerico (clase ArbolGen) para agregar las operaciones *verificarPatron(lisPatron), frontera()* y *listaQueJustificaLaAltura*() con definición equivalente a la indicada para árbol binario. Debe tener cuidado al definir hoja y camino, ya que las condiciones son distintas en ambos tipos de árboles.
