Estructuras de datos y Algoritmos. Curso 2009/10
Unidad Temática III: Estructuras de datos de Búsqueda y su Jerarquía Java

Tema 2. Introducción a la Representación Jerárquica de una EDA: árbol Binario

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Informática Aplicada

Índice

Objetivos y Bibliografía

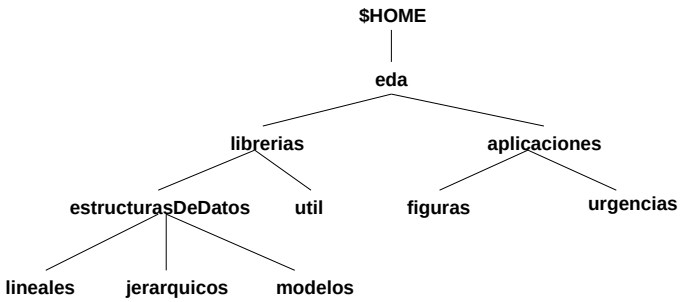
El principal objetivo de este tema es presentar el **árbol Binario** (**AB**) como la estructura jerárquica en la que basan su definición las Representaciones según un árbol de Diccionario y Cola de Prioridad que se estudian en esta asignatura: el árbol Binario de Búsqueda (ABB) para ambas EDA y el Montículo Binario (MB) solo para la segunda, que se estudiarán en los dos temas siguientes de esta misma unidad.

Para llevar a cabo el estudio del AB el primer apartado introduce los conceptos generales asociados a la estructuración de una Colección de datos según un **árbol**, lo que permitirá en su segundo apartado, obtener los de árbol Binario: su definición y propiedades, los tipos de Exploración In-Orden, Pre-Orden, Post-Orden y Por Niveles que admite y, en base a todo ello, las dos condiciones que debe cumplir un AB para soportar efectivamente la Búsqueda Dinámica característica de Diccionario y Cola de Prioridad, equilibrio para su estructura y cierta propiedad de Ordenación entre sus datos; será para satisfacer estas condiciones cuando se plantearán las definiciones de ABB y MB como subtipos de un AB.

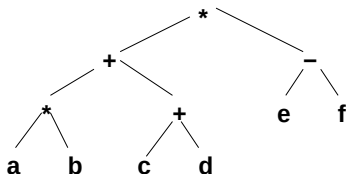
Como bibliografía básica de este tema se utilizará los apartados 5 y 6 del Capítulo 6 y los apartados del 1 a la introducción del 4 del Capítulo 17 del libro de Weiss, M.A. **Estructuras de datos en Java** (Adisson-Wesley, 2000).

1. La Estructura Jerárquica Árbol

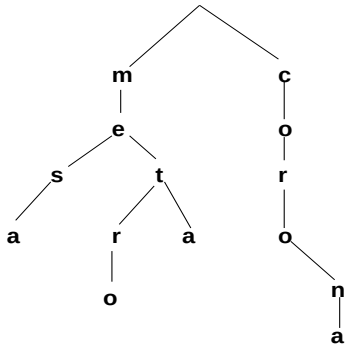
El Árbol es una estructura jerárquica ampliamente utilizada en distintas áreas de la Informática, por ejemplo para organizar algunas colecciones cuyos datos ya guardan de per-se una relación de Jerarquía entre ellos. Un ejemplo de este tipo de colección es la compuesta por los distintos directorios y ficheros en los que un sistema operativo organiza la información, por lo que su estructuración según un árbol resulta la más natural; la siguiente figura lo muestra gráficamente para el directorio de prácticas de EDA de un usuario dado:



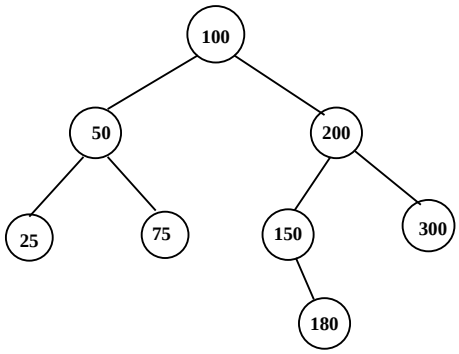
Otro ejemplo de datos que guardan entre sí una relación de Jerarquía es el de los operadores y operandos que componen una expresión aritmética a evaluar; a modo de ejemplo, la siguiente figura muestra el árbol que representa la expresión aritmética que en notación Infija y totalmente parentizada se escribe como (((a∗b) + (c+d)) ∗ (e−f)).



También el árbol es la forma de representación que permite la búsqueda eficiente de una palabra en una colección; por ejemplo, si la colección `mes`, `mesa`, `meta`, `metro`, `coro` y `corona` utiliza una Representación Lineal, una operación de búsqueda de una palabra requiere un tiempo acotado superiormente por la talla de la Colección. Sin embargo, si la misma colección se representa sobre un Árbol de Palabras o *Trie*, como el que muestra gráficamente la siguiente figura, la búsqueda solo exige un tiempo proporcional a la longitud de dicha palabra y, por tanto, independiente de la talla o número de palabras de la colección.



De hecho, **la Representación según determinados tipos de árbol de Diccionario o Cola de Prioridad soporta la Búsqueda Dinámica característica de estas EDA**, esto es permite garantizar que un un dato de clave dada o el de máxima prioridad se pueda `recuperar`, `insertar` y `eliminar` en tiempo sublineal. Por ejemplo el siguiente árbol representa la Colección de `Integer` 100, 50, 200, 25, 75, 150, 300 y 180:



En este árbol, cara a favorecer el proceso de Búsqueda Dinámica de un dato dado, los `Integer` de la colección se disponen de la siguiente forma ordenada: el primero, el `100`, ocupa la denominada Raíz del árbol y, a partir de ésta, cada dato menor que el primero se sitúa en la zona del árbol situada a la izquierda de su Raíz o subárbol Izquierdo y cada dato mayor que el primero en la zona del árbol situada a la derecha de su Raíz o subárbol Derecho. Además, para cada subárbol Izquierdo o Derecho el proceso se repite a partir de su Raíz; por ejemplo, el dato `75` al ser menor que `100` pero mayor que `50` se sitúa a la izquierda de la Raíz, ocupada por `100`, y a la derecha del subárbol cuya Raíz ocupa `50`. Por tanto, para encontrar el `Integer` `75` en el árbol bastarán `3` comparaciones, mientras que si se hubiera utilizado una LEG o un `array` para disponer los `Integer`, en el Peor de los Casos el `75` se hubiera encontrado en un tiempo lineal con la talla. árboles de este tipo, denominados árboles Binarios de Búsqueda, se estudiarán más adelante con detalle para conocer bajo qué condiciones específicas garantizan una Búsqueda Dinámica con coste logarítmico aún en el Peor de los Casos.

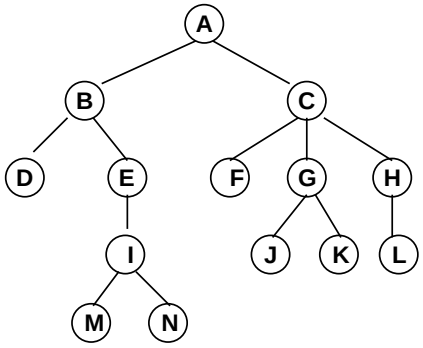
Introducido el papel del árbol en la Representación e Implementación eficiente de las EDA de Búsqueda, en las siguientes secciones de este apartado se realiza la definición formal de dicha estructura y se introducen tanto la terminología empleada para referirse a sus componentes como algunos conceptos generales que servirán de referencia en el resto de apartados de este tema.

1.1. Definición y terminología asociada

Un **árbol** es una estructura jerárquica que se puede definir por medio de **un Conjunto de nodos**, uno de los cuales es distinguido como la **Raíz** del árbol, y **un Conjunto de Aristas** tal que cualquier nodo `H`, a excepción de la Raíz, está conectado por medio de una Arista a un único nodo `P`; se dice entonces que `P` es el **Padre** de `H` y `H` es un **Hijo** de `P`. Si un nodo no tiene ningún Hijo se denomina **Hoja**; excluidas las Hojas, al resto de nodos del árbol, los que tienen algún hijo, se les denomina nodos **Internos**.

Para ejemplificar las definiciones realizadas, en la siguiente figura se muestra un árbol con `14` nodos etiquetados con los `Character` del `A` al `N`; nótese que el nodo etiquetado con `A` es la

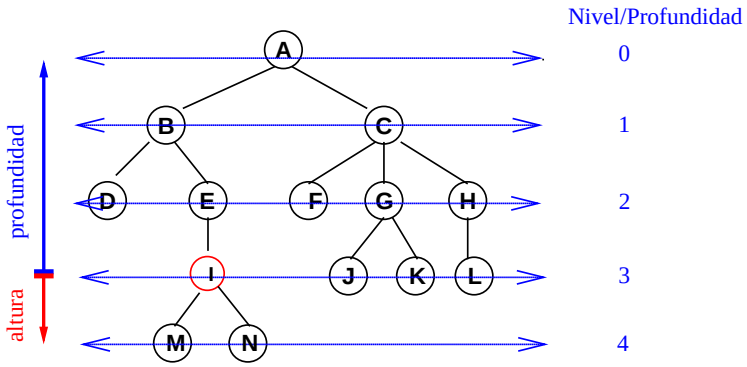
Raíz del árbol y los etiquetados con `B` y `C` son sus Hijos; el nodo con etiqueta `C` es el Padre de los nodos con etiquetas `F`, `G` y `H` y el nodo `I` tiene dos Hijos etiquetados como `M` y `N`. Asimismo, los nodos con etiquetas `D`, `M`, `N`, `F`, `J`, `K` y `L` son Hojas y el resto nodos Internos.



1.2. Caminos entre nodos: profundidad, nivel y altura de un nodo

Un árbol se caracteriza porque desde la raíz a cada uno de los nodos hay un **Camino** único cuya **longitud** viene dada por el número de Aristas que lo componen; por ejemplo, en el árbol de la figura anterior el camino desde la raíz hasta la hoja etiquetada como `N` es `A-B-E-I-N` y tiene longitud `4`. A partir de la longitud de un camino se definen la profundidad, el nivel y la altura de un nodo y un árbol como sigue.

Se denomina **profundidad** de un nodo a la longitud del camino que va desde la raíz hasta él; por tanto, la profundidad de la raíz es `0`. Además, se dice que todos los nodos que están a la misma profundidad están en el mismo **nivel**, siendo cero el de la raíz. Por ejemplo, en la figura siguiente los nodos `D`, `E`, `F`, `G` y `H` ocupan el nivel `2` del árbol, pues todos ellos tienen la misma profundidad, también `2`. Esta misma figura permite observar también que la representación gráfica que se hace de un árbol es precisamente por niveles, empezando por el de la Raíz, el cero, hasta el que ocupan las hojas. Finalmente, la **altura** de un nodo se define como la longitud del camino que va desde tal nodo hasta la hoja más profunda bajo él. En base a esta definición, la altura de un árbol es la de su nodo raíz, `4` la del árbol de la figura anterior, y la altura de una Hoja es cero; así, los nodos I, G y H tienen altura 1, los E y C altura 2 y el B altura 3.



1.3. Relaciones entre nodos: tamaño de un nodo

Utilizando la nomenclatura típica de la Genealogía, a los nodos de un árbol que tienen el mismo padre se les denomina **Hermanos**; así, en el árbol de `Character` de la anterior figura los nodos `F`, `G` y `H` son Hermanos porque tienen el mismo padre, el nodo etiquetado con `C`. Asimismo, si hay un camino desde el nodo `u` hasta el nodo `v`, se dice que `u` es un **Ascendiente** de `v` y que `v` es un **Descendiente** de `u`. Si $u \neq v$, entonces `u` es un **Ascendiente Propio** de `v` y `v` es un **Descendiente Propio** de `u`; así, en el árbol ejemplo son Ascendientes Propios del nodo etiquetado con `N` los nodos `I`, `E`, `B` y `A` y son Descendientes Propios de `C` los nodos `F`, `G`, `H`, `J`, `K` y `L`.

En base a la nomenclatura que se acaba de introducir, el **tamaño** de un nodo se define como el número de descendientes que tiene y el de un árbol como el de su nodo Raíz; así en el ejemplo del árbol de `Character` el tamaño del nodo `C` es `7`, el del nodo `B` `6` y el del árbol `14`.

2. Árbol Binario (AB)

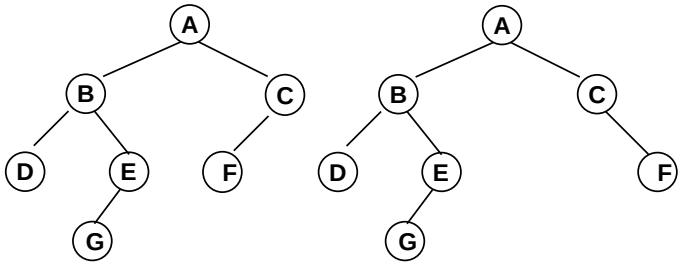
En este apartado se estudia un tipo específico de árbol denominado **Árbol Binario**: su definición y propiedades y las operaciones de recorrido por niveles y en profundidad. Será la base para la definición en los temas siguientes del Árbol Binario de Búsqueda (ABB) y el Montículo Binario (MB) que resultan de imponer ciertas condiciones como el equilibrio para su estructura y cierta propiedad de ordenación entre sus datos.

Notar finalmente que, si bien su estudio queda fuera del alcance de los objetivos de esta asignatura, los Árboles Binarios admiten otros usos de gran interés en Computación; por ejemplo son binarios los árboles de expresión que se utilizan en el diseño de compiladores para la representación y evaluación de expresiones aritméticas con operadores binarios y el árbol de Codificación que se construye en el algoritmo de compresión de datos de Huffman (ver detalles en la sección `11.2.4` y el apartado `12.1`, respectivamente, del libro de Weiss).

No es objetivo del tema ofrecer una implementación concreta en Java de un árbol Binario y será en los Temas 3 y 4 de esta misma Unidad Didáctica en los que se estudiarán las implementaciones de ABBs y MB respectivamente.

2.1. Definición y propiedades

Un **Arbol Binario** se define como un Árbol en el que ningún nodo puede tener más de dos hijos, que son distinguidos como hijos **Izquierdo** y **Derecho**. Nótese entonces que el árbol de `Character` que se viene usando como ejemplo **no** es un Árbol Binario: uno de sus nodos, el etiquetado con `C`, tiene `3` Hijos. Sin embargo, **sí** son Árboles Binarios los que se muestran en la siguiente figura, ambos con `7` nodos etiquetados con `Character` y que solo difieren entre sí en que el nodo con etiqueta `F` es el Hijo Izquierdo del etiquetado con `C` en el AB de la izquierda y su Hijo Derecho en el de la derecha:

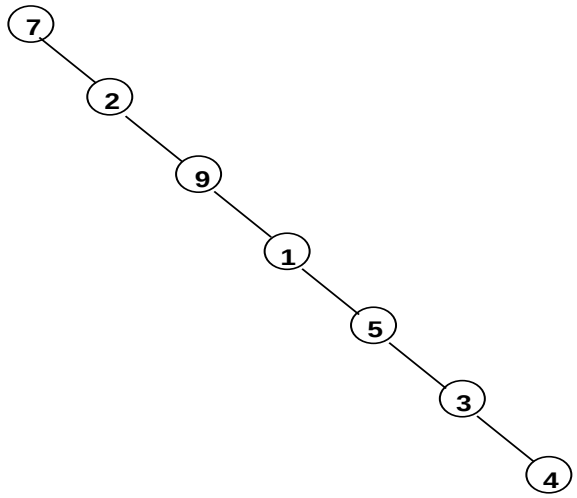


Un resultado muy interesante que se puede extraer de esta definición, fácilmente demostrable por Inducción, es el siguiente: **el número máximo de nodos del nivel `i` de un AB de altura dada `H` es 2^{i},** $0 \leq i \leq H$. A partir de este resultado y en base a las definiciones previamente realizadas, para el mismo AB de altura dada `H` se tiene que:

1. su **número máximo de nodos** es $\sum_{nivel=0 \ldots H} 2^{nivel} = 2^{H+1} - 1$;
2. su **número máximo de hojas** es $(2^{H+1} - 1) - (\sum_{nivel=0 \ldots H-1} 2^{nivel}) = 2^{H}$;
3. su **número máximo de nodos internos** es $(2^{H+1} - 1) - (2^{H}) = 2^{H} - 1$.

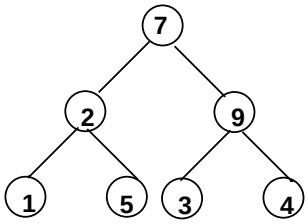
Nótese que dentro de los límites establecidos para el número de nodos de un AB (total, por nivel, etc.), lo que determina su topología y la relación concreta entre su altura y su tamaño no es más que la forma en la que los `N` datos se distribuyen en sus `H` niveles. En concreto, considérense las siguientes instancias de distribución de una Colección de datos sobre un AB, así como las variaciones en la topología y relación entre `H` y `N` a las que dan lugar:

- **Instancia `1`:** cada dato de la colección se dispone en un nivel diferente del AB, por lo que $H = N-1$ y se tiene un AB **Completamente Degenerado** o, en realidad, una Secuencia o Lista de nodos de talla `N`. La siguiente figura muestra un AB Completamente Degenerado de altura $H = 6$ que resulta de distribuir según esta primera instancia, en este orden, los `Integer` 7, 2, 9, 1, 5, 3 y 4 ($N = 7 = H+1$):



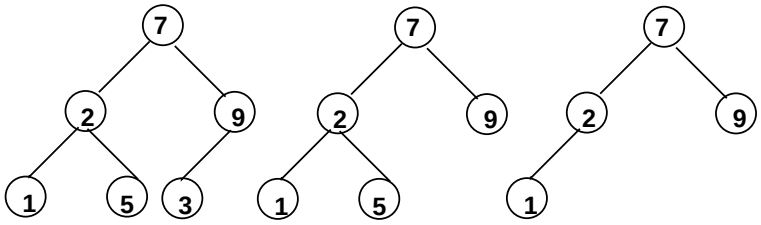
- **Instancia `2`:** cada nivel `i`, $0 \leq i \leq H$, del AB está lleno, i.e. cada uno de los 2^{i} nodos de cada nivel -número máximo de nodos por nivel- está ocupado por uno de los `N` datos de la Colección; cuando esto ocurre se tiene un AB **Lleno** de altura $H = \lfloor \log_2 N \rfloor$ y tamaño

$N = 2^{H+1}-1$; como ilustra claramente la siguiente figura, si se distribuyen sobre un AB los mismos $N = 7$ `Integer` del ejemplo precedente, en el mismo orden que entonces pero según esta segunda instancia, se obtiene un AB Lleno con $H = 2 = \lfloor \log_2 N \rfloor$:



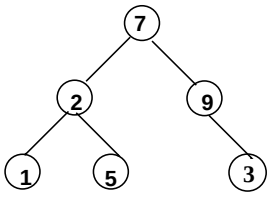
- **Instancia 3:** cada nivel i, $0 \leq i \leq H$, del AB está lleno a excepción quizás del último, el H, que en tal caso debe de tener situados todos sus nodos, las hojas del árbol, tan a la izquierda como sea posible; en este caso se tiene un AB **Completo** y en él se cumplen lo siguiente: $H = \lfloor \log_2 N \rfloor$ y $2^H \leq N \leq 2^{H+1}-1$.

  Nótese que, por definición, mientras que un AB Lleno es un AB Completo -se da entonces la igualdad en las fórmulas anteriores- no es cierta la afirmación inversa; la siguiente figura ilustra este hecho mostrando todos los AB Completos que se pueden obtener a partir del AB Lleno de la figura precedente simplemente eliminando de la Colección, respectivamente, los datos 4, 4 y 3 y 4, 3 y 5:

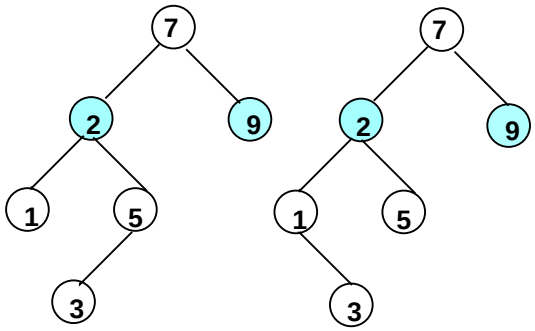


- **Instancia 4:** los N datos de la colección se distribuyen en los distintos niveles del AB de tal forma que la diferencia de alturas entre su hijo derecho y su hijo izquierdo es como máximo 1. Esta cuarta instancia da lugar a un AB **Equilibrado**, para el que se cumplen las siguientes desigualdades: $H \leq \lfloor \log_2 N \rfloor$ y $N \leq 2^{H+1}-1$.

  Nótese que no solo los AB Completos y Llenos son Equilibrados. También lo son aquellos como el que se muestra a continuación, un AB que sin ser Completo presenta una diferencia de alturas cero entre su Hijo Derecho y su Hijo Izquierdo; este AB se ha obtenido a partir de los mismos datos que el primero de los AB Completos de la figura anterior pero situando el 3 en el hijo derecho del nodo que contiene al 9 en lugar de en el izquierdo:



Ahora bien, si para la misma colección de este último ejemplo el dato 3 ya no se sitúa en un hijo del nodo que contiene al 9 sino en uno del nodo que contiene al 1 ó al 5 se



obtiene un AB **no** Equilibrado; como se muestra en la siguiente figura, la diferencia de alturas entre los subárboles del nivel `1` de tal AB, de Raíz sombreada, es mayor que `1`:
Cuestión: para equilibrar cualquiera de los dos AB de la figura previa, ¿dónde se podría insertar el dato `4`?

De la clasificación topológica o estructural realizada para los AB es fácil extraer dos conclusiones. La primera y más inmediata de ellas es que **la altura `H` de un AB se puede acotar por una función de su tamaño `N`**; específicamente,

- `H` está acotada inferiormente por la función $\lfloor \log_2 N \rfloor$, la función logarítmica que expresa la altura máxima de un AB Equilibrado;
- `H` está acotada superiormente por `N-1`, la función lineal que expresa la altura exacta de un AB Completamente Degenerado.

Obsérvese asimismo que utilizando estas dos cotas las instancias significativas que se podrían plantear durante el análisis del coste de una operación sobre un AB son la `1` y la `4`, las que originan respectivamente un AB Completamente Degenerado y un AB Equilibrado.

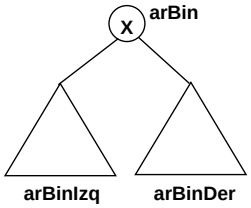
La segunda conclusión se puede extraer añadiendo a la primera el siguiente hecho: al ser por definición la altura `H` de un AB la longitud de su `Camino Más Largo (CML)`, el coste Temporal de cualquier `Exploración del CML` de un AB es estrictamente del orden de `H`. Por lo tanto,
si $T_{ECML}(N)$ es el coste Temporal Asintótico de una `Exploración del CML` de un AB de tamaño `N`, entonces:

- **en el Mejor de los Casos**, si el AB sobre el que se realiza la operación es Equilibrado, $T_{ECML}(N) \in \Omega(\log N)$, la altura de tal AB;
- **en el Peor de los Casos**, si el AB sobre el que se realiza la operación es Completamente Degenerado, $T_{ECML}(N) \in O(N)$, la altura de tal AB.

Las dos conclusiones que se acaban de presentar permitirán más tarde determinar las condiciones efectivas que debe cumplir un AB para soportar la Búsqueda Dinámica.

2.2. Definición recursiva de un Árbol Binario

Un Árbol Binario bien es un AB vacío o bien tiene un nodo Raíz y dos subárboles Binarios, uno Izquierdo y otro Derecho, que también pueden ser vacíos. Con la siguiente figura se ilustra gráficamente esta definición para un AB no vacío denominado `arBin`, cuyo nodo Raíz ocupa el dato `x` y cuyos subárboles Izquierdo y Derecho se denominan respectivamente `arBinIzq` y `arBinDer`:



Nótese que esta definición proporciona una descomposición recursiva de un AB en la que es posible basar el diseño recursivo de sus operaciones de exploración en profundidad; recuérdese del tema de Diseño Recursivo que si el dominio de los datos de un problema admite una descomposición recursiva entonces su solución para una cierta talla se puede expresar de forma fácil y natural en términos de la solución al mismo problema pero para una talla menor.

2.3. Operaciones de Exploración de un AB

Visto que los nodos que componen la estructura de un AB se disponen bien por Caminos (Ramas) o bien por Niveles, en función de la semántica concreta de la operación a realizar dos son los tipos de Exploración entre los que puede interesar elegir:

- **en Profundidad**, esto es «bajando» por las ramas del AB y empezando siempre por la situada más a la izquierda;
- **por Niveles**, esto es visitando los nodos del AB por niveles y, dentro de cada nivel, siempre de izquierda a derecha.

A su vez la Exploración en Profundidad de un AB se puede realizar en **Pre-Orden** o en **Post-Orden** o en **In-Orden** según interese procesar cada nodo a visitar, respectivamente,

- **antes de** visitar sus dos hijos empezando por el izquierdo, o **en Pre-Orden**;
- **después de** visitar sus dos hijos empezando por el izquierdo, o **en Post-Orden**;
- **después de** visitar su hijo izquierdo **y antes de** visitar su hijo derecho, o **en In-Orden**.

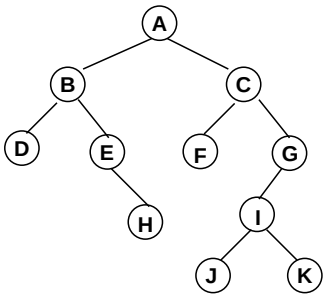
Obsérvese también que sería igualmente correcta una definición simétrica de estos tres mismos tipos, esto es visitando primero el Hijo Derecho y luego el Izquierdo.

En base a estas definiciones de tipos de Exploración en Profundidad, `altura()` y `tamanyo()` son operaciones de Recorrido en Post-Orden puesto que para calcular la altura y tamaño de un AB, por definición las de su nodo Raíz, antes es necesario haber calculado las de sus Hijos. Sin embargo, la operación `duplicadoDe`, que hace que el AB sobre el que se aplica sea un duplicado de un AB `original` dado, es un Recorrido en Pre-Orden pues solo una vez que el nodo Raíz de `original` ha sido duplicado resulta posible duplicar sus Hijos; también es en Pre-Orden la

Búsqueda de un dato `x` en un AB ya que solo después de haber comprobado que `x` no es el dato que ocupa su nodo Raíz tiene sentido seguir buscándolo en sus Hijos.

Ejercicios propuestos:

1. Indíquense los tipos de Recorrido en Profundidad de un árbol (Binario) de Expresión que obtienen la expresión aritmética que éste representa totalmente parentizada en notación `Infija`, `Postfija` y `Prefija`.
2. Suponiendo que la operación `toString` de un AB se define como un Recorrido por Niveles, determínese el (`String`) resultado de su ejecución sobre el AB de `Character` que se muestra en la siguiente figura:



   Obténgase después el resultado de la misma operación pero suponiendo que, en lugar de por Niveles, se define como un Recorrido en Pre-Orden, un Recorrido en Post-Orden y un Recorrido en In-Orden. Compuébese también que el orden en el que aparecen los datos del AB en cada `String` resultado es precisamente el orden en el que se visitan sus nodos durante el correspondiente Recorrido.

Establecido su tipo, la transcripción algorítmica de una operación de Exploración se puede obtener describiendo la visita a los nodos del AB en el orden o secuencia que determina tal tipo, básicamente la forma en que se accede al siguiente nodo a visitar a partir de uno dado que inicialmente coincide con su Raíz. A continuación se muestra cómo, recursivamente primero para las operaciones de Exploración en Profundidad de un AB e iterativamente después para las de Exploración por Niveles, pues como se podrá comprobar de inmediato el tipo de una operación de Exploración también determina la naturaleza recursiva o iterativa del algoritmo que la describe.

Advertir también que aunque no se dará ahora una implementación completa en Java de la clase que representa un árbol Binario, en lo que sigue para describir las operaciones de manejo de un AB supondremos que:

- Un árbol Binario de tamaño `N` es un objeto de cierto tipo `AB<T>` que `TIENE UNA raiz` de tipo `NodoAB<T>`.
- Las operaciones sobre un `AB` deberían ser públicas y dinámicas.
- Un objeto de tipo `NodoAB<T>` `TIENE UN dato` de tipo `T`, y dos objetos `NodoAB<T>` `izq, der` para acceder respectivamente a su hijo izquierdo y a su hijo derecho.

2.3.1. Exploración en Profundidad de un AB

La definición de una Exploración en Profundidad de un AB resulta muy fácil y natural cuando se utiliza la definición recursiva de un nodo de un AB: al «bajar» por un AB el nodo que sigue o antecede a uno dado tanto en Pre-Orden como en In-Orden y Post-Orden solo puede ser uno de sus Hijos, el Izquierdo o el Derecho según el orden.
Cuestión: utilizando el mismo argumento que justifica como natural la definición recursiva de una Exploración en Profundidad, arguméntese el motivo por el que resulta imposible describir recursivamente la Exploración por Niveles de un AB.

Así pues, y recordando lo dicho en el tema dedicado a la Recursión como herramienta de diseño, el problema de explorar en Pre-Orden o Post-Orden o In-Orden del nodo Raíz de un AB se puede describir trivialmente en términos de la solución al mismo problema pero de tamaño menor, i.e. recursivamente. A modo de ejemplo, la siguiente sería la descripción recursiva del Recorrido en Pre-Orden de un cierto nodo `n` no vacío de un AB:

- **Caso base y su solución:** si `n` es una Hoja, típicamente, tratar el dato que contiene;
- **Caso general y su solución:** si `n` no es una Hoja, la siguiente secuencia de acciones describen su visita en Pre-Orden:
  1. tratar el dato que contiene;
  2. si su Hijo Izquierdo no es un nodo vacío entonces recorrerlo en Pre-Orden;
  3. si su Hijo Derecho no es un nodo vacío entonces recorrerlo en Pre-Orden;

Nótese que haciendo implícitos el caso base y su solución la siguiente sería una transcripción algorítmica válida del Recorrido en Pre-Orden de un cierto nodo `n` no vacío de un AB:

```
/** SII n!=null: recorre en Pre-Orden el nodo n de un AB **/
T' recorrerPreOrden(nodoAB<T> n){
    T' resdatoI = tratar(n.dato);
    // Atención: la Precondición no garantiza que los Hijos de n no sean vacíos
    if ( n.izq!=null ) T' resIzqI = recorrerPreOrden(n.izq);
    if ( n.der!=null ) T' resDerI = recorrerPreOrden(n.der);
    T' resRecorrerI = combinar(resdatoI, resIzqI, resDerI);
    return resRecorrerI;
}
```

Obviamente, si `n` fuese la Raíz del AB a recorrer, el esquema anterior serviría para recorrer en Pre-Orden un AB no vacío; atendiendo entonces a la identificación AB-nodo, el esquema de Recorrido en Pre-Orden de un AB, vacío o no, sería:

```
/** recorre en Pre-Orden un AB **/
void recorrerPreOrden(){
    if ( !this.esVacio() ) recorrerPreOrden(this.raiz);
    else procesarABVacio();
}
```

Cuestión: indíquense las modificaciones que requieren los esquemas anteriores para expresar el Recorrido en Post-Orden e In-Orden de un AB.

Una vez obtenidos los esquemas recursivos de Recorrido en Profundidad de un AB y dando por supuesto que, efectivamente, hacen lo que deben de hacer -recorrer en Pre-Orden o Post-Orden o In-Orden un AB simplemente recorriendo en el mismo orden su nodo Raíz- resulta indispensable demostrar que además lo hacen en un tiempo finito y acotado superiormente por el tamaño del AB; de lo contrario, y por muy correctos que fueran, no serían procedimientos válidos. Para ello basta analizar su coste Temporal como sigue: sea `x` el tamaño del nodo a recorrer en `UnOrden` (`PreOrden`, `PostOrden` o `InOrden`) con el esquema `recorrerUnOrden` y sea la constante $k \geq 1$ el coste Temporal de `tratar` el dato que contiene dicho nodo; la Relación de Recurrencia que expresa la Complejidad Temporal de una llamada cualquiera a `recorrerUnOrden` será, en función del grado de equilibrio del AB, una u otra de las siguientes:

- **en el Peor de los Casos**, cuando se recorre un AB Completamente Degenerado, $T_{recorrerUnOrden}{}^{P}(x > 1) = 1 * T_{recorrerUnOrden}{}^{P}(x - 1) + k$
- **en el Mejor de los Casos**, cuando se explora un AB Equilibrado, $T_{recorrerUnOrden}{}^{M}(x > 1) = 2 * T_{recorrerUnOrden}{}^{M}(\frac{x}{2}) + k$

Aunque cada una de estas relaciones se resuelve, y acota, utilizando Teoremas distintos, el 1 con $a = c = 1$ para el Peor de los Casos y el 3 con $a > 1$ y $c = 2$ para el Mejor, el resultado que se obtiene para ambas es el mismo: $T_{recorrerUnOrden}(x) \in \Theta(x)$ independientemente de la instancia. Como la llamada principal al esquema se realiza sobre el nodo Raíz del AB, $x = N$ y entonces el coste Temporal del Recorrido en `UnOrden` (`PreOrden`, `PostOrden` o `InOrden`) de un AB es lineal con su tamaño para cualquier instancia, i.e. $T_{recorrerUnOrden}(N) \in \Theta(N)$.

Con los siguientes ejemplos se ilustra el uso de los esquemas presentados a la hora de describir una determinada operación de Recorrido en Profundidad de un AB.

Ejemplo 1. Supóngase que se quiere diseñar la operación `toStringPreOrden()` que obtiene un `String` con los datos de un AB ordenados en PreOrden. Para ello, obviamente, hay que instanciar el esquema de Recorrido en Pre-Orden de un AB como sigue:

```
/** devuelve los datos de un AB ordenados en Pre-Orden **/
String toStringPreOrden(){
    if ( !this.esVacio() ) return toStringPreOrden(this.raiz); else return "*";
}
// SII n!=null: obtiene los datos del nodo n en Pre-Orden
String toStringPreOrden(nodoAB<T> n){
    String res = n.dato.toString()+"-";
    if( n.izq!=null ) res += toStringPreOrden(n.izq);
    if( n.der!=null ) res += toStringPreOrden(n.der);
    return res;
}
```

Nótese que en este caso concreto `procesarABVacio()` consiste en devolver el `String "*"` para advertir que, al estar vacío, el AB no se explora. Siguiendo un proceso análogo, la transcripción algorítmica de `toStringPostOrden()` y `toStringInOrden()` sería como sigue:

```
/** devuelve los datos de un AB ordenados en Post-Orden **/
String toStringPostOrden(){
    if ( !this.esVacio() ) return toStringPostOrden(this.raiz); else return "*";
}
```

```
// SII n!=null: obtiene los datos del nodo n en Post-Orden
String toStringPostOrden(nodoAB<T> n){
    String res = "";
    if( n.izq!=null ) res += toStringPostOrden(n.izq);
    if( n.der!=null ) res += toStringPostOrden(n.der);
    String res += n.dato.toString()+"-";
    return res;
}
/** devuelve los datos de un AB ordenados en In-Orden **/
String toStringInOrden(){
    if ( !this.esVacio() ) return toStringInOrden(this.raiz); else return "*";
}
// SII n!=null: obtiene los datos del nodo n en In-Orden
String toStringInOrden(nodoAB<T> n){
    String res = "";
    if( n.izq!=null ) res += toStringInOrden(n.izq);
    String res += dato(i).toString()+"-";
    if( n.der!=null ) res += toStringInOrden(n.der);
    return res;
}
```

Además del esquema que se ha instanciando en el ejemplo anterior, existe otro muy similar pero en el que el nodo vacío es, en lugar del Hoja, el caso base de la Recursión. Las definiciones de la operaciones `tamanyo()` y `altura()` de un AB que se realiza a continuación, así como la de la cuestión que las acompaña, proporcionan un buen ejemplo de uno de los motivos por los que puede ser necesario utilizar este otro esquema: obtener una descripción recursiva lo más compacta y eficiente posible.

Ejemplo 2. La descripción más directa y natural del Recorrido en Post-Orden de un AB que realiza la operación `tamanyo` sería la siguiente: por definición, el tamaño de un AB es el tamaño de su nodo Raíz y, también por definición, el tamaño de un nodo `n` de un AB viene dada por la expresión `1 + tamanyo(n.izq) + tamanyo(n.der)`. Codificando este análisis, las operaciones que calculan la altura de un AB y la de uno de sus nodos serían las siguientes:

```
/** obtiene el tamaño de un AB **/
int tamanyo(){ return tamanyo(this.raiz); }
// obtiene el tamaño del nodo n
int tamanyo(nodoAB<T> n){
    if ( n==null ) return 0;
    else return 1 + tamanyo(n.izq) + tamanyo(n.der);
}
```

Cuestión: una definición recursiva alternativa de `tamanyo` sería la siguiente: si un AB es vacío su tamaño es cero y sino el de su nodo Raíz, o algorítmicamente:
`int tamanyo(){ if ( this.esVacio() ) return 0; else return tamanyo(this.raiz());}`
Tras escribir la operación homónima y recursiva que calcula el tamaño de un nodo no vacío de un AB indíquese el motivo por el que esta segunda definición de `tamanyo()` resulta menos pertinente que la primera.

Ejemplo 3. Siguiendo el mismo esquema empleado para `tamanyo()`, la operación `altura()` de un AB se podría pensar en describir como sigue:

```
/** obtiene la altura de un AB **/
int altura(){ return altura(this.raiz); }
// obtiene la altura del nodo n
int altura(nodoAB<T> n){
    if ( n==null ) return 0;
    else return ( 1 + Math.max(altura(n.izq), altura(n.der)) );
}
```

Nótese sin embargo que, por desgracia, tan compacta descripción presupone que la altura de un nodo vacío es 0, **¡ pero cero es la altura de una Hoja !** La buena noticia es que existe una forma muy sencilla de resolver este conflicto sin cambiar ni la definición del caso base ni la del general: que -1 sea la altura de un nodo vacío. En ese caso,

```
int altura(nodoAB<T> n){
    if ( n==null ) return -1;
    else return ( 1 + Math.max(altura(n.izq), altura(n.der)) );
}
```

Cuestión: una solución alternativa a la ya presentada para `altura()` es definir como caso base de la recursión el nodo hoja, que es el que tiene altura cero. Descríbanse para este caso los esquemas algorítmicos de las operaciones que calculan la altura de un AB y la de uno de sus nodos, indicando también si son más adecuados que los ya presentados.

Para concluir con los ejemplos de Exploración en Profundidad de un AB se podría plantear ahora los de la inserción y el borrado de un dato en él, aunque como se verá más tarde cualquiera de estas operaciones se puede describir también mediante una Búsqueda por Niveles.

Una vez familiarizados con los esquemas de Recorrido en Profundidad de un AB resulta muy fácil identificar las pocas modificaciones que necesitan para transformarse en esquemas de Búsqueda. A modo de ejemplo se presenta ahora el esquema algorítmico que describe la operación de Búsqueda de un dato dado `x` en un AB, que si se recuerda se ha definido como una Exploración en Pre-Orden pues solo tiene sentido seguir buscando el dato `x` en el AB una vez comprobado que no es el dato que ocupa su raíz.

```
/** obtiene la primera aparición en Pre-Orden de x en un AB;
  * si x no estuviera lo advierte lanzando la Excepción ElementoNoEncontrado **/
T recuperar(T x) throws ElementoNoEncontrado{
    NodoAB<T> res = recuperar(x, this.raiz);
    if ( res == null ) throw new ElementoNoEncontrado("al buscar: el dato "+x+" no está");
    return res.dato;
}
// obtiene el primer nodo en Pre-Orden de actual que contiene a x, null si x no está
NodoAB<T> recuperar(T x, NodoAB<T> n){
    NodoAB<T> res = null;
    if ( n!=null ){
        if ( n.dato.equals(x) ) res = actual;
        else{ res = recuperar(x, n.izq);
              if ( res == null ) res = recuperar(x, n.der);
        }
    }
    return res;
}
```

Obsérvese en primer lugar que el esquema de Recorrido en Pre-Orden del que se parte es el de caso base nodo vacío, como ocurre en `tamanyo()` y `altura()`: se debe advertir que `x` no está tanto si el AB es vacío como si al concluir la Exploración en Pre-Orden de todos sus nodos no se ha encontrado. Tras esta elección, nótese que las modificaciones que sufre el esquema de Recorrido para transformarlo en el de Búsqueda en Pre-Orden son mínimas, solo afectan a su caso general y dan cuenta del hecho de que al encontrar `x` en un nodo del AB ya no hace falta seguir buscándolo.

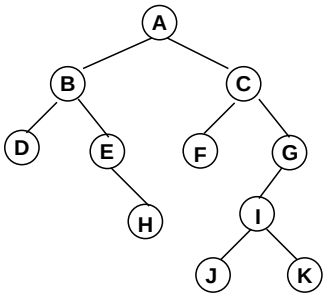
En lo referente al coste Temporal de cualquier esquema de Búsqueda en Profundidad, es fácil comprobar que en el Peor de los Casos, cuando no se encuentra el dato buscado, es lineal con el tamaño del AB y en el Mejor de los Casos, cuando el dato que se busca es el que ocupa la Raíz del AB, es del orden de una constante.

Ejercicio propuesto: Determínese el tipo de Exploración en Profundidad específico, si Recorrido o Búsqueda In-Orden o Pre-Orden o Post-Orden, que define las siguientes operaciones sobre un AB y descríbase usando su esquema: `imagenEspecularDe`, que hace que el AB sobre el que se aplica sea la imagen especular de un AB `original` dado; `numeroHojas`, que obtiene el número de Hojas de un AB, y `borrarHojas`, que borra las Hojas de un AB; `toStringNivel`, que dado un número de nivel `k` obtiene un `String` con los datos de los nodos del nivel `k` de un AB; `minimo`, que obtiene como resultado el mínimo de los datos que componen un AB; `enQueNivel`, que dado un dato `x` obtiene el nivel de un AB en el que se encuentra el nodo que contiene a `x`.

Para concluir con la definición recursiva de la Exploración de un AB cabe recordar que sin la Pila de la Recursión sería imposible ejecutar cualquier método recursivo que la implemente. En otras palabras, es la gestión LIFO de los Registros de Activación asociados al conjunto de llamadas que origina la llamada más alta a uno de estos métodos la que permite que todos los datos del AB se procesen en Pre-Orden, Post-Orden o In-Orden; la realización de los siguientes ejercicios permitirá al alumno comprobar que esto es así, y también recordar el procedimiento iterativo específico que utiliza el intérprete Java para llevar a cabo dicha gestión LIFO.

Ejercicios propuestos:

1. Obténganse los árboles de Llamadas que se generan al aplicar sobre el AB de la figura las operaciones `toStringPreOrden()`, `toStringPostOrden()` y `toStringInOrden()`; para cada uno de ellos indíquese además dentro de un círculo el orden de generación de sus nodos y dentro de un cuadrado el de terminación.



2. Generalizando a partir del resultado del ejercicio anterior y suponiendo que se emplea un esquema recursivo para explorar el AB anterior en Profundidad, explíquese el motivo por el que su nodo Raíz, el primero al que se accede en la llamada más alta, se visita en último lugar durante una exploración en Post-Orden y en quinto lugar durante una en In-Orden; razónese también cómo es posible durante un exploración en Pre-Orden visitar su nodo Raíz en primer lugar y acceder y procesar su Hijo Derecho en sexto lugar.

2.3.2. Exploración por Niveles de un AB

La definición de la Exploración por Niveles de un AB requiere, como su nombre indica, un acceso por niveles a sus nodos: desde un nodo y nivel dados, el siguiente nodo a visitar es su Hermano o si no tiene, el primer descendiente en ese mismo nivel de algún antepasado. Como la definición recursiva de un nodo solo permite acceder desde él a uno de sus Hijos, nótese entonces que solo es posible plantear la descripción iterativa:

> la Exploración en Anchura de un AB se define como la exploración iterativa de la estructura Lineal que contiene la Secuencia de los nodos del AB ordenada por Niveles; de donde su esquema iterativo asociado de coste lineal no es otro que el conocido esquema de Exploración (Recorrido o Búsqueda) de una estructura Lineal.

Obviamente, si no se especifica el proceso de obtención de la estructura Lineal que contiene los nodos del AB ordenados por Niveles la definición que se acaba de dar resulta tan inútil como general. Sin embargo esta definición deja de ser general al especificar el proceso de obtención de tal estructura puesto que éste depende de la representación en memoria que se tenga del AB; específicamente, las implementaciones básicas de la Exploración en Anchura de un AB dado son las dos que se exponen a continuación: la primera es válida para la representación enlazada de un AB esbozada en el apartado anterior, y la segunda para cualquier AB en el que la inserción se realice por Niveles, y que por tanto admite una Representación Contigua en memoria.

Exploración por Niveles de un AB

La estructura recursiva del nodo Raíz de un AB hace imposible la definición recursiva de una Exploración en Anchura, además no es suficiente por sí sola para obtener la definición iterativa de cualquier tipo de Exploración de un AB, la de Anchura incluida; la solución pasa por utilizar, además, una memoria que haga las veces de la Pila de la Recursión, esto es una EDA Lineal que invariablemente, en todos y cada uno de los pasos de la iteración contenga convenientemente ordenada la Secuencia de nodos del AB que aún quedan por visitar a partir de uno dado, el nodo `actual` que se visita en cada iteración. Es más, el mismo procedimiento iterativo que permite al intérprete Java gestionar la Pila de la Recursión se puede aplicar para gestionar la EDA auxiliar y, por tanto, para visitar los `N` nodos del AB en el orden que determina el tipo de la Exploración, en la `i`-ésima iteración su `i`-ésimo nodo $\forall$`i:1`$\leq$`i`$\leq$`N`; nótese entonces que ni la Exploración ni su descripción iterativa se realizan sobre el propio AB a explorar, sino sobre la EDA auxiliar Lineal que contiene en cada momento la Secuencia de nodos del AB que aún quedan por visitar.

En el caso de la Exploración en Anchura que nos ocupa, la memoria auxiliar debe ser una Cola; nótese que los nodos de un nivel del AB, `1`$\leq$nivel$\leq$`H`, son los Hijos no vacíos de los nodos situados en el nivel anterior, empezando siempre por el situado más a la izquierda. Por tanto,

para que en la n-ésima iteración el primer nodo de la memoria auxiliar coincida con el n-ésimo nodo por Niveles del AB es necesario que los Hijos no vacíos de n se inserten, empezando por el Izquierdo, al final de esta memoria auxiliar; en otras palabras, es necesario que tenga una gestión FIFO o que sea una Eda Lineal de tipo Cola. La descripción algorítmica para la Exploración en Anchura sería:

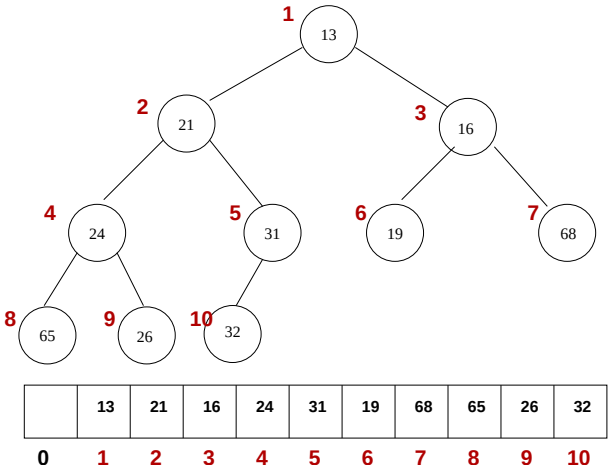
```
/** explora Por Niveles un AB  **/
void explorarPorNiveles(){
    Cola<NodoAB<T>> memoriaAuxiliar = new ArrayCola<nodoAB<T>>();
    memoriaAuxiliar.encolar(this.raiz);
    inicializarResultado;
    while ( !memoriaAuxiliar.esVacia() ){
        nodoAB<T> n = memoriaAuxiliar.desencolar();
        tratar(n.dato);
        if ( n.izq!=null ) memoriaAuxiliar.encolar(n.izq);
        if ( n.der!=null ) memoriaAuxiliar.encolar(n.der);
        combinar(resExplorar, resActual):
    }
}
```

Ejercicios propuestos:

1. Instánciese `explorarPorNiveles()` para describir la operación `toStringPorNiveles()`; tras comprobar que tanto su coste Temporal como el Espacial son exactamente del orden del tamaño del AB, realícese la traza de la ejecución de `arBin.toStringPorNiveles()` para comprobar su correcto funcionamiento.
2. Instánciese el esquema `explorarPorNiveles()` para describir la operación que, con perfil `void insertaPorNiveles(T x)`, inserta `x` por niveles en un AB, es decir lo inserta detrás del primero de los nodos del AB que aún carezca de alguno de sus Hijos o último nodo del AB según el orden que determina la Exploración por Niveles; analícese también el coste Temporal Asintótico del esquema de Búsqueda resultante.
3. Instánciese `explorarPorNiveles()` para describir `toStringPreOrden()` iterativamente; compárense entonces los costes Temporal y Espacial de las implementaciones iterativa y recursiva de esta operación e indíquese cuál es la más eficiente.

Exploración en Anchura de un AB Completo: Representación Implícita de un AB

Independientemente de la estructura de un AB, una idea alternativa para describir iterativamente, de forma natural y con un coste lineal la Exploración *in-situ* y en Anchura de un AB es imponer la inserción por niveles de sus datos, i.e. insertar cada nuevo dato detrás del primero de los nodos del AB que aún carezca de alguno de sus Hijos; esta idea, aunque en principio pueda parecerlo, no es extraña ni ajena a la solución del problema planteado: si el tipo de Exploración determina el orden o secuencia en el que se visitan los nodos de un AB entonces la inserción por Niveles de, uno tras otro, `N` datos en un AB permite situar sus nodos ordenados por Niveles ¡ en un simple `array` !, cada uno en la posición del `array` que corresponde a su orden de inserción como se ilustra gráficamente en la siguiente figura:



- la posición `cero` del `array` se deja libre para reflejar que la Raíz del AB no tiene Padre;
- el primer dato insertado en el AB (vacío entonces), el `13` que ocupa el primer nodo de su Recorrido por Niveles y Raíz del árbol, se sitúa en la posición `1` del `array`;
- el segundo dato insertado en el AB, el `21` que ocupa el segundo nodo de su Recorrido por Niveles e Hijo Izquierdo de la Raíz del AB, se sitúa en la posición `2` del `array`;
- el tercer dato insertado en el AB, el `21` que ocupa el tercer nodo de su Recorrido por Niveles e Hijo Derecho de la Raíz del AB, se sitúa en la posición `3` del `array`;
- ...
- Y así sucesivamente hasta situar el último dato insertado en el AB, el `32` que ocupa el `N`-ésimo y último nodo de su Recorrido por Niveles, en la posición `N` del `array`.

Como resulta trivial deducir a partir de lo anterior, un AB construido insertando sus datos por Niveles es Completo y, por tanto Equilibrado; pero además, cualquier operación de Exploración sobre él se puede describir utilizando el esquema iterativo de Recorrido o Búsqueda sobre el `array` que contiene la Secuencia ordenada por Niveles de sus datos, su último nodo ocupa siempre la posición `N` en dicho `array` y por tanto la inserción por Niveles (detrás del último nodo del AB Completo), el tamaño y la altura del AB se pueden estimar en un tiempo constante, su Raíz se ocupa siempre la posición `1` en dicho `array` por lo que se puede acceder a ella en un tiempo constante, ... En realidad, tanto la solución que se acaba de presentar al problema de la descripción iterativa de la Exploración en Anchura de un AB como las muchas ventajas que de ella se derivan no son más que consecuencias directas de una importante propiedad de un AB Completo: **se puede representar sin ambigüedad alguna almacenando el resultado de su Recorrido por Niveles en un `array`, su denominada representación Implícita**; a partir de esta propiedad en el Tema 4 de esta misma Unidad se definirá la Representación Contigua de un AB como base para la Representación de un Montículo Binario.

3. Condiciones efectivas para la Búsqueda Dinámica en un AB

El análisis del coste de sus operaciones de Exploración ha hecho obvio que un AB no constituye de per-se un soporte válido para la Búsqueda Dinámica que caracteriza a las Representaciones eficientes de las EDA Cola de Prioridad y Diccionario; sin embargo, imponiendo

determinadas condiciones a un AB, para ser más exactos una a su estructura y otra a la relación de orden que guardan entre sí sus datos, es posible obtener dos subtipos suyos que sí lo son: el árbol Binario de Búsqueda (ABB), que como se estudia en el próximo tema es un AB que soporta la Búsqueda Dinámica de cualquier dato `x` en un tiempo promedio logarítmico y, por tanto, puede representar tanto a una Cola de Prioridad como a un Diccionario, y el Montículo Binario o *Heap*, que como se estudia en el tema `4` de esta misma Unidad constituye «la» Representación de la EDA Cola de Prioridad al soportar la Búsqueda Dinámica del dato de máxima prioridad con mayor eficiencia temporal y espacial que la que proporciona un ABB.

Es más, estas dos condiciones para la Búsqueda Dinámica en un AB se pueden extraer de un resultado ya conocido: **si** un AB de tamaño `N` es Equilibrado **entonces** el coste Temporal Asintótico de cualquier operación de Exploración de su Camino Más Largo está acotado superiormente por `log N`, su altura máxima. De ahí que,

1. **Propiedad Estructural:** solo en un **AB Equilibrado** se puede plantear reducir el coste de la Búsqueda de un dato en el Peor de los Casos, lineal con `N`, al del Recorrido de su `CML`.
2. **Propiedad de Ordenación:** para poder restringir en el Peor de los Casos la Búsqueda en un AB Equilibrado al Recorrido de su `CML` es necesario haber establecido de antemano, *a-priori*, el único Camino del AB donde se puede encontrar el nodo que lo contiene o bien, mejor aún, la posición exacta de tal nodo en el AB; y para ello no queda otra alternativa que **imponer entre los datos del AB una cierta relación de orden**.

Nótese que una vez garantizada la eficiencia de la Búsqueda en un AB mediante la conjunción de estas dos propiedades, la inserción o el borrado de uno de sus datos no debe suponer su ruptura sino que, por el contrario, debe garantizar su mantenimiento; en otras palabras, para soportar la Búsqueda Dinámica un AB no solo debe cumplir las propiedades Estructural y de Ordenación sino que además debe definir sus operaciones de inserción y borrado como las Búsquedas del lugar exacto del AB donde luego se espera encontrarlo y, una vez encontrado, insertarlo o eliminarlo.