Tema 7 Estructuras avanzadas de datos: punteros, listas, colas, pilas y árboles

Contenido

1 Concepto de estructura de datos dinámica

Todas las estructuras de datos estáticas ocupan siempre la misma cantidad de memoria desde que se crean hasta que se destruyen.

Por **ejemplo**, cuando creamos un array de números enteros, tenemos que dimensionarlo, decir cuál es el número máximo de enteros que puede contener. Si lo dimensionamos para 20 números enteros, y sólo guardamos uno, las otras 19 posiciones del array estarán reservadas para contener un número entero. De hecho, estarán inicializadas por defecto a cero. Independientemente de que introduzcamos más o menos números enteros, siempre estamos utilizando toda la memoria reservada para los 20 números, con lo que se estará desperdiciando memoria.

Pero si por alguna causa en nuestro programa necesitamos guardar más de 20 números enteros, tenemos dos opciones:

- Para guardar el vigésimo primer número sería necesario crear un nuevo vector de al menos 21 posiciones, copiar los 20 números del antiguo vector al nuevo, añadir el número al vigésimo primero, y actualizar la referencia del antiguo vector para que apuntara al nuevo (con veintiún elementos).
- La otra opción es que el programa directamente no permita que nuestro vector pueda crecer más de lo inicialmente establecido.

En general, podemos decir que las **estructuras estáticas** de datos **no** hacen un **uso eficiente** de la memoria.

En cambio, con las estructuras de datos dinámicas en cada momento se usa exactamente la memoria que se necesita para almacenar los datos que contienen.

Si se inserta un nuevo elemento en la estructura, su tamaño crece justo en el tamaño que ocupa ese elemento, y si se elimina un elemento de la estructura, el tamaño de ésta disminuye exactamente en el tamaño que ocupaba el elemento eliminado.

En general las estructuras de datos dinámicas sí hacen un uso eficiente de la memoria, aunque casi siempre a costa de complicar los algoritmos de manipulación de estas estructuras (insertar, modificar, eliminar, buscar, procesar o listar elementos).

2 Punteros. (Referencias en Java)

En programación por **puntero** entendemos un tipo de dato que corresponde a una **dirección de memoria que a su vez referencia a un dato de otro tipo**. Una variable de tipo puntero **lo único que va a contener por tanto es una dirección de memoria** (una referencia) que será la que realmente contendrá el dato.

Al construir una estructura de datos usando punteros, podemos hacer que su tamaño cambie y se ajuste en cada momento de forma exacta al número de elementos que contiene.

- Un puntero apuntará a un **nuevo elemento de la estructura**, creado en cualquier lugar de memoria que estuviera libre, de forma que el tamaño de la estructura crece y la memoria ocupada por la estructura se ajusta al nuevo tamaño.
- Al eliminar un elemento de la estructura, liberamos la **memoria** que ocupaba ese elemento y hacemos que el puntero que lo conectaba a la estructura apunte a nulo, que es el valor que indica que no apunta a ninguna posición de memoria, a ningún objeto. De esta manera, el tamaño de la estructura disminuye, y la memoria que ya no usa queda libre. La memoria empleada por la estructura se ajusta nuevamente al tamaño de la estructura.

Pero ya hemos visto que **en Java no existen los punteros, sino las referencias**.

Ésta es una gran ventaja frente a otros lenguajes que sí usan punteros, ya que se simplifican mucho los programas y se hacen más seguros. **Una de las causas por las que resulta más fácil trabajar con referencias que con punteros es la existencia del recolector automático de basura ("garbage collector"). Cada vez que un objeto deja de tener alguna referencia que contenga su dirección, que lo apunte, no tenemos que hacer nada para liberar la memoria**

que ocupa. El recolector de basura lo detecta y se encarga de liberar la memoria que ocupaba el objeto de forma automática.

En los lenguajes que usan punteros, no existe el recolector automático de basura, y el programador tiene que ser especialmente cuidadoso con la eliminación de los objetos. Para que la eliminación sea correcta, además de "desenganchar" el objeto haciendo que ningún puntero lo apunte hay que eliminarlo. Si no lo eliminamos, ese objeto se quedara ocupando permanentemente la memoria, sin posibilidad de usarlo, pero también sin posibilidad de utilizar la memoria que ocupa. Por eso, además de desengancharlo de la estructura, hay que conservar un puntero que nos diga dónde está esa memoria, para ejecutar alguna instrucción específica que libere esa zona de memoria. **De lo contrario, nuestro programa puede tener una "fuga de memoria" ya que se irá llenando de basura que nunca se recoge, hasta que llegue un momento en que no quede espacio en memoria para nada más, incluyendo la ejecución de nuestro programa, que se bloquearía o abortaría.**

3 Listas enlazadas

Son muchas las situaciones de la vida cotidiana en las que trabajamos usando **listas**:

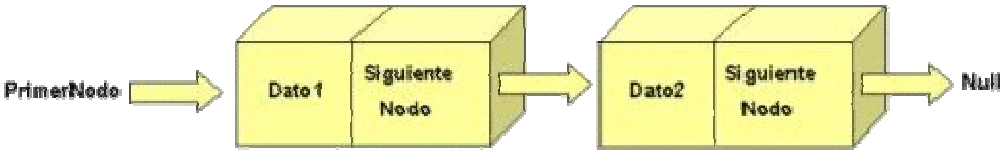
- Listas de alumnos.
- Listas de clientes.
- Listas de espera, con prioridad (espera de un órgano para un trasplante, según gravedad del enfermo) o sin prioridad (por orden de llegada, como en la cola del supermercado).
- Etc.

Por eso, cualquier lenguaje de programación debe permitir la creación y gestión de listas de muy diversos tipos.

Java permite construir listas enlazadas de cualquier tipo de elemento u objeto, a condición de que el tipo de ese elemento haya sido definido adecuadamente en una clase, o por el propio lenguaje.

Una lista enlazada es una colección de elementos colocados secuencialmente uno detrás de otro (y un array también). Por lo general, **los nodos de las listas enlazadas no se almacenan contiguamente en memoria**, sino que son adyacentes en forma lógica. Las grandes diferencias con un array son:

- **En la lista enlazada cada elemento debe saber dónde está guardado en memoria el siguiente elemento de la lista.**
- **El último elemento de la lista debe indicar de alguna forma que detrás de él no hay ningún elemento más.**
- **Debemos tener una forma de llegar hasta el primer elemento de la lista, para a partir de él, poder recorrerla pasando al siguiente elemento, hasta llegar al último de la lista.**



En un array siempre sabemos que el siguiente elemento está en la siguiente posición del array, que se obtiene sumándole uno a la posición actual. No es necesario que lo sepa el propio elemento si lo sabe el programador. Sabemos que el primer elemento está en la posición cero y que el último está en la posición length - 1. Por ahora no se aprecia una gran diferencia ni una mejora sustancial de las listas respecto a los arrays. Pero sí la hay, y es la siguiente:

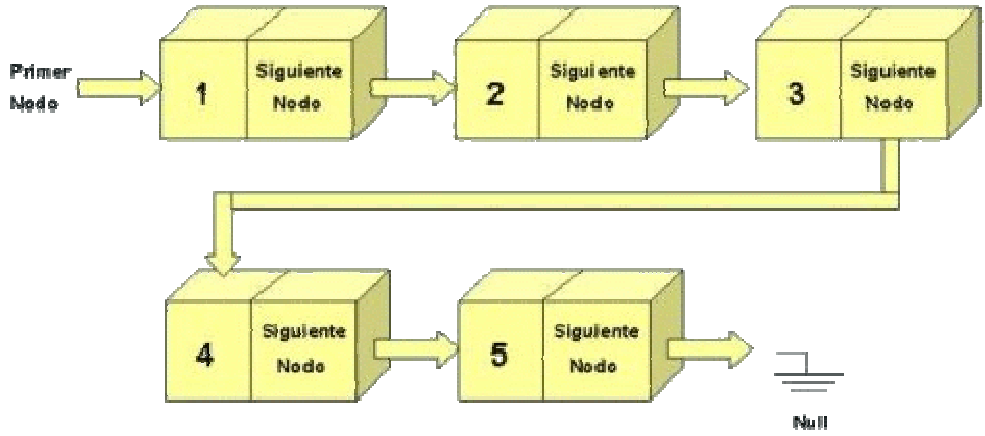
En la lista enlazada podemos tener cualquier número de elementos, desde cero hasta los que quepan en la memoria, y su tamaño crece y disminuye en tiempo de ejecución cada vez que se inserta o elimina un elemento de la lista, según las necesidades. La lista ocupa en cada momento exactamente el espacio que necesita.

3.1 Implementación en Java

Veamos cómo construir una lista enlazada en Java.

- Una lista enlazada consta de una serie de elementos llamados **nodos** (La **estructura de un nodo** se define en una **clase**, que podemos llamar **Nodo**).

- Cada nodo tiene dos partes bien diferenciadas, a modo de campos:
    - **Un dato** (o conjunto de datos) que contiene la información que realmente queremos almacenar en la estructura, y que podrá ser de cualquier tipo.
    - **Una referencia**, que podríamos llamar **`siguienteNodo`**, que enlaza con el siguiente nodo de la lista (referencia a objetos de tipo `Nodo`).
- Dispondremos de una referencia (**referencia a objetos `Nodo`**) que apunte siempre **al primer nodo de la lista,** llamado **cabecera de la lista, o `primerNodo`.** A través de él podremos acceder a todos los nodos de la lista sin más que seguir los enlaces `siguienteNodo`.
- Si la **lista** está **vacía**, la referencia **`primerNodo` apuntará a null**.
- Sabremos que hemos alcanzado el **nodo final** porque la referencia **`siguienteNodo`** del último nodo de la lista valdrá **`null`.**



Si quisiéramos hacer una lista enlazada de nodos para guardar números enteros, por ejemplo, lo primero sería definir adecuadamente la clase **`Nodo`**:

```
class Nodo {
	int dato;
	Nodo siguienteNodo;

	public Nodo(int numero){
		dato=numero;
		siguienteNodo=null;
	}
}
```

El otro paso será definir la **referencia** que permita acceder al primer nodo de la lista. En principio, se hará en la clase que cree y gestione la lista enlazada, en vez de en la propia clase **`Nodo`**, aunque esto último también es posible. La definición es simple, debe ser una referencia de tipo **`Nodo`**. Cada lista enlazada va a tener un campo que va a ser una referencia al primer Nodo de la lista.

```
	Nodo primerNodo=null;
```

Realmente podemos ver la variable **`primerNodo`** como la referencia que apunta no sólo al primer nodo de la lista, sino a toda la lista enlazada. Es como si para nosotros la lista enlazada fuera "su nodo inicial". Por eso la inicializamos a **`null`**, para indicar que la lista está inicialmente vacía.

Los campos de **`Nodo`** se han definido con nivel de acceso o visibilidad por defecto (**package**), que significa que serán accesibles desde todas las clases del mismo paquete. Así conseguimos que se puedan modificar con los métodos que realizan las operaciones sobre la lista, que estarán en otras clases del mismo paquete.

En principio, la referencia **`siguienteNodo`** la inicializamos a **`null`** en el constructor, ya que serán los métodos que inserten nodos en la lista los que se ocuparán de darle el valor adecuado. Cuando decimos que cada **`Nodo`** tiene un campo de tipo **`Nodo`** da la impresión de que estamos diciendo que cada objeto **`Nodo`** lleva dentro otro objeto **`Nodo`** completo. Pero no es así, lo que realmente se indica es que cada nodo lleva dentro una referencia con la

dirección de memoria en la que habrá otro nodo, que será el siguiente de la lista. Cada nodo guarda no al siguiente nodo, sino la dirección de memoria en la que se le puede encontrar.

Existen múltiples formas de crear una lista enlazada, y sería posible definirla en una sola clase o en varias, de forma que sus elementos estuvieran ordenados o no, etc.

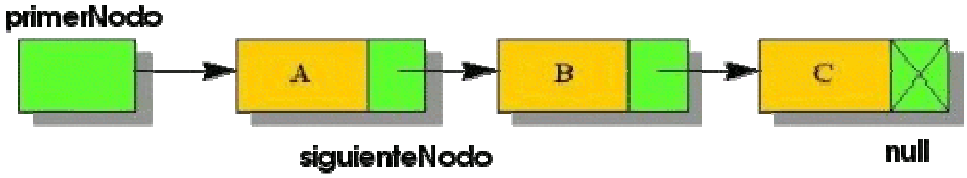
3.2 Listas enlazadas de un tipo concreto de elemento o listas genéricas

En el ejemplo anterior hemos comenzado a ver cómo definir la clase **`Nodo`** y hemos supuesto que lo que queremos almacenar son números enteros. Pero realmente es posible que necesitemos formar listas enlazadas de cualquier tipo de elementos.

Si queremos formar una lista enlazada de objetos **`Persona`**. Tenemos dos formas de hacerlo:

- **Definir la lista enlazada como una estructura de datos "concreta"**, en la que definimos el tipo de los nodos, que serán objetos de tipo **`Persona`** a los que se les añade un campo **`personaSiguiente`**, y definir en esa misma clase todas las operaciones de manejo básico de listas enlazadas. Si en el futuro queremos hacer una lista enlazada de otro tipo de elementos, por ejemplo una lista de **`Vehículos`**, tendremos que escribir todo el código de nuevo para el nuevo tipo de lista enlazada. Es decir, podemos construir la lista enlazada y definir las operaciones sobre ella de forma fuertemente ligada al tipo de los datos que realmente queremos almacenar.
- **Definir la lista enlazada como una estructura de datos abstracta,** como una lista genérica. Encapsularemos los detalles del tipo de datos concreto que queremos almacenar en la lista enlazada dentro de una clase **`Nodo`**, y crearemos otra clase **`Lista`** que definirá todas las operaciones a realizar con una lista enlazada de **Nodos**, sea cual sea el tipo de dato que encierran esos nodos. De esta manera, la clase que crea y gestiona la lista enlazada es totalmente independiente del dato concreto que se quiera almacenar. Habremos creado un tipo abstracto de dato "**`Lista`**", que podremos usar en cualquier problema que necesite gestionar una lista enlazada.

  Además de reutilizar este código, nos aseguramos de que en los futuros programas no tendremos que acordarnos de cómo se implementan cada una de las operaciones básicas con listas enlazadas, sino que sencillamente las usaremos con la garantía de que funcionan.



Además el hecho de encapsular los detalles del tipo que se quiere almacenar en la lista dentro de la clase **`Nodo`** es bastante fácil de realizar:

- **Hay que modificar en la clase `Nodo` las referencias a un tipo (a una clase) por referencias a otro tipo (otra clase)**. Por ejemplo, en la clase **`Nodo`** cambiar **`Persona`** por **`Vehículo`**, y ya tendremos una lista enlazada de vehículos en vez de una lista enlazada de personas.

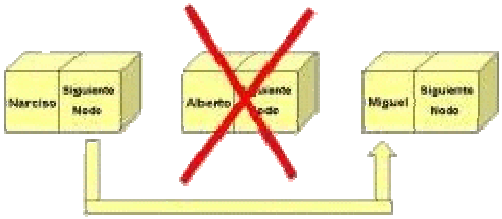
  O, mejor aún, podemos usar la clase **`Object`** para que sea el tipo del campo **`dato`** de la clase **`Nodo`**, con lo cual valdría para almacenar cualquier tipo de datos en la lista. Sería posible hasta tener distinto tipo de dato en cada nodo de la lista, ya que en Java cualquier cosa es un **`Object`**.
- **Cualquier clase que queramos usar como tipo para los elementos de la lista enlazada debe declararse de forma que incluya las siguientes definiciones:**
  - Debe **implementar** la **interfaz `Serializable`**, para que los datos de la lista se puedan guardar cómodamente en un fichero en disco. (**Sólo si se quieren almacenar los datos de forma permanente**).
  - Debe **sobreescribir el método `toString()` de forma adecuada** para definir cómo queremos que se escriban los datos de cada nodo al hacer un listado.

- Debe **sobrecargar el método `equals()` de forma que se establezca bajo qué condiciones se considera que los datos de dos nodos son iguales**, con el fin de que podamos buscar un elemento concreto dentro de la lista, y comparar cada elemento de la lista con el elemento buscado.
- Debe **sobrecargar el método `compareTo()` de forma que nos permita saber si el dato de un nodo es mayor, menor o igual que el de otro** (Sólo si la lista enlazada debe estar ordenada). Al insertar, para que la lista esté ordenada, deberemos encontrar el lugar adecuado para que se mantenga el orden, por lo que tendremos que ir comparando el nodo a insertar con todos los de la lista, para saber si el que se quiere insertar va delante o detrás del que vamos "visitando" en cada momento, hasta que encontremos el lugar adecuado para la inserción. También resultará útil en las búsquedas, ya que podremos abandonar la búsqueda sin necesidad de haber recorrido toda la lista cuando el elemento visitado sea mayor que el que buscamos (suponiendo que la lista está ordenada de menor a mayor).

Estudia como ejemplo el código de los proyectos `ListaEnlazada` y `ListaPersonas`. La primera lista es de números enteros y la segunda de objetos de la clase `Persona`. Ninguna de las dos está ordenada, y se hacen dos inserciones al principio, dos inserciones al final, dos borrados al principio y dos borrados al final.

3.3 Operaciones básicas en listas ordenadas y no ordenadas

Las operaciones básicas con listas enlazadas son:



- **ALTA: Inserción de nodos nuevos en la lista**, al principio o al final, si la lista está desordenada, o en el lugar que corresponda, si debe estar ordenada.
- **BAJA: Eliminación de un nodo de la lista**.
- **REINICIAR: Eliminación de la lista completa**. En Java es muy simple, ya que basta con igualar a `null` la referencia al primer nodo de la lista, y el recolector de basura hará el resto. En otros lenguajes que no disponen de recolector de basura, sería el programador el que tendría que cuidarse de recorrer toda la lista liberando la memoria que ocupa cada uno de los nodos de la lista.
- **BÚSQUEDA: Búsqueda o consulta de los datos de un nodo** concreto de la lista.
- **PROCESAMIENTO: Recorrido y/o procesamiento completo de la lista**.
- **LISTADO COMPLETO: Listado o consulta de todos los nodos de la lista.** Los listados son realmente un caso especial de procesamiento o recorrido de la lista.
- **LISTADO PARCIAL: Listado de los nodos que cumplan una condición**.
- **CONSULTA Nº NODOS: Consultar el total de elementos que contiene la lista**. No es imprescindible en realidad, siempre puede recorrerse la lista para contarlos, pero disponer de esa facilidad puede ser útil.
- **MODIFICAR: Modificar los datos de un nodo de la lista**.
- **GUARDAR: Guardar todos los datos de la lista en un fichero**. No es una operación sobre la lista propiamente dicha, pero es habitual que la lista se use para gestionar información, y que queramos guardar esa información de forma permanente en un fichero para poder recuperarla más tarde.
- **CARGAR: Cargar desde un fichero los datos de la lista**. Si disponemos de la posibilidad de guardar la información de la lista en un fichero, debemos tener la posibilidad de recuperarla, leyendo esa información desde el fichero.

3.4 Otros tipos de listas enlazadas

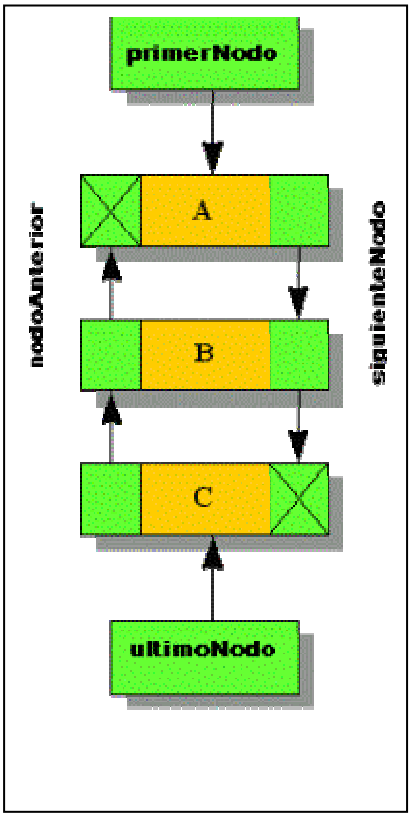
Las listas enlazadas son estructuras de datos bastante versátiles, y es bastante fácil hacer modificaciones y adaptaciones para adaptarlas mejor a muchos problemas y situaciones.

Por ejemplo, en lugar de permitir que en la lista enlazada sólo se pueda avanzar hacia adelante, puede ser interesante realizar la búsqueda de un elemento que sospechamos que será de los últimos de la lista comenzando por el final y movernos hacia el principio. De esta manera la probabilidad de encontrarlo rápidamente aumenta. Sobre todo si la lista enlazada contiene gran cantidad de nodos, ese ahorro de tiempo puede resultar interesante.

Para conseguir esa funcionalidad podemos construir una **lista doblemente enlazada**.

3.4.1 Listas doblemente enlazadas

¿Qué diferencias tiene una lista doblemente enlazada con respecto a una lista enlazada "simple"? ¿Cómo es la estructura de una lista doblemente enlazada?



- Una lista doblemente enlazada está compuesta por una serie de elementos llamados nodos (objetos de la clase `Nodo`).
- Cada nodo dispone de tres partes bien diferenciadas:
  - Una zona de datos, que podrá ser de cualquier tipo, y que contendrá la información que queremos almacenar en la lista.
  - Una referencia al siguiente elemento de la lista, que será de tipo Nodo. Podemos llamarla `siguienteNodo`.
  - Una referencia al elemento anterior en la lista, que será de tipo Nodo. Podemos llamarla `nodoAnterior`.
- Habrá una referencia de tipo `Nodo` que apunte permanentemente al primer nodo de la lista. Podemos llamarla `primerNodo`.
- Habrá una referencia de tipo `Nodo` que apunte permanentemente al último nodo de la lista. Podemos llamarla `ultimoNodo`.
- Si la lista está vacía, tanto `primerNodo` como `ultimoNodo` apuntarán a `null`.
- El último nodo de la lista tendrá a `null` su campo `siguienteNodo`.
- El primer nodo de la lista tendrá a `null` su campo `nodoAnterior`.

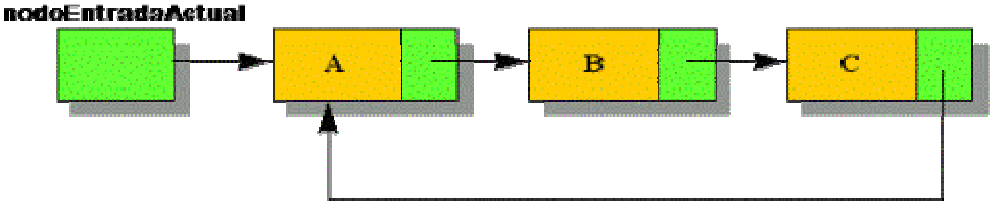
Todas las operaciones de **inserción y borrado** de elementos en una lista doblemente enlazada se complican un poco al tener que cuidar de la conexión adecuada de más enlaces y referencias. Esa complicación extra sólo se verá compensada si la estructura representa mejor la realidad de nuestro problema, o si se consigue alguna mejora en la eficiencia.

La mejora en la eficiencia puede venir derivada del hecho de que previamente conozcamos alguna información sobre un dato a la hora de buscarlo, borrarlo o insertarlo en la lista, de forma que podamos sospechar a priori que comenzando a buscar por uno de los dos extremos se va a encontrar antes su posición. Por **ejemplo**, en una lista doblemente enlazada y ordenada alfabéticamente por el nombre de doscientas mil personas, si tenemos que buscar a Ylenia, seguro que mejora la eficiencia del algoritmo si empezamos la búsqueda por el final y vamos retrocediendo en la lista. Pero por ejemplo, para María, ya no está tan claro por qué extremo de la lista interesa comenzar la búsqueda, dependerá de los nombres concretos que almacena la lista. Además, la mejora de la eficiencia si el número de personas en la lista es de doscientas en lugar de doscientas mil, no compensa el esfuerzo adicional de implementar este tipo de lista.

Pero supongamos que queremos hacer una aplicación que nos muestre los datos de cada persona en una pantalla a modo de ficha, con la posibilidad de mostrar en esa pantalla la persona anterior o la siguiente de la lista, desplazándonos con dos botones anterior y siguiente, con posibilidad de ir directamente a la última persona de la lista o a la primera. En este caso, una lista doblemente enlazada se ajusta perfectamente a la realidad de nuestro problema, mejor que una lista enlazada simple, por lo que será la estructura que usaremos, sin que importen las consideraciones de eficiencia.

3.4.2 Listas circulares enlazadas

Los nodos de la lista se pueden organizar de muy diversas formas según el propósito.



Podemos tener una situación en la que un programa tenga que ir comprobando periódicamente y de **forma cíclica** los datos de todos los nodos de la lista para realizar alguna operación, o comprobación, de forma que una vez que ha revisado todos tiene que volver a empezar por el principio. Si la lista es circular, esto no requiere ninguna operación especial. Empezar por el principio sigue siendo avanzar al siguiente elemento.

Para este tipo de situaciones puede ser interesante guardar los datos en una **lista circular enlazada**.

- **Una lista circular enlazada es realmente una lista enlazada "simple" en la que nos encargamos de que la referencia al nodo siguiente del último nodo de la lista apunte al primer nodo de la lista.** De esta manera, el siguiente del último nodo vuelve a ser el primer nodo de la lista, por lo que realmente no existe ni un primer nodo ni un último nodo.
- **No obstante seguimos necesitando una referencia a un nodo de la lista para poder acceder a ella por algún punto.** Podríamos llamar a esa referencia **`nodoEntradaActual`** (es actualmente el nodo de entrada a la lista).

Por ejemplo, podemos pensar en que nuestra lista de personas representa realmente los datos de los **socios de un club** deportivo comunitario, en el que:

- pueden inscribirse nuevos socios para hacer uso de las instalaciones,
- o darse de baja aquellos que ya no deseen seguir siendo socios.
- los estatutos del club establecen que habrá un presidente, nombrado sólo por un trimestre, para que la rotación se agilice.
- También fijan un orden para el relevo de la presidencia. Ese orden es la antigüedad en el club, de forma que los nuevos socios tardarán en asumir la presidencia el mayor tiempo posible.

De esta manera se les permite adquirir un buen conocimiento del funcionamiento del club antes de su turno. Por tanto, el último socio en inscribirse será el último en asumir la presidencia. Si en algún momento se completa la lista de socios de forma que todos han sido presidentes, le tocará el turno al que lo fue hace más tiempo.

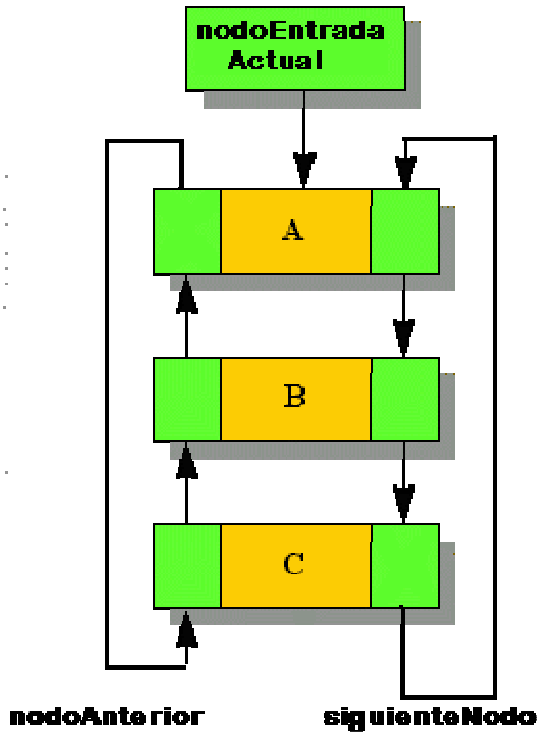
Manteniendo los datos en una lista circular es muy fácil saber cuál es el siguiente presidente del club. Basta con que la referencia de entrada a la lista circular, nodoEntradaActual apunte al presidente del club. Cada vez que hay que nombrar un nuevo presidente, basta con avanzar esa referencia al siguiente elemento de la lista. Los nuevos socios se insertarán justo delante del presidente, de forma que habrá que completar toda una vuelta hasta que les toque el turno a ellos**. Antes de que le toque el turno a un nuevo socio habrán tenido que ser presidentes todos los que eran socios del club en el momento que se inscribió**. De esta manera, además, los socios del club están ordenados según el tiempo que les falta para asumir de nuevo la presidencia. Si un presidente causa baja en el club, la presidencia pasará al siguiente socio de la lista. En todo momento debe haber un presidente del club, salvo que el club no tenga ni siquiera un socio.

3.4.3 Listas circulares doblemente enlazadas

También podemos trabajar con listas circulares doblemente enlazadas, en las que cada nodo tuviera un enlace al elemento anterior y un enlace al elemento siguiente de la lista, aunque su uso no es muy frecuente.

Para su implementación habría que tener en cuenta las siguientes cuestiones:

- Una **lista circular doblemente enlazada** es básicamente una lista doblemente enlazada en la que tenemos cuidado de que la referencia **`siguienteNodo` del último nodo de la lista apunte siempre al primer nodo de la lista** (el siguiente del último vuelve a ser el primero) y de que la referencia **nodoAnterior del primer nodo de la lista apunte siempre al último nodo de la lista** (el anterior del primero vuelve a ser el último).
- Eso significa que realmente no hay un primer nodo ni un último nodo, aunque seguimos necesitando una referencia a

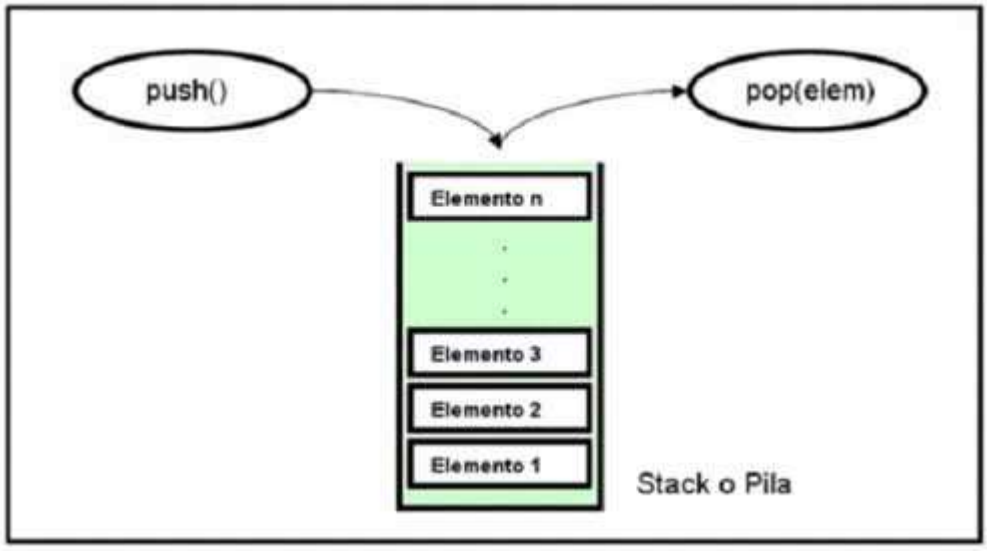


algún nodo de la lista que nos permita acceder a sus nodos. Esa referencia podría llamarse **`nodoEntradaActual`**.

- Si la lista está vacía **`nodoEntradaActual`** debe apuntar a **`null`.** Si no apunta a `null`, es porque al menos hay un nodo en la lista.

4 Pilas

Una pila (stack en inglés) es una estructura de datos lineal que solo tienen un único punto de acceso fijo por el cual se añaden, eliminan o se consultan elementos. El modo de acceso a los elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir).

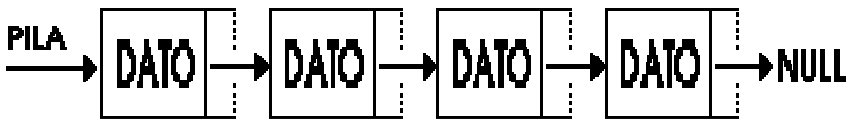


Las características de una pila son:

- Una pila es una estructura de datos secuencial, una serie de datos.
- Los datos se almacenan en la estructura por orden de llegada.
- Sólo se puede insertar un nuevo elemento en la estructura por un extremo, llamado **cima** o cabecera.
- La operación de añadir un nuevo elemento en la pila se suele llamar "push", ya que es como empujar a todos los elementos hacia abajo para meter uno nuevo en la cima.
- Sólo se pueden extraer elementos de la estructura por un extremo, que es justamente el mismo extremo por el que se introducen, la cima o cabecera.
- La operación de extraer (eliminar) un elemento de la estructura suele llamarse "pop".
- La otra operación asociada a la pila es preguntar si está vacía o no.
- Puede ser necesario, o no, disponer de una operación para saber si la pila está llena o no.

4.1 Implementación de una pila en Java

Para implementar esta estructura de datos hay más de una opción.



La implementación de pilas puede hacerse:

- **Con arrays**, manteniendo en todo momento en una variable la posición del índice donde está el último elemento introducido (o el primer hueco si se prefiere) Impone un tamaño máximo para la pila y obliga a reservar espacio de memoria para todos los posibles elementos de la pila, aunque no se usen. Al tener un tamaño máximo obliga a implementar también una operación para preguntar si la pila está llena o no.

- **Con listas enlazadas**, en las que cada nuevo nodo pasa a ser el primero de la lista, y donde sólo se puede borrar el primer nodo de la lista. No impone límite al número de elementos y aprovecha eficientemente la memoria, al ser una estructura de datos dinámica. Una pila es inherentemente dinámica, por lo que esta implementación es a priori más natural.

Las operaciones que debe tener disponibles el tipo Pila son las siguientes:

Operaciones básicas con pilas	
Operación	Descripción
Añadir (push)	Insertar un nuevo dato en la cabecera de la pila.
Eliminar (pop)	Sacar un dato de la cabecera de la pila.
Pila Vacía	Comprobar si la pila está vacía.
Pila llena	Comprobar si la pila está llena. Sólo si se usa un array para implementarla.
Vaciar pila	Sacar todos los elementos y dejar la pila vacía.
Cima	Obtener el elemento de la cima de la pila.
Tamaño máximo	Número máximo de elementos que puede contener. Sólo si se usa un array para implementarla.
Elementos	Consultar el número de elementos que tiene la pila.

5 Colas

El concepto de cola es algo habitual en la vida diaria:

- La cola del supermercado, o de la carnicería o del pan.
- La cola del autobús.
- etc.

Si pensamos en la cola de personas en espera de ser **operados**, lo normal es que tenga más prioridad aquella persona cuya vida corra peligro que aquella persona que tiene trastornos que no suponen un riesgo vital. Aunque la persona cuya vida corre peligro acabe de llegar a la cola, le será asignada una alta prioridad, de forma que adelante a todos o casi todos los que ya están en la cola.

¿Cómo gestionar una cola de prioridades? La forma más simple es utilizar una lista ordenada por prioridad de colas. Siempre se atenderán antes los elementos de la primera cola, la de máxima prioridad. Si esta cola está vacía, se atenderá la siguiente cola, de la que se irán sacando elementos por el principio, hasta agotarla. Antes de sacar un elemento de una cola será imprescindible comprobar que no hay ningún nuevo elemento en ninguna de las colas de mayor prioridad. De esa manera, el último elemento de la última cola (la de menos prioridad) sólo será "atendido" (eliminado de la cola) cuando todos los demás elementos de todas las demás colas sean atendidos, es decir, cuando sea el único elemento de toda la cola de prioridades.

Para gestionar toda esta información, es útil almacenarla justamente en una estructura de datos cuyo funcionamiento sea semejante al funcionamiento de una cola real. Y por eso esa estructura se llama también cola.

5.1 Especificaciones de una cola

Es una estructura de datos secuencial, en la que **los datos están almacenados en estricto orden de llegada**, de forma que **el último introducido es el último de la cola**, en la que **los elementos se introducen siempre en el mismo extremo de la lista (final de la cola)**, y **se extraen siempre del extremo contrario de la lista (cabecera de la cola)**. Es una **estructura de datos tipo FIFO** (First In- First Out, el primero que llega es el primero en salir).

De forma más esquemática vamos a ver cuáles son las características de una cola:

- Una cola es una estructura de datos secuencial, una serie de datos.
- Los datos se almacenan en la estructura por orden de llegada.
- Sólo se puede añadir un nuevo elemento en la estructura por un extremo, el final de la cola.

- Sólo se pueden extraer elementos de la estructura por un extremo, que es justamente el extremo contrario, el comienzo de la cola, llamado cima, frente o cabecera.
- La otra operación asociada a la cola es preguntar si está vacía o no.
- Puede ser necesario, o no, disponer de una operación para saber si la cola está llena o no. Será necesario si la cola la implementamos mediante arrays, y no lo será si la implementamos mediante nodos enlazados por referencias.

5.2 Implementación de una cola en Java

Al igual que ocurría con las pilas, disponemos de más de una forma de implementar una cola, cada una con sus ventajas e inconvenientes, según el uso que vayamos a hacer de la cola.

La implementación de colas puede hacerse:

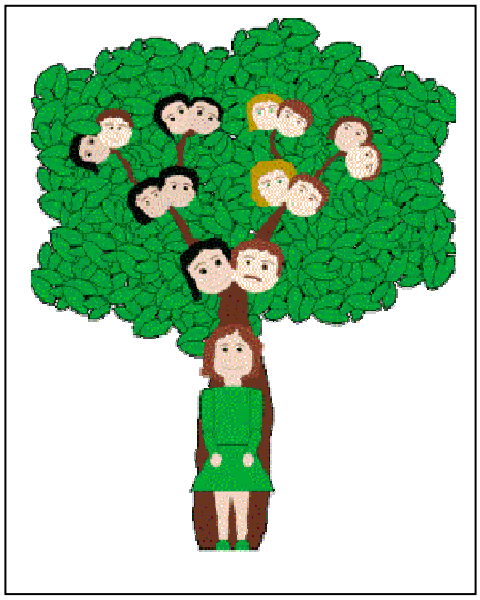
- **Con arrays**, manteniendo en todo momento en una variable `finalCola` la posición del índice donde está el último elemento introducido (o el primer hueco si se prefiere) y otra variable `frenteCola` en la que se indique dónde está el primer elemento, que será el próximo en ser retirado de la cola. Esta implementación impone un tamaño máximo para la cola y obliga a reservar espacio de memoria para todos los posibles elementos de la cola, aunque no se usen. Al tener un tamaño máximo obliga a implementar también una operación para preguntar si la cola está llena o no.
- **Con listas enlazadas**, en las que cada nuevo nodo pasa a ser el último de la lista, y donde sólo se puede borrar el primer nodo de la lista. No impone límite al número de elementos y aprovecha eficientemente la memoria, al ser una estructura de datos dinámica. Una cola es inherentemente dinámica, por lo que esta implementación es a priori más natural.

Las operaciones que debe tener disponibles el tipo Cola son las siguientes:

Operaciones básicas con colas	
Operación	Descripción
Añadir	Insertar un nuevo dato en la cabecera de la cola.
Eliminar	Sacar un dato del final de la cola.
Cola Vacía	Comprobar si la cola está vacía.
Cola Llena	Comprobar si la cola está llena. Sólo si se usa un array para implementarla.
Vaciar cola	Sacar todos los elementos y dejar la cola vacía.
Cabecera o Frente	Obtener el elemento de la cabecera de la cola.
Final	Obtener el elemento del final de la cola.
Tamaño máximo	Número máximo de elementos que puede contener. Sólo si se usa un array para implementarla.
Elementos	Consultar el número de elementos que tiene la cola.

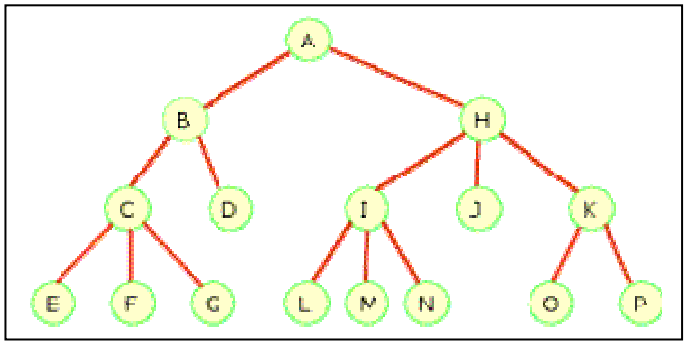
6 Árboles

Hasta ahora las estructuras de datos que hemos visto eran **lineales**, tanto en el caso de los arrays, como las listas enlazadas, las colas o las pilas. Los elementos de la estructura estaban unos detrás de otros, de forma que para cada elemento, había un único elemento que es el siguiente y un único elemento que es el anterior.



Pero en la vida real existen situaciones en las que las relaciones entre unos elementos y otros se establecen de una manera más jerárquica, de forma que cada elemento de la estructura "depende" de otro jerárquicamente superior, etc. Podemos encontrar bastantes ejemplos:

- El árbol genealógico en el que cada persona tiene dos padres, que a su vez tienen dos padres, etc. y cada persona tiene varios hijos, que a su vez tienen varios hijos, etc.
- La lista de carpetas y archivos que contiene el disco duro, que puede considerarse como una carpeta que a su vez contiene más carpetas y archivos, que a su vez contienen otras carpetas, etc.
- etc.



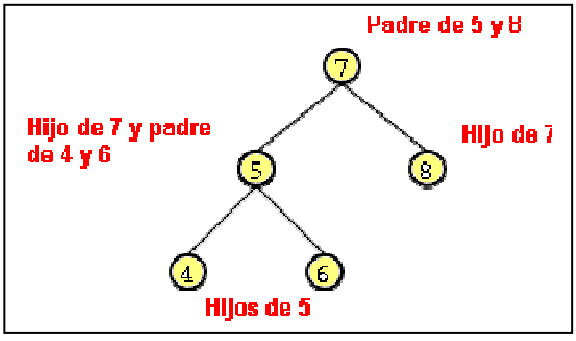
Para representar de forma conveniente las relaciones que existen entre los datos que se quieren almacenar, resulta útil disponer de una estructura de datos que represente esa relación de jerarquía o dependencia. **Esa estructura jerárquica de datos son los árboles.** (Se llaman árboles por que pueden representarse como árboles invertidos en los que hay un elemento raíz, que representa el tronco principal, a partir del cual van surgiendo las ramas que llevan a los demás elementos)

Existen también situaciones en las que los datos no se relacionan entre sí de forma jerárquica, pero conviene almacenarlos en un árbol como si lo hicieran, con el propósito de mejorar la eficiencia de los algoritmos de búsqueda, inserción o eliminación. Este es el caso de los árboles de búsqueda. Por **ejemplo**, un árbol binario (de cada nodo parten dos ramas) que esté ordenado, de forma que dado un nodo todos los de su izquierda sean menores, y todos los de la derecha mayores. A la hora de buscar un elemento, lo comparamos con la raíz del árbol, de forma que si ese elemento es menor que el que hay en la raíz tendremos que seguir buscando en la rama izquierda, con lo que ya hemos evitado comparar con todos los elementos de la rama derecha, ya que todos son mayores que el raíz. Y en cada nueva rama podremos descartar una de las dos sub-ramas. **Con eso conseguimos que el número de comparaciones necesarias para encontrar nuestro elemento se disminuya drásticamente.**

6.1 Especificación de un árbol

La nomenclatura que se usa para especificar lo que es un árbol se parece a la terminología familiar de un árbol genealógico. Eso es fruto de la relación entre esta estructura de datos y la forma de relacionarse jerárquicamente que los datos tienen en muchas situaciones.

Un árbol puede definirse de la siguiente forma:

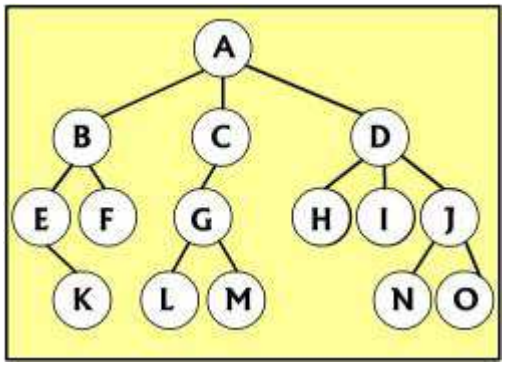


- Es un conjunto, eventualmente no vacío, de elementos del mismo tipo llamados nodos.
- De entre todos los nodos existe un **nodo** "distinguido" al que llamamos **raíz** del árbol, de forma que dispondremos de una referencia que permanentemente apunte al nodo raíz.
- Los restantes elementos del árbol forman una colección

de cero o más **subárboles disjuntos entre sí** (los subárboles no comparten nodos)

- A los sucesores inmediatos de un nodo N se les llama **hijos** de N, y N se dice que es su **padre**. A los hijos de un mismo nodo se les llama **hermanos**.
- **Cada nodo contendrá** una parte de **datos**, que incluirá toda la información que realmente se quiere almacenar en la estructura, **y varias referencias a** cada arista de la que puede colgar un subárbol (cada uno de **los posible nodos hijos**). Esas referencias se usan para construir y mantener la estructura.
- Los nodos que no tienen hijos se llaman **hojas**.
- Un camino desde la raíz a una hoja se llama **rama**.
- El final de una rama se marca poniendo a `null` su referencia.
- La **profundidad o altura** de un árbol es el número de nodos que contiene la rama más larga.
- Cada nodo del árbol tiene asignado un **número de nivel** de la siguiente forma:
    - El nodo raíz tiene número de nivel 0.
    - Un nodo N distinto del raíz, tiene nº de nivel una unidad superior al de su padre o antecesor.

La definición recursiva anterior pone de manifiesto que los árboles son estructuras inherentemente recursivas, y por tanto, cabe esperar que los algoritmos de manejo de árboles también sean inherentemente recursivos.



Hay un problema con el número máximo de hijos o subárboles que puede tener cada nodo de un árbol.

Si cada nodo puede tener un número grande de hijos o subárboles, (pongamos más de 10, por ejemplo) puede resultar muy incómodo incluir tantos campos en cada nodo y asignarle identificadores distintos a cada uno. En este caso hay dos posibilidades:

- **Si el número de subárboles posibles, aunque alto, es más o menos fijo y conocido**. En este caso cada nodo incluye:
    - El campo (o campos) de datos.
    - Un vector de referencias a los posibles subárboles.
- **Si el número de subárboles posibles además de alto es desconocido**, pudiendo variar en un rango muy amplio (desde no tener ningún hijo hasta tener cientos o incluso miles de ellos): usar un vector sería muy ineficiente, ya que nos obligaría a reservar un espacio fijo para miles de referencias, que en la mayoría de los casos seguramente no se necesitarían. En este caso es preferible que cada nodo incluya:
    - El campo (o campos) de datos.
    - Una referencia al primer hijo de ese nodo.
    - Una referencia al siguiente hermano de ese nodo.

De esta forma, la referencia al primer hijo es en realidad la referencia al primer nodo de una lista enlazada de hijos, que podrá contener tantos elementos como se quiera. Cada nodo tendrá tantos hijos como se necesite, y sólo se ocupará el espacio que sea necesario para los hijos que realmente tenga en cada momento.

Las representaciones que ahorran espacio suelen hacerlo a costa de complicar los algoritmos de manejo de la estructura. Sin embargo, esta última representación de un árbol general, tiene la ventaja de que nos ofrece una visión de cualquier árbol general como si fuese un árbol binario (cada nodo contiene un campo de datos y dos referencias a otro nodo) lo cual puede ser interesante, ya que nos permite reutilizar en parte los algoritmos de manejo de árboles binarios para cualquier tipo de árbol. Eso tiene la ventaja de que los árboles binarios son quizás los más claros y fáciles de entender. En los siguientes apartados se estudiarán los árboles binarios.

6.1.1 Representación sin referencias y punteros

También es posible representar estructuras arborescentes sin el uso de referencias o punteros a nodos, simulándolos mediante los índices de un array:

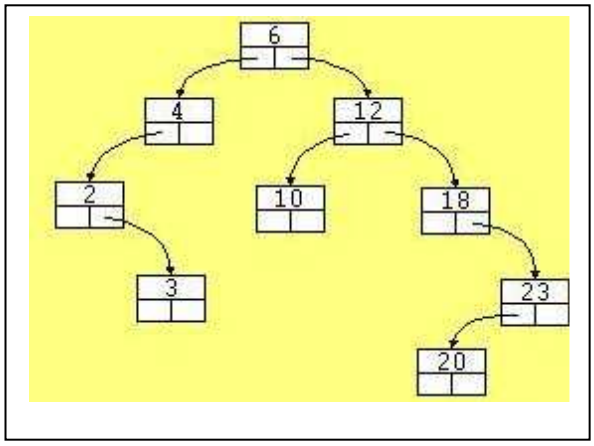
Primera forma: Mediante un solo array.

- Para un árbol n-ario (cada nodo tiene un máximo de n hijos) se sitúa el nodo raíz en la posición 0 del array
- Para el nodo en la posición i del array, sus hijos estarán en las posiciones i*n+1, i*n+2, ... i*n+n
- Para el nodo en la posición i del array, su padre está en la posición indicada por la parte entera de (i-1)/n

De esa forma se puede acceder directamente tanto al padre como a cualquiera de los hijos de cualquier nodo. Sin embargo, es una estructura estática, que nos obliga a reservar espacio de almacenamiento para todos los nodos.

Segunda forma: Mediante tres arrays paralelos, a los que llamaremos INFO, HIJO y HERMANO

- INFO[k] contiene la información útil del nodo
- HIJO[k] contiene la posición del primer hijo del nodo N, o un valor nulo para indicar que N no tiene hijos.
- HERMANO[k] contiene la posición en el vector del siguiente hermano, o un valor nulo para indicar que N es el último hijo de su padre.



Tercera forma: Mediante un array bidimensional de orden m por (n+1)

- m es el número de nodos del árbol
- n es el número máximo de hijos para cada nodo. Se le suma uno porque cada columna representa un campo del nodo, y éste debe tener un campo para la dirección de cada hijo, más un campo para los datos útiles del nodo.

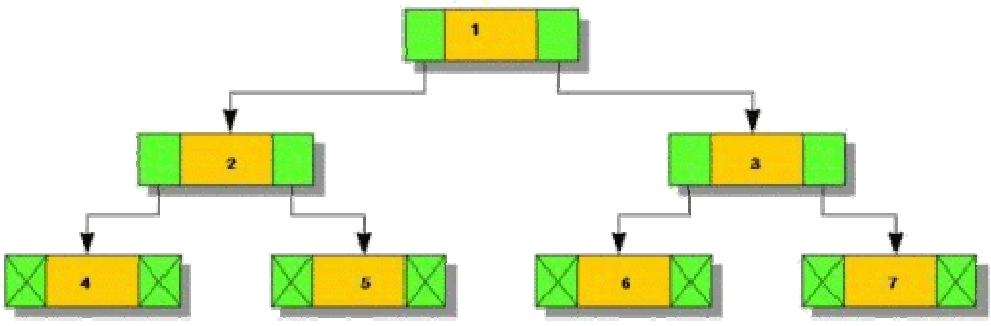
6.2 Tipos de árboles más usuales

Situaciones en las que los árboles sean de utilidad ya hemos comentado que hay muchas, y muy variadas, y cada una de ellas tiene un tipo de árbol específico que se adapta mejor que los demás a esa situación.

Vamos a ver unas nociones sobre cuáles son los tipos principales de árboles, qué utilidades pueden tener, y para qué se usan por lo general.

6.2.1 Árboles binarios

Quizás los árboles binarios sean de los más usados debido a su gran cantidad de aplicaciones y a la sencillez relativa de su estructura. Son útiles para comprender bien los algoritmos de manejo de árboles.



Básicamente un **árbol binario se define de la misma manera que hemos definido un árbol general n-ario, pero tiene la particularidad de que cada nodo puede tener como máximo dos nodos hijos, (n=2) que además están ordenados, y que solemos llamar hijo izquierdo e hijo derecho (o nodo izquierdo y nodo derecho).**

Como en el caso de los árboles generales, cada uno de los hijos, izquierdo y derecho, pueden considerarse como los nodos raíz de un subárbol binario izquierdo y de un subárbol binario derecho, que eventualmente pueden estar vacíos.

Como vimos, tenemos **tres formas de representar un árbol binario**

Implementar un árbol binario mediante un **único array**:

- El nodo raíz se sitúa en la posición 1 del array.
- Para el nodo de la posición i:
    - Su padre estará en la posición parteEntera(i/2) (salvo si i=1, ya que el nodo raíz no tiene padre)
    - Su hijo izquierdo estará en la posición 2i.
    - Su hijo derecho estará en la posición 2i+1.

Cada representación tiene sus ventajas y sus inconvenientes:

- Tiene la **ventaja** de que de esta forma, si los nodos se numeran por niveles, el nodo i estará en la posición i del array, y podemos acceder directamente al nodo padre y a los nodos hijos de uno dado.
- Tiene el **inconveniente** de que al usar un array, el árbol deja de ser una estructura dinámica, y se convierte en una estructura estática, que como hemos visto impone limitaciones en cuanto al número de elementos que puede contener y no hace un uso eficiente de la memoria.

Implementar un árbol binario mediante **tres arrays paralelos, INFO, IZQUIERDO, DERECHO:**

- INFO[k] Contiene el campo de información útil del nodo k (que está en la posición k).
- IZQUIERDO[k] Contiene la posición del hijo izquierdo del nodo de la posición k.
- DERECHO[k] Contiene la posición del hijo derecho del nodo de la posición k.

Esta representación tiene una gran ventaja y un gran inconveniente:

- **Ventaja**: Cualquier árbol general que esté representado con tres arrays paralelos INFO, HIJO, HERMANO, como vimos en el apartado 6.1, puede representarse en memoria como un árbol binario sin más que considerar que el array HIJO del árbol general representa al array IZQUIERDO del árbol binario, y el array HERMANO del árbol general representa al array DERECHO del árbol binario. Eso permite que los algoritmos de inserción, borrado, búsqueda y recorrido de árboles binarios puedan usarse para cualquier árbol general, con independencia del número de hijos que pueda tener cada nodo.
- **Inconveniente**: Nuevamente, al usar arrays, el árbol deja de ser una estructura dinámica, y se convierte en una estructura estática.

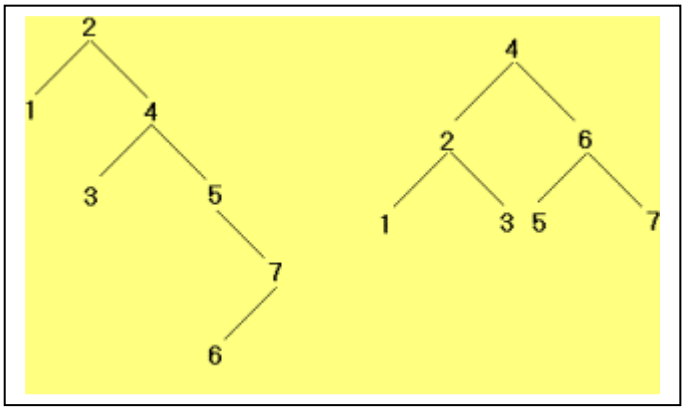
Implementar el árbol binario mediante **nodos enlazados por referencias**:

- Hay una referencia que apunta al nodo raíz.
- Cada nodo contiene:
    - Una zona de datos, que contiene la información útil del nodo
    - Una referencia al nodo raíz del subárbol izquierdo (al hijo izquierdo del nodo)
    - Una referencia al nodo raíz del subárbol derecho (al hijo derecho del nodo)

Ventajas e inconvenientes de esta representación:

- **Ventaja**: El árbol queda representado auténticamente como una estructura dinámica de datos. No hay limitación en el número de elementos que puede contener, y se hace un uso eficiente de la memoria.
- **Inconveniente**: No podemos acceder directamente a cada nodo del árbol. Hay que comenzar a buscarlo desde el nodo raíz, con lo que los algoritmos de búsqueda, inserción y eliminación de un elemento concreto del árbol, pueden ser más lentos, sobre todo si se quiere que el árbol esté ordenado por algún criterio.

6.2.2 Árboles binarios de búsqueda



Aunque hasta ahora hemos visto las características generales de un árbol binario, existen muchas variantes, dependiendo del uso que queramos hacer de la información. Uno de los usos de los árboles binarios es mejorar la eficiencia de las búsquedas. Ése es el fin que pretenden los árboles binarios de búsqueda, de ahí su nombre.

Un árbol binario de búsqueda es un árbol binario en el que **cada nodo cumple que su campo de información es**

mayor que el de cualquier nodo de su subárbol izquierdo y menor que el de cualquier nodo de su subárbol derecho.

Si se quieren permitir valores duplicados, la condición sería mayor que cualquier valor del subárbol izquierdo y **menor o igual** que cualquier valor de su subárbol derecho.

Esto nos permite mejorar las **búsquedas**, ya que en el nodo raíz podemos desechar la búsqueda en la mitad del árbol, y en cada nuevo nodo podemos volver a desechar la búsqueda en otra mitad del subárbol, y así sucesivamente.

¿Cuántas comparaciones tendremos que hacer antes de encontrar el elemento buscado o de saber que no está en el árbol?

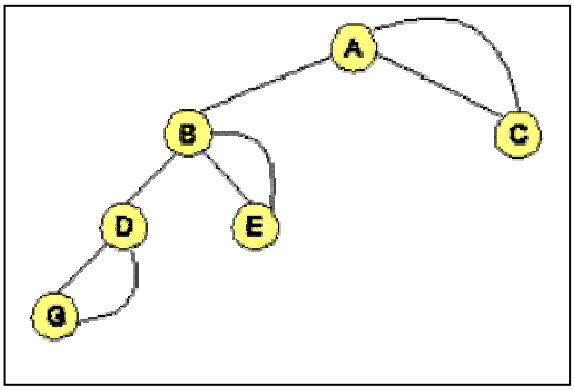
Hacemos una comprobación en cada nodo por el que vamos pasando, en cada nivel del árbol, por lo que en el peor de los casos, en el que el nodo buscado es una hoja de la rama más larga del árbol, haremos tantas comparaciones como niveles tenga el árbol. Teniendo en cuenta que en un árbol binario de **n** niveles caben $\mathbf{2^n-1}$ nodos, (aproximadamente 2 elevado a n) podemos considerarlo desde el punto de vista contrario: En un árbol de n nodos, como máximo tendremos que hacer $\mathbf{\log_2 n}$ comparaciones para encontrar el valor buscado. Se dice que la **eficiencia del algoritmo es del orden de** $\mathbf{\log_2 n}$. Esto se suele expresar como que su **eficiencia** es $\mathbf{O(\log_2 n)}$.

Si se contrasta con la eficiencia de otras estructuras, vemos lo siguiente:

- **Árboles binarios.** Las inserciones y eliminaciones se hacen cómodamente, y además la eficiencia de la búsqueda es $\mathbf{O(\log_2 n)}$, lo que quiere decir que el número de operaciones necesarias para encontrar un elemento crece mucho menos que lo que crece la estructura. Es decir, la eficiencia del algoritmo sigue siendo buena, aunque el tamaño del árbol crezca exponencialmente. Por ejemplo, con un árbol de 128 elementos tendríamos que hacer del orden de 7 comparaciones en una búsqueda ($\mathbf{\log_2 128=7}$). Pero para el doble de elementos, 256, sólo tendremos que hacer una comparación más, ya que $\mathbf{\log_2 256=8}$. Así, para 16384 elementos habría que hacer del orden de 14 comparaciones. Vemos que mientras que el número de comparaciones necesarias se ha multiplicado por 2, el número de elementos se ha multiplicado por 128.
- **Arrays lineales ordenados:** El tiempo de búsqueda también puede ser de eficiencia $\mathbf{O(\log_2 n)}$, si se usa una búsqueda binaria, pero es costosa la inserción y eliminación de elementos del array de forma que se mantenga ordenado. Mientras más elementos tenga el array, más elementos deberán desplazarse al insertar o eliminar para que el array permanezca ordenado y empaquetado (sin huecos). Por eso la eficiencia de los algoritmos de inserción y eliminación es de **O(n)**. (El número de operaciones necesarias para insertar un elemento crece proporcionalmente al aumento del número de elementos que contenga el array: Doble de elementos requieren doble de operaciones).
- **Listas enlazadas:** Se pueden insertar y eliminar cómodamente elementos en la lista, pero la búsqueda es costosa y lenta, ya que puede ser necesario recorrer toda la lista para encontrar el elemento buscado. Su eficiencia es **O(n)**

6.2.3 Árboles binarios enhebrados

Hay un detalle que puede haber pasado inadvertido hasta ahora, pero que bien entendido puede ser importante.



¿Qué pasa con las referencias al nodo izquierdo y nodo derecho de los nodos hoja, que no tienen hijos? ¿Y con la referencia de aquellos nodos que sólo tienen un hijo?

Con lo que hemos venido diciendo hasta ahora, lo que ocurre es que apuntan a `null`.

En cierta forma, estamos reservando un espacio para todas esas referencias, que no es despreciable, para a fin de cuentas no guardar nada en ellas, no apuntar a ningún sitio, que es lo que significa `null`.

¿Es ese espacio realmente no despreciable? ¿Son realmente muchas las referencias que apuntan a `null` en un árbol binario?

Si pensamos en un **árbol binario de profundidad N**, completo, en el que cada nodo, salvo los nodos hoja, tiene dos hijos. Es decir, sólo las hojas contienen referencias `null`, que es la situación en la que menos referencias `null` puedes tener, ya que las hojas, por definición, forzosamente van a tener a `null` sus dos referencias.

En ese caso, árbol de **profundidad o altura A** y, por tanto de **nivel `(A-1)`**, veamos algunos datos: (no contamos la referencia al nodo raíz.)

- Contiene en total 2^A -1 nodos. Como cada nodo tiene dos referencias, en total $2^{(A+1)}$-2 referencias. Ejemplo: con 10 niveles, tendrá 1023 nodos, y 2046 referencias
- Cada nivel contiene $2^{(A-1)}$ nodos (donde A es la altura de ese nivel). El último nivel, el de las hojas, contiene $2^{(A-1)}$ hojas. Ejemplo: con 10 niveles el nivel 10 tiene 512 nodos hoja, es decir, la mitad de los nodos del árbol son hojas.
- En el último nivel cada hoja sigue teniendo dos referencias que apuntan a `null`. Sólo en el último nivel hay siempre 2^A referencias apuntando a `null`, del total, lo que supone algo más de la mitad de todas las referencias del árbol apuntando a `null`. Ejemplo: con 10 niveles, de las 2046 referencias del árbol, 1024 apuntan a `null` (una más de la mitad).

Como resumen, aproximadamente la mitad de las referencias de un árbol binario completo de búsqueda apuntan a `null`, inevitablemente.

La idea básica de los árboles enhebrados es aprovechar las referencias a `null` de los nodos para que apunten convenientemente a uno de los nodos antecesores, de forma que sea posible mejorar la eficiencia al recorrer el árbol de una determinada manera. A estas referencias "reutilizadas" se les llama **hebras**, y a los árboles binarios que las contienen **árboles enhebrados**.

Debe ser posible distinguir las hebras de las referencias normales, añadiendo al nodo algún campo adicional que indique si la referencia es una hebra o una referencia normal, o indicando las referencias como enteros positivos y las hebras como enteros negativos, por ejemplo.

Existen distintas formas de enhebrar un árbol, asociadas a una determinada forma de recorrer los nodos del árbol. Para un recorrido "**inorden**" hay que tener en cuenta:

- Un valor `null` en un enlace derecho de un nodo p se reemplaza por una hebra al nodo que se visitaría después de p en un recorrido inorden (sucesor inorden de p).
- Un valor `null` en un enlace izquierdo de un nodo p se reemplaza por una hebra al nodo que se visitaría antes de p en un recorrido inorden (predecesor inorden de p).
- Crearemos un nodo adicional de cabecera. Será el padre del primer nodo y el sucesor del último nodo.

Para un recorrido **preorden**, existe un enhebrado similar, pero no hay un enhebrado para un recorrido **postorden**.

En definitiva, **al enhebrar un árbol conseguimos**:

- **Mejorar la eficiencia de los algoritmos de recorrido del árbol**, al poder acceder directamente al nodo siguiente a visitar y al anterior en el orden del recorrido.

Eso lo conseguimos **a costa de:**

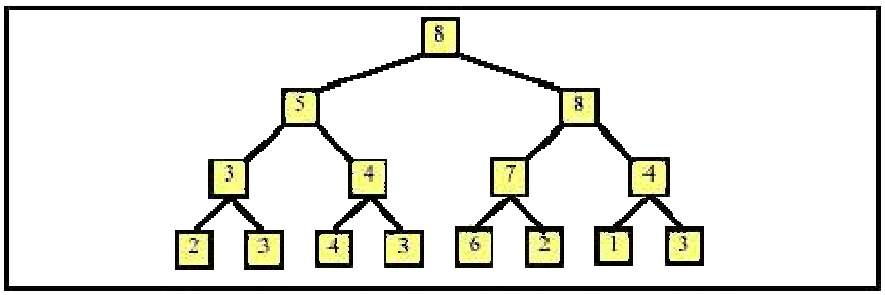
- **Complicar los algoritmos de inserción y borrado**, que tienen que preocuparse ahora de que el árbol quede correctamente enhebrado tras cada eliminación o borrado.

Si las operaciones más frecuentes que se van a hacer en el árbol son inserciones y borrados, y el recorrido completo del árbol es poco frecuente, no merece la pena.

Si por el contrario, una vez creado el árbol rara vez se insertarán o borrarán elementos, y éste se usa básicamente para recorrerlo procesando sus elementos, sí que merece la pena.

6.2.4 Árboles binarios en montón (heap)

Una de las muchas utilidades que hemos mencionado para los árboles era la ordenación de una serie de elementos.



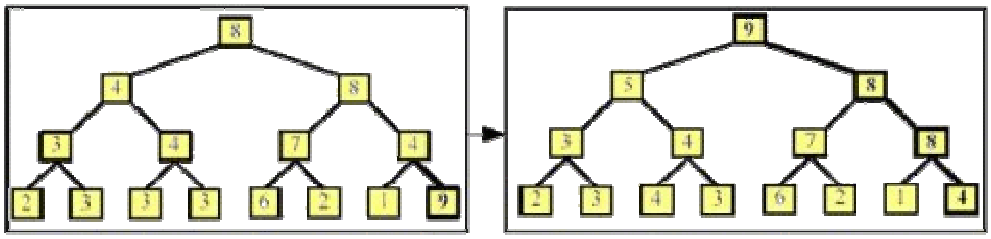
Hay un algoritmo de ordenación llamado **ordenación por montón**, que consiste en ir leyendo todos los elementos que deben ser ordenados e ir insertándolos en un árbol en montón. Para obtener la lista ordenada, basta con ir extrayendo del montón el nodo raíz, hasta que el árbol esté vacío. Supongamos que los datos están en un array que queremos ordenar. Incluso es posible implementar el árbol en montón en el propio array. Este algoritmo presenta una eficiencia de **$O(n\log_2 n)$**, tanto para el peor de los casos como para el caso medio. Es mejor que la ordenación por burbuja, con una eficiencia de **$O(n^2)$** e incluso que la ordenación rápida, que tiene una eficiencia para el caso medio también de **$O(n\log_2 n)$**, pero que tiene una eficiencia de **$O(n^2)$** para el caso menos favorable.

El algoritmo se basa en las propiedades especiales de los árboles en montón (**heap**). Pero un árbol en montón también es una excelente implementación para una cola de prioridades.

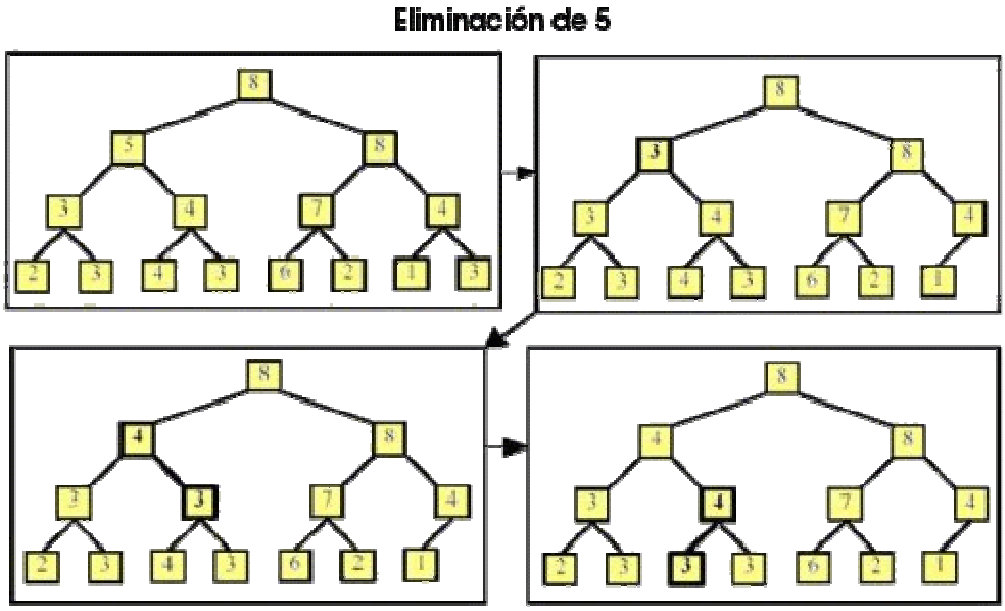
Veamos en qué consiste un árbol en montón:

- Un árbol en montón es un **árbol binario completo**, es decir, todos sus niveles tienen el número máximo de nodos posibles, excepto el último, en el que se cumple que los nodos están situados lo más a la izquierda posible.
- Cada nodo cumple que **el dato que almacena es mayor o igual que el dato de cualquier nodo perteneciente a cualquiera de los subárboles**. Es decir, la raíz siempre tiene el elemento mayor del árbol, por lo que al ir extrayendo la raíz del árbol, siempre vamos sacando el elemento mayor de los que quedan, en orden de mayor a menor.
- Para insertar un elemento, siempre lo **insertamos como la última hoja del árbol**, y comparamos repetidamente con el padre, de forma que si no cumplen la condición de montón, (el nuevo nodo no es menor que su padre) ambos se intercambian, y se sigue comparando con el nuevo padre, hasta que se cumpla la condición de montón. (en el peor de los casos, al comparar con el raíz).

Inserción de 9



- Para borrar un elemento, **se borra siempre el nodo raíz,** y se sustituye por la última hoja. Se compara repetidamente con los hijos, de forma que si es menor que el mayor de los hijos, se intercambia con este, y se vuelve a comparar con los nuevos hijos, hasta que se cumpla la condición de montón, que en el peor de los casos será cuando llegue a comparar con una hoja.



- Se puede definir análogamente un montón para establecer un orden de menor a mayor, cambiando la condición mayor o igual por menor o igual

Esa buena eficiencia que se consigue usando esta estructura en el algoritmo de ordenación, se consigue a costa de complicar los algoritmos de inserción y eliminación, que ahora deben hacer operaciones adicionales para asegurarse de que el árbol sigue siendo en todo momento un montón.

6.2.5 Árboles B y B+

Una base de datos puede contener miles, incluso cientos de miles de registros. Buscar un registro determinado en medio de tantos no es fácil, y se trata una operación muy usual.

A eso añadimos que hay que buscarlos en el disco, porque todos no caben a un tiempo en la memoria del ordenador, y que cualquier operación con una unidad de E/S es lenta, por eso hay que hacer las cosas lo mejor posible.

Debe conseguirse un algoritmo de búsqueda que sea altamente eficiente, y que sea capaz de encontrar cualquier dato en un tiempo razonable, minimizando el número de accesos necesarios al disco y garantizando que el tiempo de respuesta va a ser siempre el mismo, que no va a depender del dato concreto que buscamos.

Para eso se inventaron los **árboles B y B^+. Su principal aplicación es la gestión de las claves de los ficheros de índices de una base de datos.**

Las propiedades que los hacen muy adecuados para este fin es que el algoritmo de búsqueda de un valor dentro del árbol es extremadamente rápido. A eso se une la propiedad de que todos los nodos hojas, que son los que realmente contienen los datos, están en el mismo nivel del árbol, por lo que el tiempo de búsqueda es siempre el mismo, sea cual sea el elemento que estemos buscando y la posición que éste ocupe en el árbol. Además, las hojas del árbol pueden apuntar a las páginas en las que se divide el fichero índice para ser almacenado en disco, por lo que son muy adecuados para manejar índices muy grandes, que no caben en memoria, reduciendo al máximo el número de accesos a disco.

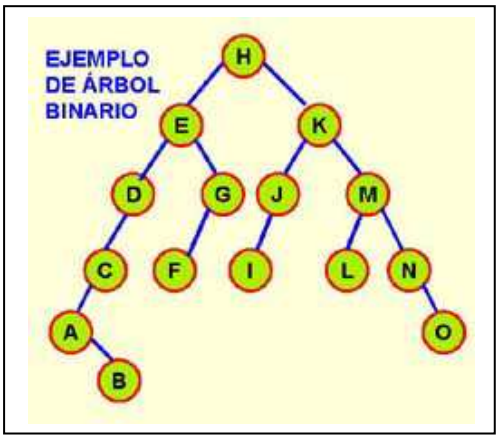
Las **características básicas** de un árbol B de orden n son:

- Cada nodo tiene un máximo de **2n** claves y un mínimo de **n**. (Cada clave es un dato a guardar en el árbol).
- Cada nodo que no sea una hoja, tiene **m+1** hijos, siendo m el número de claves que contiene en ese momento el nodo.
- Todas las hojas están en el mismo nivel, lo que hace que el tiempo de búsqueda para un valor sea constante, con independencia del valor buscado.

Un árbol B^+ consiste en combinar un fichero de datos estructurado en una secuencia de bloques de registros con un árbol B para indexar dichos bloques.

6.3 Recorrido de árboles binarios: Recorrido preorden, inorden y postorden

Ya hemos visto que se pueden almacenar los datos en árboles, y hemos visto que los árboles binarios pueden considerarse casi como un tipo general de árboles, ya que cualquier árbol puede convertirse en un árbol binario.



Pero almacenar los datos por sí mismos no suele resultar muy útil. Será necesario usarlos para algo. Y para usarlos es imprescindible poder recorrer el árbol para encontrarlos.

Con las listas enlazadas, recorrer todos los elementos era fácil. Bastaba con empezar por el principio, y continuar hasta el final pasando de siguiente en siguiente.

Pero **los árboles no son lineales**. Hay que asegurarse de haber recorrido todos los nodos de un árbol sin dejarse ninguno. Y lo que es igual de importante, hay que hacerlo sin tener que visitar más de una vez cada nodo.

Existen **tres formas básicas** de recorrer un árbol binario de forma que se garantice que todos los nodos serán visitados una y sólo una vez. Hay problemas en los que no es indiferente la forma de recorrer el árbol, y es necesario recorrerlo de una forma determinada. En las tres se visita antes el nodo izquierdo y luego el derecho. Lo que cambia es el orden en el que se visita el nodo raíz. Esa forma es la que le da nombre al recorrido:

- **Recorrido preorden: El nodo raíz se visita antes que los hijos.**

Primero se visita, por ejemplo para escribir su valor, el nodo raíz, luego se recorre en preorden el subárbol izquierdo y finalmente se recorre en preorden el subárbol derecho. La condición de parada es que el puntero que apunta al raíz valga null. Si no hay árbol que recorrer, pues no se recorre.

Usado por ejemplo para representar expresiones en notación polaca. La notación polaca consiste en escribir el operador delante de los dos operandos a los que acompaña. De esta manera no se hace necesario el uso de paréntesis ni de reglas de precedencia para saber cual es el operador que se debe ejecutar primero. El nodo contiene el operador, y los hijos izquierdo y derecho, los dos operandos, que a su vez pueden ser expresiones en notación polaca construidas a partir de un operador, o ser directamente literales.

- **Recorrido inorden: El nodo raíz se visita entre los dos nodos hijos.**

Primero se recorre el subárbol izquierdo en inorden, luego se visita el nodo raíz, y luego se recorre el subárbol derecho en inorden.

Se usa para clasificar los elementos del árbol según un determinado orden.

- **Recorrido postorden: El nodo raíz se visita después de los dos nodos hijos**.

Primero se recorre el subárbol izquierdo en postorden, luego se recorre el subárbol derecho en postorden, y finalmente se visita el nodo raíz.

Se usa para representar expresiones en los compiladores, a la hora de analizar las sentencias. También para representar expresiones matemáticas en notación polaca inversa. La diferencia con la notación polaca es que en este caso, primero se escriben los operandos, y detrás el operador. Es totalmente equivalente, y también tiene la ventaja de evitar el uso de paréntesis y reglas de precedencia de operadores.

Realmente, las implementaciones de los tres tipos de recorridos son "simétricas", cambiando sólo el orden en que se realizan las tres operaciones.

Es evidente que estos tres recorridos de un árbol son inherentemente recursivos, por lo que la implementación más habitual de estos algoritmos es usando recursividad.

6.3.1 Implementación de un árbol binario de búsqueda en Java

Como en casi cualquier estructura de datos, las operaciones más importantes y que más nos interesa conocer sobre manipulación de árboles binarios son las relacionadas con añadir y borrar elementos del árbol, recorrer todos los elementos para procesarlos, buscar un valor determinado, saber si está vacío o no, el número de elementos que contiene o cual es el elemento por el que comienza la estructura (la raíz del árbol).

Las operaciones que debería tener disponibles un árbol binario son:

Operaciones básicas con un árbol binario de búsqueda	
Operación	**Descripción**
Añadir	Insertar un nuevo dato en el árbol.
Eliminar	Borrar un dato del árbol.
Árbol Vacío	Comprobar si el árbol está vacío.
Recorrido preorden (raíz, izquierdo, derecho)	Recorrer todos los nodos del árbol, visitándolos una sola vez: primero visita el nodo raíz, luego recorre el subárbol izquierdo (hijo izquierdo) en preorden y finalmente recorre el subárbol derecho (hijo derecho) también en preorden. El recorrido puede efectuarse para procesar todos los nodos o simplemente para hacer un listado de todos ellos.
Recorrido inorden (izquierdo, raíz, derecho)	Recorrer todos los nodos del árbol, visitándolos una sola vez : primero recorre el subárbol izquierdo (hijo izquierdo) con un recorrido inorden, luego visita el nodo raíz y finalmente recorre inorden el subárbol derecho (hijo derecho)
Recorrido postorden (izquierdo, derecho, raíz)	Recorrer todos los nodos del árbol, visitándolos una sola vez: primero recorre el subárbol izquierdo (hijo izquierdo) en postorden, luego recorre el subárbol derecho (hijo derecho) también en postorden, y finalmente se visita el nodo raíz. .
Búsqueda	Buscar y consultar la información del elemento deseado del árbol.
Raíz	Consultar la información del nodo raíz.
Elementos	Consultar el número de elementos que tiene el árbol.