1. Concepto de estructura de datos dinámica

Concepto de estructura de datos dinámica

Cuando como programador terminas completamente tu primer programa, eres la persona que mejor lo conoce y quien puede sacarle el máximo rendimiento. Conoces todas sus posibilidades y limitaciones. Ésta es la sensación que tiene ahora ***Víctor****. Acaba de concluir un programa de uso interno en la empresa para la que trabaja,* ***SI Andalucía****. El programa consiste en la gestión de proyectos, mediante el que se recogen todos los datos de cada proyecto desde su diseño hasta que se entrega al cliente, especificando cada día las aportaciones realizadas por cada uno de los participantes en el proyecto. Con este programa al finalizar, pueden conocer exactamente el trabajo realizado por cada uno de los miembros del proyecto.*
Víctor *ha quedado muy satisfecho de su programa, se siente eufórico, capaz de programar cualquier aplicación y así se lo hace saber a* ***José****. Éste le explica que ha hecho un buen trabajo, pero que no debe confiarse. Esto no ha hecho más que empezar. Le explica que cada vez que se enfrente a una nueva aplicación, debe partir de cero y realizar un profundo análisis de la situación en que se va a utilizar. Además le cuenta que hay algunos temas que aún no conoce como son las estructuras dinámicas de datos y que se utilizan con mayor frecuencia de lo que parece.* ***Víctor*** *recuerda que ya hablaron de eso al principio, cuando comenzaron a formarle en programación, y que efectivamente no tienen ni idea de cómo o cuándo usarlas.*

El concepto de **estructura de datos dinámica** se entiende mejor como contraposición al concepto de estructura de datos estática del que hemos venido hablando en las dos unidades anteriores.

Todas las estructuras de datos estáticas tenían algo en común. ¿Recuerdas lo que era?

Efectivamente. La forma de almacenar los datos en la memoria.

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

Por **ejemplo**, cuando creábamos un array de números enteros, teníamos que dimensionarlo, decir cuál era el número máximo de enteros que podía contener. Si lo dimensionábamos para 20 números enteros, y sólo guardábamos uno, las otras 19 posiciones del array estarían reservadas para contener un número entero. De hecho, estarían inicializadas por defecto al valor cero. Con independencia de que introdujéramos más o menos números enteros, siempre estaríamos utilizando toda la memoria reservada para los 20 números, con lo que se estaría desperdiciando memoria.

Pero si por alguna causa en nuestro programa necesitáramos guardar más de 20 números enteros, tendríamos 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.

Todo un lío, que requiere tiempo de procesamiento, o una limitación muy estricta en el número de elementos que puede contener el array, además de usar en la mayoría de los casos más memoria de la estrictamente necesaria.

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

¿Es posible usar estructuras de datos que no impongan esas limitaciones ni tengan esos inconvenientes?

Justamente para eso sirven las **estructuras dinámicas de datos**.

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

Si se inserta un nuevo elemento o dato 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.

Podemos concluir generalizando que, las estructuras de datos dinámicas sí hacen un uso eficiente de la memoria, aunque en la mayoría de los casos es a costa de complicar los algoritmos de manipulación de estas estructuras (insertar, modificar, eliminar, buscar, procesar o listar elementos)

Por **ejemplo**, y aunque sea anticiparnos un poco a los apartados siguientes,

- si tenemos una lista enlazada de números enteros,
- sólo necesitaremos el espacio necesario para los números que contenga en cada momento,
- más un espacio adicional para almacenar una referencia al siguiente elemento de la lista por cada número que contenga,
- más una referencia que nos diga cual es el primer elemento de la lista.

El tamaño adicional requerido para almacenar esas referencias puede considerarse bien empleado, ya que a cambio la estructura de datos puede crecer y disminuir de tamaño según las necesidades. Si en un momento dado la lista está vacía, sólo ocupará el espacio de la referencia al primer elemento de la lista, que contendrá el valor null. Si por el contrario necesito almacenar muchísimos números en la lista, la única limitación para que crezca el tamaño de la estructura todo lo que necesitemos será la propia capacidad de memoria del ordenador en que se ejecute el programa. Y en cada caso, la estructura ocupará exactamente sólo el tamaño necesario para los elementos que contiene.

El objetivo de los siguientes apartados será comprender cómo se consigue construir estructuras de datos dinámicas, particularizándolo para el caso de Java, y ver las estructuras de datos dinámicas más usuales: listas enlazadas, pilas, colas y árboles, así como las operaciones que suelen realizarse sobre ellas.

2. Punteros. (Referencias en Java)

Punteros. (Referencias en Java)

__José__ le dice a __Víctor__ que lo primero que debe tener claro es la utilización de las referencias en Java. Se trata de un concepto básico unido al recolector de basura que ya conoce. Estos dos conceptos son la base de toda la programación con estructuras de datos dinámicas en Java, ya que cualquier estructura dinámica utilizará referencias para gestionar sus elementos y cuando los abandone, será el recolector de basura el que automáticamente libere la memoria que habían usado.
__José__ dice que habitualmente las estructuras de datos dinámicas son recursos que el programador utiliza en ocasiones concretas para agilizar partes de sus programas.

En la mayoría de los lenguajes las estructuras de datos dinámicas se pueden construir gracias al uso de punteros. ¿Recuerdas de la unidad 2 lo que es un puntero?

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, es decir, que almacene un valor que indique que no apunta a ninguna posición de memoria, a ningún objeto. De esta manera, el tamaño de la estructura decrece, 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 mencionado que **en Java no existen los punteros, sino las referencias**.

Básicamente es una de las grandes ventajas de Java frente a otros lenguajes que sí usan punteros, ya que se simplifican mucho los programas y se hacen más seguros. Las referencias de Java ya sabemos lo que son, ya que venimos usándolas desde hace bastantes unidades. **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 detecta esa circunstancia 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, no sólo hay que "desenganchar" el objeto haciendo que ningún puntero lo apunte. Si lo dejamos sin ningún puntero pero no lo hemos eliminado, haría que ese objeto se quedara ocupando permanentemente la memoria, sin posibilidad de usarlo, pero también sin posibilidad de liberar la memoria que ocupa. Además de desengancharlo de la estructura, debemos conservar un puntero que nos diga donde 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.**

Autoevaluación

3. Listas enlazadas

Listas enlazadas

*Definitivamente **Víctor** se ha convencido de que aún le quedan muchas cosas por conocer, pero es un chaval con inquietudes y esto, lejos de aburrirle, le anima a seguir aprendiendo sobre programación. Ya ha probado la sensación de concluir un programa y ha decidido que esto es lo que le gusta hacer y a lo que quiere dedicarse.*
*Ha quedado con su amigo **José** en que le va a pasar unos apuntes y ejercicios que tiene sobre este tema de cuando estudiaba el ciclo formativo de Informática, para que conozca mejor las estructuras dinámicas de datos y pueda practicarlas. Dice que una vez que asimile esto, le mostrará algunas aplicaciones que utilizan listas enlazadas y pilas. Al ojear estos apuntes Víctor se da cuenta de que existen varios tipos de listas dependiendo del modo en que se enlacen los elementos. Aunque se siente un poco perdido con estos nuevos conceptos, en el fondo le gusta el tema.*

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).
- Listas de productos nocivos para la salud.
- Lista de títulos de los libros disponibles en una biblioteca.
- Lista de amigos o compañeros de trabajo.
- Listas de correo.

- Etc.

Seguro que tú mismo puedes pensar en muchas más situaciones de la vida real en las que disponer de una lista puede resultar de ayuda. Y los programas se construyen para facilitarnos la vida, para resolver más fácilmente y más eficientemente los problemas de la vida real. Por ello, cualquier lenguaje de programación debe permitir la creación y gestión de listas de muy diversos tipos.

Java no es la excepción, desde luego, y **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**.

Pero, ¿qué es una lista enlazada? Básicamente lo que cabe esperar:

Una lista es una colección de elementos colocados secuencialmente uno detrás de otro.

¿Nada más? Un array también es una colección de elementos colocados secuencialmente uno detrás de otro. ¿Cuál es la diferencia, entonces?

Cierto, hay diferencias importantes con un array:

- **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 saber 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.**

Bueno, pero en cierta manera, 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 hay necesidad de que lo sepa el propio elemento si lo sabe el programador. Y sabemos que el primer elemento está en la posición cero y que el último está en la posición length - 1. Aún no se aprecia una gran diferencia ni una mejora sustancial de las listas respecto a los arrays.

Cierto. Visto así parece que no hay una gran diferencia, 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. Ni más, ni menos.

Eso sí es una diferencia importante. Los arrays se dimensionaban al crearlos, y no cambiaban su número de elementos en toda la ejecución del programa. Aunque almacenáramos un solo elemento, o ninguno, ocupábamos el tamaño para todos los indicados en la dimensión del array. Y si necesitábamos más, ese array ya no nos servía, había que construir otro nuevo más grande y hacer que la referencia dejara de apuntar al viejo array para apuntar al nuevo de mayor tamaño.

3.1. Construcción en Java

Construcción en Java

Pero la pregunta ahora es, ¿Cómo consiguen las listas enlazadas cambiar de tamaño y ajustar la memoria usada estrictamente al tamaño necesitado? Ya lo hemos contestado en parte en el apartado anterior. Construyendo la estructura por medio de referencias. Veamos como lo haremos 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 que guardaran 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 como veremos más adelante, 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, que significa que serán accesibles desde todas las clases del mismo paquete. Con ello conseguimos que puedan ser modificados por 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 tendrán que ocupar de darle valor adecuadamente. 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. No te confundas, lo que estamos indicando realmente es que cada nodo va a llevar dentro una referencia que contendrá una 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. Eso es un asunto que abordaremos en los siguientes apartados.

Autoevaluación

3.2. Listas enlazadas de un tipo concreto de elemento

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.

Imagina que queremos formar una lista enlazada de objetos **`Persona`**. Tenemos básicamente 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 **`ListaEnlazadaNodos`** 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 "**`ListaEnlazadaNodos`**", 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, como verás, 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 de otro (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.
  Incluso podríamos 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`**. Pero para que esto tuviera utilidad, es necesario usar una serie de características de Programación Orientada a Objetos, que es algo que abordaremos en las dos próximas unidades.
- **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 el interface `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 quiere 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)

A continuación te mostramos el **código necesario para crear una lista enlazada de personas** (los nodos contendrán como dato objetos **`Persona`**) **como una estructura de datos abstracta**, que es la forma más general y más recomendable de plantear la construcción de listas enlazadas.

En este caso, la lista no está ordenada, por lo que todas las inserciones se realizan en la cabecera de la lista, es decir, cualquier nueva persona se inserta como primera persona de la lista. Al no importar el orden podría insertarse en cualquier otra posición, incluyendo el final de la lista, pero eso nos obligaría a recorrer la lista entera antes de hacer la inserción, y eso es menos eficiente que insertarlo directamente como primer elemento de la lista. A fin de cuentas, si el orden no importa, la primera posición es tan buena como cualquier otra. Como siempre, hemos intentado incluir en los comentarios del código las explicaciones detalladas de lo que se está haciendo, y porqué. Por eso es recomendable que los leas detenidamente para comprender la implementación de las listas enlazadas y sus operaciones básicas.

Descarga el Proyecto ListaEnlazadaPersonas

También te ponemos a continuación el **código necesario para crear una lista enlazada de personas como una estructura de datos concreta**. Esta forma, como podrás imaginar por lo explicado anteriormente, es menos elegante, y menos recomendable en la mayoría de los casos, pero desde luego es posible y también funciona correctamente, aunque no es reutilizable. La lista tampoco está ordenada, y los elementos se insertan siempre en la cabecera de la lista.

Descarga el Proyecto ListaEnlazadaPersonasConcreta

Cuando hayas leído los dos ejemplos detenidamente, habrás comprobado que si en un nuevo problema necesito una lista enlazada de otro tipo de objetos (Vehículos, por ejemplo) en el primer ejemplo, bastaría sustituir la clase Persona por la clase Vehículo, y hacer la clase de Gestión adecuada. En el segundo ejemplo, a pesar de que todos los métodos serían extremadamente parecidos, tendría que escribir de nuevo el código completo para el programa. Ésa es la gran diferencia entre un caso y otro, **la reusabilidad del software construido**.

Operación	Demo
Insertar un nuevo Elemento	**DEMO: Visualiza un ejemplo de inserción de elementos en una lista no ordenada**
Borrar un Elemento	**DEMO: Visualiza un ejemplo de borrado de elementos en una lista no ordenada**

3.3. Operaciones básicas en listas ordenadas y no ordenadas.

Operaciones básicas en listas ordenadas y no ordenadas

En los dos ejemplos de los apartados anteriores aparecen realmente todos los métodos que implementan las operaciones básicas con listas enlazadas. Eso sí, ambos ejemplos están hechos para el supuesto de que la lista no esté ordenada.

¿Cuáles son esas operaciones básicas con listas enlazadas?

- **ALTA: Inserción de nodos nuevos en la lista**, al principio, si la lista está desordenada, o en el lugar que corresponda, si debe estar ordenada.
- **BAJA: Eliminación de un nodo de la lista**.
- **REINICIALIZAR: Eliminación de la lista completa**. En Java es extremadamente 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 ocupan 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.

La siguiente tabla muestra los métodos en los que puedes encontrar implementadas cada una de las operaciones básicas para cada uno de los dos ejemplos del apartado anterior. En la mayoría de los casos, hay una clase de gestión (columna de la izquierda) que llama a un método genérico, que se encarga de preguntar algún dato necesario, para luego invocar a otro método de la clase que define la lista (columna de la derecha). Este segundo método es el que realmente implementa esa operación.

	LISTAENLAZADAPERSONASCONCRETA		LISTAENLAZADAPERSONA
Operaciones y métodos que las implementan	**Lista enlazada de un tipo definido. (no reutilizable)**		**Lista enlazada de un tipo ge**
	Clases que contienen los métodos		**Clases que contienen**
	GestionListaPersonas	**Persona**	**GestiónListaEnlazada**
ALTA	añadir()	Los 3 constructores: Persona()	añadir()
BAJA	eliminar()	eliminarPersona()	eliminar()
REINICIALIZAR	reiniciarLista()	actualizarPrimeraPersona() actualizarTotalPersonas()	reiniciarLista()
BÚSQUEDA	buscar()	buscarPersona()	buscar()
PROCESAMIENTO O RECORRIDO	actualizarEdadEnAñoNuevo()	-----	actualizarEdadEnAñoNuevo()
LISTADO COMPLETO	listado()	listarPersonas()	listado()
LISTADO PARCIAL	listadoMayoresEdad()	-----	listadoMayoresEdad()
CONSULTA Nº NODOS	-----	obtenerTotalPersonas()	-----
MODIFICAR	modificarPersonas()	-----	modificarPersonas()
GUARDAR	guardar()	-----	guardar()
CARGAR	cargar()	-----	cargar()

Es posible que pienses que realizar esas operaciones sobre una lista enlazada que sí deba permanecer ordenada por algún criterio, suponga grandes cambios sobre el código que hace cada una de esas operaciones.

Realmente hay cambios, pero yo no diría que sean "grandes cambios".

A continuación te ponemos el código del ejemplo anterior referido a la lista genérica, reutilizable, pero adaptado para gestionar una lista enlazada de personas que **debe permanecer ordenada alfabéticamente por nombre, en orden ascendente (de la A a la Z)**. Para el tipo de lista no genérica, las adaptaciones serían bastante similares, por lo que no merece la pena comentarlas, ya que se trata de una forma menos aconsejable de hacer las cosas.

Para ayudarte a identificar los cambios, hemos eliminado todos los comentarios que contenía el código de todas las clases del ejemplo, y sólo hemos insertado nuevos comentarios en aquellos lugares que introducen algún cambio debido a que la lista debe permanecer siempre ordenada. Es el código de esos métodos que aparecen comentados al que le debes prestar atención en el siguiente ejemplo.

Descarga el proyecto ListaEnlazadaGenericaOrdenada

3.4. Otros tipos de listas enlazadas

Otros tipos de listas enlazadas

Habrás notado que las listas enlazadas son estructuras de datos bastante versátiles. El hecho de que se puedan implementar como estructuras de datos "fijas o concretas" o bien como estructuras "abstractas" de datos así lo prueba.

De hecho, las listas son algo bastante "plástico", y teniendo claro los conceptos de cómo se definen y cómo se realizan las operaciones básicas, es bastante fácil hacerles modificaciones y adaptaciones que les permitan adaptarse mejor a multitud de problemas y situaciones.

En concreto, algunas de estas modificaciones más o menos típicas son de las que te vamos a hablar en este apartado.

¿Por qué poder movernos sólo hacia delante en la lista enlazada? ¿Podría 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 atrás? De esta manera la probabilidad de encontrarlo rápido aumentaría. Sobre todo si la lista enlazada contiene gran cantidad de nodos, ese ahorro de tiempo puede resultar interesante.

Podemos construir una **lista doblemente enlazada** para conseguir esa funcionalidad.

3.4.1. Listas doblemente enlazadas

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`.

Como podrás imaginarte 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 y de forma mucho más clara la realidad de nuestro problema, o si se consigue alguna mejora en la eficiencia.

Esta mejora en la eficiencia pude 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 Zacarías, seguro que mejora la eficiencia del algoritmo si empezamos la búsqueda por el final y vamos retrocediendo en la lista. Pero por ejemplo, para Manuel, ya no estará tan claro por qué extremo de la lista interesa comenzar la búsqueda. Dependerá de los nombres concretos que almacena la lista, y eso a priori no lo podemos conocer. 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 imagina 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.

En la siguiente figura te presentamos el formato aproximado que tendrían esas fichas en una ventana de la aplicación para mostrar los datos de cada persona en pantalla:

En el **código** siguiente tienes un ejemplo básico de cómo hacer una lista doblemente enlazada de Personas. No dispone de un interfaz gráfico tan interesante como la que se muestra en la imagen anterior, pero ya nos vamos acercando al punto en el que aprenderemos a hacerlo. Por ahora sólo realiza las operaciones básicas de insertar una nueva persona en la lista, borrar una persona de la lista, buscar una persona en la lista y mostrar el listado completo de la lista en orden alfabético ascendente o descendente. Para poder hacer ese listado en orden descendente es imprescindible que la lista esté doblemente enlazada. La lista está ordenada por el campo nombre.

Descarga el proyecto ListaDoblementeEnlazada

Autoevaluación

3.4.2. Listas enlazadas circulares

Listas enlazadas circulares

Pero las posibilidades de las listas enlazadas no acaban ahí. Como dijimos son algo bastante versátil, y podemos pensar en situaciones en las que los nodos de la lista se organicen de muy diversas formas con algún 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. Un **ejemplo** sería un sistema operativo multitarea, que asigna un cierto quantum de tiempo a cada una de las aplicaciones que se ejecutan de forma concurrente.Una vez atendida la última aplicación, hay que volver a atender a la primera. Si la lista es circular, esto no requiere ninguna operación especial. Empezar por el principio sigue siendo avanzar al siguiente elemento. Cada nueva aplicación en ejecución será un nuevo nodo insertado en la lista circular, y cada aplicación que finaliza será un nodo eliminado de la lista circular.

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

- **Una lista enlazada circular 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 coger 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 coger 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.

El enlace siguiente te ofrece todo el proyecto hecho en Java con NetBeans de ese programa.

Descarga el proyecto ListaEnlazadaCircular

3.4.3. Listas circulares doblemente enlazadas

Listas circulares doblemente enlazadas

Naturalmente podemos pensar en tener listas circulares doblemente enlazadas, en las que en una lista circular cada nodo tuviera un enlace al elemento anterior y un enlace al elemento siguiente de la lista. Te dejamos a tu imaginación la búsqueda de una situación real en la que pudiera resultar útil o ventajoso ese tipo de listas, aunque la realidad es que su uso es bastante infrecuente. Tampoco de ofrecemos una implementación de este tipo de listas, por que sería ahondar demasiado en el mismo tema.

Pero sí te indicamos las cuestiones a tener en cuenta para su implementación:

- **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 por que al menos hay un nodo en la lista.

Autoevaluación

4. Pilas.

Pilas

*Cuando **Víctor** llega al tema de las pilas, aprecia que los conceptos siempre son los mismos y que las diferencias están en la concepción de la estructura y por tanto en la forma en que serán programadas. Observa que efectivamente la pila es similar a la lista, pero que la primera es bastante más sencilla y las referencias a los elementos disminuyen, ya que para un elemento el siguiente y el anterior son el mismo.*

¿Recuerdas que cuando hablábamos de **recursividad** mencionamos el uso de una pila?

Decíamos que cada nueva invocación al **método recursivo** se almacenaba en la pila, para luego ir descargándola a medida que se iba completando y poder continuar con la invocación anterior, y así hasta que se agotaba la pila, lo que era señal de que se había completado la primera invocación recursiva al método, y por tanto el problema estaba resuelto. Usábamos la pila para guardar las distintas llamadas al método para luego recuperarlas en orden contrario a como se introdujeron. Es la máquina virtual Java y el Sistema Operativo los que se encargan de gestionar la pila de memoria en ese caso.

¿Pero a qué nos referimos en programación cuando hablamos de pila?

Es una estructura de datos secuencial, en la que los datos están almacenados en estricto orden de llegada de forma que el primero es el último introducido, en la que los elementos se introducen siempre en el mismo extremo de la lista, y se extraen siempre del mismo extremo de la lista. A ese extremo se le llama cabecera o cima de la pila. Es una estructura de datos tipo LIFO (Last In- First Out, o en buen romance, el último que entra es el primero que sale)

Piensa en un montón de platos, en el que sólo hay una forma de colocar un plato en el montón, que es situarlo en la cima, encima de todos los demás. Por otra parte, cuando necesitemos coger un nuevo plato del montón (pila de platos, hasta se llama así en el lenguaje natural) sólo puedes coger el plato que hay en lo alto del montón. Para coger cualquier otro plato, primero tendrás que quitar de encima (sacar de la pila) todos los que tiene encima.

Como ves, no es difícil entender el funcionamiento de esta estructura de datos. De forma más esquemática vamos a ver cuales son las características de una pila:

- **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 meter 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 sacar o 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 sacar o 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

Implementación de una pila en Java

Seguramente ya tendrás más o menos claro cómo se puede construir un programa que implemente esta estructura de datos, pero tenemos más de una opción disponible.

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.

Nosotros en este caso vamos a optar por la implementación mediante listas enlazadas. Las operaciones que debe tener disponibles el tipo Pila son las siguientes:

Operaciones básicas con pilas

Operación	Descripción
Meter (push)	Insertar un nuevo dato en la cabecera de la pila.
Quitar, o sacar (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.
Limpiar pila	Sacar todos los elementos y dejar la pila vacía.
Cima	Obtener el elemento de la cima de la pila.
Tamaño pila	Número máximo de elementos que puede contener.
Elementos	Consultar el número de elementos que tiene la pila.

No obstante, si se implementa la pila como una lista enlazada, no existe obligación de imponer un **tamaño máximo**, y por tanto no es necesario implementar la operación "pila llena", ya que siempre debería devolver el valor falso. Tampoco hay que fijar un tamaño máximo, aunque sí podemos implementar fácilmente una operación que nos indique cuantos elementos tiene la pila.

Al final del apartado tienes un enlace a un proyecto NetBeans en el que encontrarás una implementación en Java de una pila.

Imagina que queremos mantener la lista de mensajes con novedades, intercambiados entre un alumno y un tutor relacionados con el seguimiento del módulo profesional de Programación en Lenguajes Estructurados. La aplicación está hecha de forma que puede mostrar en pantalla un único mensaje en cada momento. Los tutores introducen los mensajes en la lista personalizada de cada alumno, y cada alumno retira los mensajes descargándolos de la lista una vez que los lee.

Puesto que se trata de novedades, es previsible que queramos obligar a que el alumno vea siempre primero los mensajes correspondientes a novedades más recientes, que son justamente las últimas que hemos introducido en la lista. La última novedad en entrar en la lista es la primera novedad en salir al descargarse y leerse (LIFO). Esto se hace cómodamente con una pila. Evidentemente, por simplificar, en el ejemplo el único usuario va a poder tanto meter como sacar los mensajes en la pila. Pero lo que importa es el funcionamiento de la estructura: El último introducido es el primero que se lee.

Descarga el proyecto ImplementacionPila

Autoevaluación

5. Colas.

Colas

__Carmen__ se interesa por lo que hace su compañero __Víctor__ y le aclara algunos conceptos que tiene un poco confusos cuando llega a las colas. __Carmen__ le explica que mientras que las pilas son una estructura en la que los elementos salen por el mismo lugar por el que han entrado, en las colas hay que distinguir una zona de entrada y otra de salida, de modo que un elemento sólo puede salir de la cola una vez que lo hayan hecho todos los que estaban por delante. También le aclara que una cola es como una lista enlazada en la que no pueden insertarse elementos entre dos existentes, el único modo de hacerlo es al final. Se trata por tanto de una estructura muy "educada" que no permite la existencia de elementos que se "cuelen".

Si el concepto de pila es algo habitual en la vida diaria, qué decir del concepto de **cola**. Seguro que sabes perfectamente lo que es una cola, y cual es su funcionamiento básico. Y es posible que hoy mismo hayas sido "un elemento de alguna cola":

- La cola del supermercado, o la carnicería o del pan.
- La cola del autobús.
- La cola de los coches en el atasco de entrada a una ciudad
- La cola de la ventanilla de venta de billetes de la estación de trenes o autobuses
- La cola de trabajos pendientes de imprimir que has enviado a una impresora conectada en red en la oficina, y que tiene más trabajo del que puede atender. La impresora sólo puede imprimir un documento cada vez, pero en cuanto finaliza un trabajo, saca de la cola el siguiente trabajo a imprimir, que será el que más tiempo lleva en la cola, el que primero se envió de todos los que quedan pendientes, y lo imprime. Y así, mientras que en la cola queden trabajos pendientes.
- La cola de espera para ser atendido por algún especialista de la seguridad social o ser operado de alguna dolencia.
- etc.

Al igual que en la vida diaria, en las aplicaciones informáticas es frecuente tener que programar para conseguir que la aplicación pueda atender las peticiones de servicio o a las peticiones de realización de determinadas tareas de múltiples usuarios, entendiendo como usuarios tanto a operadores humanos como a otras aplicaciones, y es habitual que esas peticiones sean atendidas en orden de llegada, de forma que el primero en entrar en la cola es el primero en ser atendido.

No hay más que pensar en una aplicación funcionando en red, a la que se conectan los trabajadores de distintas oficinas para trabajar con los mismos datos, almacenados en una base de datos. Es muy probable que existan momentos en los que varios trabajadores o usuarios soliciten acceder a la misma información para hacer modificaciones. Sólo un usuario podrá modificar los datos en cada momento, ya que de lo contrario no podríamos tener certeza de cuales son las modificaciones que finalmente quedarán como buenas. Incluso es posible que la aplicación tenga un máximo de usuarios que puedan usar de forma concurrente la aplicación, y que en un momento dado haya más usuarios intentando usar la aplicación de los que ésta puede atender.

En estas situaciones es útil crear una cola de tareas solicitadas, o una cola de usuarios pendientes de atender, de forma que en cuanto sea posible, se atenderá la tarea o usuario que primero llegó a la cola.

Otro ejemplo son los sistemas operativos. Cualquier ordenador doméstico suele tener un único procesador, lo que significa que en cada momento (cada ciclo de reloj del procesador) puede ejecutar una única instrucción, perteneciente a una única aplicación. Sin embargo hoy en día los **sistemas operativos** que usan estos ordenadores son multitarea, lo que quiere decir que tú puedes tener ejecutándose simultáneamente muchas aplicaciones. Por ejemplo, puedes estar descargándote música de Internet, al mismo tiempo que suena una canción de un CD en el reproductor multimedia, mientras que escribes un correo a un amigo con información que estás sacando de una página de Internet y de un documento que tienes abierto en el procesador de texto. Es un ejemplo nada rebuscado.

¿No habíamos dicho que el procesador sólo puede ejecutar una instrucción en cada momento? ¿Como puede tener en ejecución todas esas aplicaciones al mismo tiempo?

Mediante el sistema operativo, que forma una cola de procesos activos. El sistema operativo le asigna a cada proceso un cierto tiempo de CPU(algunos milisegundos o quantum) durante los cuales el procesador sólo atiende a un proceso. Pasado ese tiempo, le pasa el turno al siguiente proceso de la cola, y pasa al proceso anterior al final de la cola y así sucesivamente hasta que finaliza la ejecución de cada aplicación. Lo que ocurre es que el procesador es tan rápido, comparado con la velocidad a la que nosotros procesamos la información, que el usuario tiene la sensación de que cada una de esas tareas está recibiendo del procesador la atención en exclusiva. Parece que todas las aplicaciones se ejecutan a la vez.

El proceso puede complicarse bastante, por ejemplo gestionando una **cola de prioridades**. No todos los procesos ni todas las tareas tienen la misma prioridad, y el sistema operativo puede asignar una prioridad a cada una, de forma que avance en la cola por delante de todas las que son de menos prioridad, aunque hayan llegado después.

Volviendo a la realidad. Piensa en la cola de personas en espera de ser **operados**. Evidentemente, tendrá más prioridad aquella persona cuya vida corra peligro que aquella persona que tiene trastornos, pero que no suponen un riesgo vital serio. Por más que 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? Realmente la forma más simple quizás sea como 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

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, o en buen romance, 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 meter o insertar un nuevo elemento en la estructura por un extremo, el final de la cola.**
- **Sólo se pueden sacar o extraer elementos de la estructura por un extremo, que es justamente el extremo contrario, el comienzo de la cola, llamado cima 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

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.
  La lista circular de socios de un club deportivo, que hemos implementado en el ejemplo del final del apartado 3.3.2., puede considerarse de hecho como la implementación de una cola, en la que los elementos a los que "ya les tocó el turno" no se eliminan de la cola, si no que vuelven al final de la cola a la espera de un nuevo turno.

Nosotros en este caso vamos a optar por la implementación mediante listas enlazadas. 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.
Limpiar cola	Sacar todos los elementos y dejar la cola vacía.
Cabecera o Frente	Obtener el elemento de la cabecera de la cola.
Cola o Final	Obtener el elemento del final de la cola.
Tamaño cola	Número máximo de elementos que puede contener.
Elementos	Consultar el número de elementos que tiene la cola.

Como en el caso de las pilas, si se implementa la cola como una lista enlazada, no existe obligación de imponer un tamaño máximo, y por tanto no es necesario implementar la operación "cola llena", ya que siempre debería devolver el valor falso. Tampoco hay que fijar un tamaño máximo, aunque sí podemos implementar fácilmente una operación que nos indique cuantos elementos tiene la cola.

A continuación tienes un enlace a un proyecto NetBeans en el que encontrarás una implementación en Java de una cola.

Se trata de gestionar la lista de reserva de plazas para un curso gratuito de alemán que organiza el Ayuntamiento.

- Como el curso tiene una demanda enorme, muy superior a la oferta de plazas que el Ayuntamiento puede hacer, se han planteado sacar sucesivas convocatorias.
- Cada vez que se acabe un curso, se convocará e iniciará otro, hasta que se satisfaga toda esa demanda.
- Para ello han pensado que la inscripción en los cursos será por riguroso orden de llegada.
- Se creará una única lista de solicitudes, en la que figurará el nombre de la persona, el número de orden y la fecha y hora de solicitud.
- Cada vez que se convoque un nuevo curso, se irán sacando personas de la lista para matricularlas, hasta completar las plazas disponibles en el curso.

- Los que queden en la lista, quedarán a la espera de una nueva convocatoria.
- Si en el momento de terminar un curso no hubiera gente suficiente en la lista de reserva de plazas para iniciar otro, la lista permanecerá abierta hasta que se complete el número de personas necesarias para hacer el curso, de modo que la lista va a estar permanente e indefinidamente abierta para que la gente se apunte.

Evidentemente, una cola se ajusta a este problema perfectamente.

Descarga el proyecto ImplementacionCola

6. Árboles.

Árboles

*Finalmente **Víctor** se encuentra con los árboles. En principio parece que son algo totalmente diferentes al resto de estructuras de datos dinámicas, pero pronto descubre que sencillamente son como listas en las que un elemento puede tener varios siguientes. Evidentemente esto implica la aparición de nuevas operaciones y tareas sobre la estructura. Realmente **Víctor** no consigue ver utilidad en los árboles, como venía haciendo con cada una de las otras estructuras y así se lo comenta a **José**. Éste le explica que son muy útiles y que quizás el uso más conocido sea en los juegos de estrategia, especialmente en el ajedrez. Un ordenador que está "jugando" al ajedrez elabora para cada movimiento del contrario, una estructura en árbol para determinar el movimiento que a la larga le aportará una situación más ventajosa. Añade que hay otros juegos de ordenador que utilizan estas técnicas, y que también son muy utilizadas por ejemplo en estudios de estrategias de mercado.*

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 no todo es siempre tan lineal, y 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 organigrama de una empresa (o de cualquier institución) en la que hay un director general, que tiene a su cargo a varios subdirectores, que tienen a su cargo a varios jefes de sección, que tienen a su cargo varios empleados, etc.
- 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.
- Los distintos componentes de una expresión matemática, en la que existe un operador y dos operandos asociados a él, que pueden ser a su vez subexpresiones formadas por un operador y dos operandos, etc.
- Un árbol para ordenar una serie de elementos mejorando el tiempo de búsqueda.
- Un árbol para implementar de forma sencilla una cola con prioridad
- Un árbol para mejorar la eficiencia del manejo de índices en Sistemas Gestores de Bases de Datos.
- 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. Tal es el caso de los árboles de búsqueda. Por **ejemplo**, piensa en 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.

Especificación de un árbol

Como podrás comprobar, hasta 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 evidente relación entre esta estructura de datos y la forma de relacionarse jerárquicamente que los datos tienen realmente 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.

Conviene que tengamos en cuenta que 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 de un árbol binario se tratara, (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 y sus algoritmos.

6.1.1. Representación sin referencias y punteros

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

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.

El propósito de esta unidad no es ver de forma exhaustiva todos los algoritmos de inserción, búsqueda, eliminación y recorrido de todos los tipos de árboles que existen, ni siquiera de todos los tipos principales. Tampoco pretendemos ver exhaustivamente las múltiples aplicaciones que poseen. Basta con que adquieras la noción de cómo se construye una estructura de datos en árbol, cuales son los tipos principales de árboles, qué utilidades pueden tener, y para qué se usan por lo general. Sólo es una aproximación, que podrás completar consultando los enlaces para saber más.

PARA SABER MÁS:
En este interesante enlace encontrarás la implementación de los distintos eventos para el tratamiento de árboles en Java
Árboles en Java

6.2.1. Árboles binarios

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 ideales para comprender bien los algoritmos de manejo de árboles, y extender lo que aprendas con ellos para aplicarlo a algoritmos de otro tipo de árboles, no es complicado. Sólo un poco más largo. A fin de cuentas, los árboles binarios no son más que un tipo particular de árbol.

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

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.

¿Cómo pueden ayudarnos a buscar más eficientemente un elemento entre todos los que forman parte de la estructura?

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 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 log2 n comparaciones para encontrar el valor buscado. Se dice que la **eficiencia del algoritmo es del orden de $\log_2 n$**. Esto se suele expresar como que su **eficiencia** es **$O(\log_2 n)$**.

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 **$O(\log_2 n)$**, 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 (**$\log_2 128=7$**). Pero para el doble de elementos, 256, sólo tendremos que hacer una comparación más, ya que **$\log_2 256=8$**) Así, para 16384 elementos habría que hacer del orden de 14 comparaciones. Fíjate 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 **$O(\log_2 n)$**, 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)**

Al final de esta unidad tienes un enlace a un ejemplo de la implementación de un árbol binario de búsqueda. Ahí se comprobarán las operaciones que se pueden realizar con árboles, en general, y con árboles binarios de búsqueda en particular.

6.2.3. Árboles binarios enhebrados

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, para indicar que esa rama no existe, que el camino termina ahí.

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?

Piensa 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, con N niveles, veamos algunos datos: (no contamos la referencia al nodo raíz.)

- Contiene en total 2^n -1 nodos. Como cada nodo tiene dos referencias, en total $2^{(n+1)}$-2 referencias. Ejemplo: con 10 niveles, tendrá 1023 nodos, y 2046 referencias
- Cada nivel contiene $2^{(n-1)}$ nodos. El último nivel, el de las hojas, contiene $2^{(n-1)}$ 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^n 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, quédate con la idea de que aproximadamente la mitad de las referencias de un árbol binario de búsqueda apuntan a null, inevitablemente.

¿No podrían usarse para apuntar a algo más interesante que a null?

Esta es la idea básica de los árboles enhebrados. Se aprovechan 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. A continuación te decimos lo que debes tener en cuenta para enhebrar un árbol para un recorrido "**inorden**":

- **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 la 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.

¿Merece la pena? Ni sí, ni no. Como siempre, depende del uso que vayamos a hacer del árbol. Si las operaciones más frecuentes son inserciones y borrados, y el recorrido completo del árbol es poco frecuente, no la merece.

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.

PARA SABER MÁS:
En el siguiente enlace a un documento pdf que trata sobre árboles en general, encontrarás información detallada sobre los árboles enhebrados, a partir de su página 49.
Árboles Enhebrados [Versión en caché]

6.2.4. Árboles binarios en montón (heap)

Árboles binarios en montón (heap)

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

Hay un algoritmo de ordenación especialmente elegante, 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. Imagina 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 $\mathbf{O(nlog_2n)}$, tanto para el peor de los casos como para el caso medio. Es mejor que la ordenación por burbuja, con una eficiencia de $\mathbf{O(n^2)}$ e incluso que la ordenación rápida, que tiene una eficiencia para el caso medio también de $\mathbf{O(nlog_2n)}$, pero que tiene una eficiencia de $\mathbf{O(n^2)}$ para el caso menos favorable.

Evidentemente, 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. De nuevo, gracias a sus especiales propiedades.

Veamos en qué consiste un árbol en montón, y dónde puedes encontrar información detallada de cómo construirlo.

- 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)

- 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.

Los detalles sobre los algoritmos de inserción y eliminación en un montón, puedes consultarlos en el enlace siguiente.
Árboles en Montón [Versión en caché]

6.2.5. Árboles B y B+

Árboles B y B^+

¿Has pensado alguna vez que una base de datos puede contener miles, incluso cientos de miles de registros? Buscar un registro determinado en medio de tantos no debe ser fácil. Y no se trata de una operación rara, sino todo lo contrario a veces es más usual de lo que sería deseable.

Si 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, podemos imaginar la desesperación del usuario que quiere consultar un dato, si las cosas no se hacen bien.

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.

PARA SABER MÁS:
En el siguiente enlace podrás profundizar sobre los árboles B y B^+
Árboles B y B+ [Versión en caché]

DEMO: Visualiza una pequeña explicación de los árboles B

DEMO: Visualiza la eliminación de elementos en un árbol B

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

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 mismo 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**. ¿Cómo puedo estar seguro de haber recorrido todos los nodos de un árbol sin dejarme ninguno atrás? Y lo que es igual de importante, ¿cómo puedo 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. La condición de parada es que el puntero que apunta al raíz valga null.

Usado 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. La condición de parada sigue siendo la misma, que el puntero que apunta al raíz valga null.

Usado 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. Sería posible, como en todos los casos de uso de la recursividad, hacer una implementación iterativa, usando una pila. Pero la solución recursiva en este caso es mucho más clara y evidente, y tan eficiente como la iterativa. En el ejemplo del apartado siguiente, será la solución recursiva la que usaremos.

6.4. Implementación de un árbol binario de búsqueda en Java

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 destacado por el que comienza la estructura (la raíz del árbol)

En la siguiente tabla tienes esas operaciones indicando al lado una descripción y el método que las implementa en el ejemplo del enlace que hay tras esta tabla (ImplementacionArbolBinario).

Operaciones básicas con un árbol binario de búsqueda

Operación	Descripción	Métodos que implementan la operación en el ejemplo.	
		Clase GestionArbol	Clase Arbol
Añadir	Insertar un nuevo dato en el árbol.	**añadir()**	**añadirNodo()**
Eliminar	Borrar un dato del árbol.	**eliminar()**	**eliminarNodo()**
Árbol Vacío	Comprobar si el árbol está vacío.	**arbolVacio()**	**arbolVacio()**
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.	**listadoMayorMenor()**	**preorden()**
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)	**listadoDesordenado()**	**inorden()**
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. .	**listadoMenorMayor()**	**postorden()**
Búsqueda	Buscar y consultar la información del elemento deseado del árbol.	**buscar()**	**buscarNodo()**
Raíz	Consultar la información del nodo raíz.	**consultarRaiz()**	**consultarRaiz()**
Elementos	Consultar el número de elementos que tiene el árbol.	**consultarTotalNodos()**	**obtenerTotalNodos()**

En el **ejemplo** puedes comprobar que la mayoría de los métodos son recursivos. Es posible también hacer una definición iterativa de los mismos, pero es más complicada.

Es especialmente interesante que sigas el algoritmo de borrado, que quizás resulte el más complicado de todos.

Básicamente debes tener en cuenta que al borrar un nodo:

- Si el nodo a borrar no tiene hijos, basta con poner a null su referencia
- Si el nodo a borrar sólo tiene 1 hijo izquierdo, basta con que su referencia apunte a ese hijo izquierdo
- Si el nodo a borrar sólo tiene 1 hijo derecho, basta con que su referencia apunte a ese hijo derecho.
- Si el nodo a borrar tiene dos hijos, es algo más complicado. Hay que sustituir al nodo raíz por el sucesor inorden (el menor de los mayores) o por el antecesor inorden (el mayor de los menores), para que siga siendo un árbol binario de búsqueda. En el ejemplo hemos optado por sustituirlo por su sucesor inorden, es decir, el menor de los mayores.
    - Si el hijo derecho del nodo a borrar no tiene hijo izquierdo, él será el sucesor inorden del nodo a borrar
    - Si el hijo derecho sí tiene hijo izquierdo, habrá que seguir esa rama siempre por la izquierda hasta el final, donde encontraremos el sucesor inorden, que por definición no puede tener hijo izquierdo. (si lo tuviera, sería menor, y por tanto sería éste el menor de los mayores (sucesor inorden)
    - Una vez localizado el sucesor inorden, debe ponerse en el lugar del raíz, y para ello hay que hacer:
        - Que la referencia al hijo izquierdo del sucesor inorden apunte al hijo izquierdo del nodo a borrar.
        - Que el hijo izquierdo del padre del sucesor inorden apunte al hijo derecho del sucesor inorden.
        - Que la referencia al hijo derecho del sucesor inorden apunte al hijo derecho del nodo a borrar.
        - Que la referencia al nodo a borrar apunte al sucesor inorden.

Te sugerimos que pruebes el programa introduciendo personas con los siguientes nombres en el orden que se indican:

1. Gonzalo - 2. Daniel - 3. Juan - 4. Francisco - 5. Inés - 6. Blas - 7. Luis - 8. Ana - 9. Manuel - 10. Carlos - 11. Kirukou - 12. Hilario - 13. Enrique - 14.Xavi

El recorrido inorden nos mostrará los datos ordenados alfabéticamente.

El recorrido Preorden mostrará la lista (indicamos iniciales) G-D-B-A-C-F-E-J-I-H-L-K-M-X

El recorrido Postorden mostrará la lista : A-C-B-E-F-D-H-I-K-X-M-L-J-G

Aquí tienes el enlace al ejemplo de implementación del un árbol binario en Java. Como siempre, es recomendable que además de comprobar cómo funciona, leas el código con atención, entendiendo las operaciones que se realizan con la ayuda de las explicaciones incluidas como comentarios dentro del código. Imprimir las clases de los ejemplos también puede resultar de ayuda para hacer esa lectura, sobre todo si te resulta pesado hacer la lectura en pantalla, cambiando de unas clases a otras. Pero piensa en todo el papel y la tinta que vas a gastar..., y en tu bolsillo, y en el medio ambiente.

Descarga el proyecto ImplementacionArbolBinario

PARA SABER MÁS.
En este enlace podrás ver de una forma global las estructuras de datos vistas hasta ahora y las distintas diferencias entre ellas.
Estructuras de Datos [Versión en caché]
En este enlace encontrarás de forma resumida, cómo implementar la mayoría de las estructuras vistas en esta unidad.
Estructuras de Datos y Algoritmos en Java [Versión en caché]