P1. (2 Puntos) **Pregunta sobre la práctica**. Supongamos que se añade una operación en **`PlayerIF`** tal que permite generar de forma automática una lista de reproducción para cada año que contenga todas las canciones publicadas ese año, siempre que hubiera canciones en el repositorio publicadas ese año. Por ejemplo: generaría una lista de reproducción con identificador "1974" con todas las canciones publicadas en 1974, otra con identificador "1969" con todas las canciones publicadas en 1969… y así sucesivamente, de forma que todas las canciones del repositorio pertenecerían a una de estas listas de reproducción. Si, por ejemplo, no existiera ninguna canción en el repositorio publicada en 1981, no se debería crear la lista con identificador "1981". Indique cómo se podría realizar esta operación de manera eficiente y qué consecuencias tendría en el resto de componentes del reproductor (como añadir o modificar métodos de alguna clase) en el código. Justifique su respuesta.

Dado que nuestro reproductor tiene una operación que permite añadir a una lista de reproducción todas las canciones del repositorio que cumplan unos criterios de búsqueda, si realizamos una búsqueda de todas las canciones que fuesen publicadas en un año concreto, podríamos crear la lista de reproducción de ese año de forma automática.

Se podría repetir el proceso anterior para cada año para el que hubiera canciones, pero esto plantea dos problemas fundamentales:

1. A priori no sabemos cuales son los años para los que existen canciones en el repositorio, lo que nos llevaría a hacer un análisis de su contenido para averiguar este dato.

2. El proceso descrito es innecesariamente ineficiente. Para cada búsqueda se debería recorrer el repositorio por completo. En el peor de los casos, para un total de n canciones en el repositorio, estaríamos hablando de un coste $O(n^2)$ (suponiendo que cada canción del repositorio haya sido publicada en un año diferente) sólo recorriendo el repositorio.

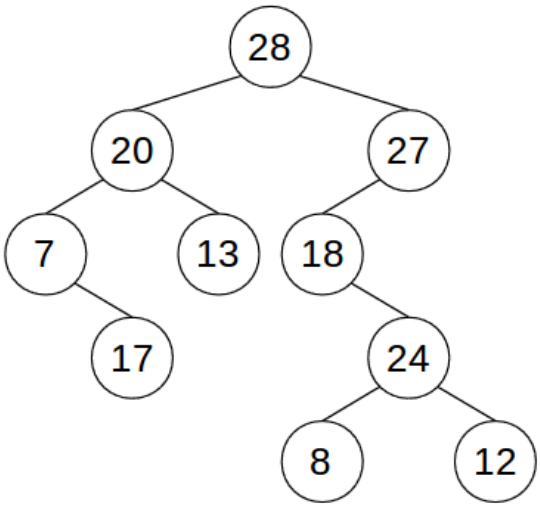
Una solución más eficiente consistiría en realizar un único recorrido del repositorio de canciones y para cada canción existente obtener su año de publicación (lo que nos lleva a la necesidad de un getter de este atributo dentro de la clase que implemente `TuneIF`).

Una vez tenemos el año de publicación, se podría crear la lista de reproducción cuyo identificador coincida con el año (si ya existe no hay problemas, pues la operación de creación de una lista no hace nada si ya existe una lista con ese identificador) y meter la canción en ella. De esta forma, se realiza un único recorrido del contenido del repositorio, que supone un coste O(n), porque la obtención de la canción y, por ende, del año de su publicación, tienen un coste independiente del tamaño del problema.

Para cada canción, habría que tener en cuenta los costes adicionales de comprobar si la lista de reproducción de ese año existe o no y de insertar el identificador de la

canción en ella). Comoquiera que estos costes serían equivalentes para ambas aproximaciones, la solución con un único recorrido sería más eficiente.

1. Considérese el siguiente árbol binario:



a) (0.75 Puntos) Especifique cuáles serían los recorridos de dicho árbol binario en preorden, postorden e inorden.

Recorrido en preorden: en primer lugar la raíz, luego el hijo izquierdo y, por último, el hijo derecho: 28, 20, 7, 17, 13, 27, 18, 24, 8, 12.

Recorrido en postorden: en primer lugar el hijo izquierdo, luego el hijo derecho y, por último, la raíz: 17, 7, 13, 20, 8, 12, 24, 18, 27, 28.

Recorrido en inorden (o en orden): en primer lugar el hijo izquierdo, luego la raíz y, por último, el hijo derecho: 7, 17, 20, 13, 28, 18, 8, 24, 12, 27.

Nótese, a modo ilustrativo, que este árbol no es un árbol binario de búsqueda (o árbol de búsqueda binaria). Algo que podemos comprobar directamente ya que este último recorrido no produce una secuencia ordenada.

b) (1.25 Puntos) ¿Cuál sería la forma más conveniente de recorrer un árbol binario en cada uno de los siguientes supuestos? [Nota: No tiene por qué ser necesariamente uno de los recorridos citados]

1. Se busca un elemento que debe estar en un nodo hoja.

   Si el elemento debe estar en un nodo hoja, lo más sensato es emplear un recorrido en profundidad. Dentro de los recorridos habituales, sería preferible un recorrido en postorden ya que dejaríamos para el final la consulta de la raíz.

2. Se busca un elemento que no puede estar en un nodo hoja.

   Sabemos que el elemento no puede estar en un nodo hoja, pero no sabemos si el elemento está cerca de las hojas o de la raíz. En este caso, cualquier recorrido nos serviría. No obstante, ya que no está en las hojas (y éstas podrían ser muchas), una modificación de un recorrido en anchura en el que no se consideren las hojas, podría ser beneficiosa.

3. Se busca un elemento que tiene más probabilidad de encontrarse cerca de la raíz.

   En este caso, dado que el elemento es más probable encontrarlo cerca de la raíz, es preferible un recorrido en anchura sobre un recorrido en profundidad, ya que así recorreremos los niveles cercanos a la raíz antes que el resto del árbol.

4. Se trata de un árbol de enteros en el que se cumplen las siguientes condiciones:

   - El valor de cada nodo es siempre mayor que el de cualquiera de sus hijos.
   - El valor de cualquiera de los nodos de uno de los hijos es siempre mayor que el valor de cualquiera de los nodos del otro hijo.

   Queremos recorrerlo para generar la lista ordenada (de mayor a menor) de los enteros que componen el árbol.

   En este caso, tenemos que recorrer primero la raíz (cuyo valor es mayor que el de cualquiera de sus hijos) y luego recorrer (recursivamente, empleando este mismo recorrido) el árbol cuya raíz es el hijo con mayor valor, ya que todos sus nodos tienen un valor mayor que cualquiera de los nodos del otro hijo. Una vez recorrido ese hijo, recorreremos el otro (nuevamente, con este mismo recorrido) y así obtendremos la lista ordenada de los valores del árbol.

2. (2 Puntos) Se dice que un árbol general es k-ario si cada uno de sus nodos tiene a lo sumo k hijos. Implemente en JAVA una función

   ```
   int getMinArity(TreeIF<T> tree)
   ```

   que obtenga el mínimo valor de k para el que se cumple que el árbol dado por parámetro es k-ario.

   Suponiendo que el árbol no es vacío, la idea es ir recorriendo todos los sus nodos y comprobar cuál es su número de hijos, para quedarnos con el máximo valor de todos ellos. Para ello, realizaremos el siguiente proceso para un nodo cualquiera:

   - Calcular el número `k` de hijos de ese nodo.
   - Para cada uno de ellos, calcular (recursivamente) la aridad a_i de dicho hijo.
   - Devolver el máximo entre `k` y todos los a_i para `i` desde `1` hasta `k`.

   Así pues, el código del método sería el siguiente:

```
int getMinArity(TreeIF<T> tree) {
  ListIF<TreeIF<T>> children = tree.getChildren();
  int max = children.size();
  Iterator<TreeIF<T>> it = children.iterator();
  while ( it.hasNext() ) {
    int a = getMinArity(it.getNext());
    if ( a > max ) { max = a; }
  }
  return max;
}
```

3. Se desea implementar una estructura de datos que almacene los contactos de una red social (tipo Facebook) en la que cada usuario se puede identificar por un número natural. Se deben poder realizar las siguientes operaciones:

   O1) Añadir contactos (el usuario `i` y el usuario `j` son ahora amigos).

   O2) Eliminar contactos (el usuario `i` y el usuario `j` han dejado de ser amigos).

   O3) Preguntar por todos los amigos de un usuario.

   O4) Preguntar si dos usuarios son amigos en la red.

   O5) Preguntar cuáles son los amigos en común de dos o más usuarios.

   Bajo las siguientes suposiciones:

   S1) El número medio de contactos de una persona es varios órdenes de magnitud más bajo que el número de personas en la red social.

   S2) Es mucho más frecuente preguntar por los contactos de una persona que preguntar si dos personas están en contacto.

   Se pide [Nota: no es necesario codificar en Java en los apartados b y c]:

   a) (1 Punto) Especificar una interfaz **`ContactsIF`** adecuada para el problema.

      Para las siguientes operaciones, se tendrá como precondición que el(los) índice(s) (consideremos, sin pérdida de generalidad, que éstos son de tipo `int`) que se le pasen como parámetro han de corresponderse con usuarios que existan realmente en la red social. Así pues tendremos:

      - **O1:** `void addContact(int i, int j);`
      - **O2:** `void removeContact(int i, int j);`
      - **O3:** `ListIF<Integer> getAllContacts(int i);`
      - **O4:** `Boolean isContact(int i, int j);`

      Adicionalmente, para el siguiente método, se tendrá que el tamaño de la lista de entrada es de, al menos, dos elementos, que todos se corresponden con índices de usuarios existentes en la red social y que ninguno de ellos estará repetido (aunque esto último no sería estrictamente necesario, piense por qué):

      - **O5:** `ListIF<Integer> getAllcommonContacts(ListIF<Integer> users);`

   b) (1.5 Puntos) Supongamos que se implementa esta interfaz con una clase en la que la información de contactos se almacena mediante una matriz `NxN` de booleanos, en la que `N` es el número de usuarios de la red y el elemento `i,j` de la matriz indica si los usuarios i y j están conectados o no.

      1. ¿Cómo se implementarían las operaciones O3 y O4?

         Comenzando por la operación O4, su implementación sería muy sencilla, ya que bastaría con devolver directamente el valor (i,j) de la matriz.

         Para la operación O3 habría que recorrer la fila i-ésima de la matriz y, para cada elemento (i,j) si contiene un valor verdadero añadir el índice j a la lista de amigos del usuario i. Si contuviese un valor falso, no se haría nada.

2. ¿Cuál sería el coste esperable (en una implementación razonable) para estas dos operaciones, en función del número de usuarios `N` y el número de amigos de cada usuario?

   El coste de la implementación de la operación O4 descrita anteriormente sería constante con respecto al tamaño del problema, si asumimos que la matriz está implementada como un vector estático de NxN posiciones con acceso directo a cada una de ellas.

   Por otro lado, el coste de la implementación de la operación O3 descrita, sería lineal con respecto al número `N` de usuarios de la red social, ya que tendríamos que recorrer toda la fila i-ésima (compuesta por `N` usuarios) y comprobar uno a uno si son (o no) amigos del usuario `i`. Además, cada uno de los amigos deberá insertarse en una lista. Si denotamos por `A` el número de amigos de un usuario, entonces el coste de esta operación sería lineal con respecto al producto de `NxA` (pues el coste de insertar un elemento en una lista es lineal con respecto a su tamaño). En este caso, dado que `N` es varios órdenes de magnitud mayor que `A` (según la suposición S1), podríamos simplificarlo a `O(N)`.

3. ¿Es este coste razonable, a la vista de las suposiciones S1 y S2? ¿Por qué?

   El de la operación O4 sí lo es, pero según la suposición S2 es una operación menos frecuente que O3, cuyo coste es muy elevado, ya que (según nos indica la suposición S1) `N` es varios órdenes de magnitud mayor que `A`.

c) (1.5 Puntos) Proponer una implementación que emplee una estructura de datos alternativa más acorde con las suposiciones S1 y S2.

   Según la suposición S2, la operación más frecuente va a ser O3, por lo que se deberá primar la eficiencia de dicha operación. Para eso, podríamos almacenar directamente las listas de amigos de cada uno de los usuarios de la red, de forma que el coste de ejecutar la operación O3 se redujese.

   Una primera aproximación sería utilizar una lista en la que cada elemento sea, a su vez, una lista de amigos. Así, para implementar O3 tendríamos que recorrer la lista hasta situarnos en el elemento que contiene la lista de amigos del usuario buscado. Sin embargo, si en lugar de almacenar estas listas en otra lista lo hiciéramos en una matriz unidimensional, el coste de acceso a la lista de amigos de cualquier usuario sería independiente a cualquier tamaño de problema, por lo que será esta estructura la que proponemos.

   Además, de cara a mejorar la eficiencia de las operaciones O4 y O5, exigiremos que los índices almacenados en las listas de amigos estén ordenados de forma creciente.

   1. Explicar cómo se implementarían las operaciones O3, O4, y O5.

      Con la estructura propuesta, la implementación de O3 se limitaría a devolver el contenido en la posición i-ésima de la matriz.

En cuanto a la operación O4, habría que obtener la lista de amigos del usuario `i` y buscar en ella el índice `j`. Si existe, entonces `j` es amigo de `i` (y habría que devolver verdadero). Si no existe, entonces los usuarios `i` y `j` no son amigos. Dado que la lista de amigos de cualquier usuario está ordenada, podemos detener la búsqueda en cuanto nos encontremos con un índice superior a `j`.

Por último, para realizar la operación O5 deberemos comenzar por inicializar la lista de amigos comunes (llamémosla `lac`) con la lista de amigos del primer usuario presente en la lista de entrada `users`.

A continuación, para cada uno de los demás usuarios presentes en la lista de entrada `users`, actualizaremos `lac` al resultado de intersecar `lac` con la lista de amigos de ese usuario. Repetiremos este proceso hasta intersecar la lista de amigos de todos los usuarios presentes en la lista de entrada `users` o hasta que `lac` sea la lista vacía, momento en el cual ya nunca más se modificará.

Por último, devolveremos `lac` como resultado de la operación.

2. Razonar sobre el coste esperado de las operaciones O3 y O4, y compararlo con el de la primera implementación.

   El coste de ejecutar la operación O3 sería constante con respecto al tamaño del problema, ya que el acceso a una posición de una matriz se realiza en tiempo constante y el contenido de dicha posición ya es el resultado de la operación. Este coste es mucho mejor que el de la primera implementación.

   En cuanto al coste de O4, primero tendríamos que acceder a la lista de amigos del usuario i (coste constante) y luego recorrer dicha lista para buscar si el usuario j está presente. Este recorrido supone un coste lineal con respecto al número A de amigos de un usuario, para cada uno de los cuales realizaremos una serie de operaciones (comparación del valor con el índice buscado) de coste constante. Por lo tanto, el coste de O4 estará ahora en O(A). Es un coste mayor que el de la implementación anterior, sin embargo, dado que esta operación es menos utilizada que O3, podemos concluir que esta segunda implementación es más eficiente.