P1. (2 Puntos) **Pregunta sobre la práctica**. Supongamos que se añade una operación en **`TuneCollectionIF`** tal que permita eliminar una canción del repositorio a partir de su identificador. Indique qué consecuencias tendría esta operación en el resto de componentes del reproductor y cuáles serían los cambios a realizar (como añadir o modificar métodos de alguna clase) en el código. Justifique su respuesta.

En primer lugar, hay que indicar que el identificador a eliminar debe estar en el repositorio, es decir, existe una canción en el repositorio que puede referenciarse mediante dicho identificador. Una vez hecho esto, hay que decidir qué se hace con esa canción presente en el repositorio. Recordemos que éste se representa mediante un array de canciones, en el que el índice donde se almacena cada canción se corresponde con el identificador (menos 1, ya que los índices de los array comienzan en 0 y los identificadores de las canciones en 1).

Hay dos opciones:

1. Marcar la posición adecuada del array como eliminada.
2. Desplazar todas las canciones presentes en el repositorio con un índice superior una posición hacia abajo en el array.

En cualquier caso, el borrado de una canción en el repositorio implica:

i. Eliminar toda aparición del identificador eliminado de todas las listas de reproducción y de la cola de reproducción, ya que estas estructuras sólo almacenan identificadores de canciones presentes en el repositorio. Opcionalmente también de las últimas canciones reproducidas, aunque podría quedar como un histórico.

ii. Para ello hay que incluir una nueva operación que elimine toda aparición de un identificador de la cola de reproducción, similar a la que realiza esta operación en las listas de reproducción.

iii. Se debe añadir, también, una operación en la clase Player que realice las llamadas oportunas a las operaciones de eliminación del identificador en todas las listas de reproducción y en la cola de reproducción.

iv. Si se ha optado por la opción 1, se deberá tener cuidado a la hora de añadir identificadores a las listas de reproducción o a la cola de reproducción, porque no se podrá añadir un identificador que referencie a una canción eliminada. Esto afectaría a las operaciones que realizan búsquedas en el repositorio.

v. Si se ha optado por la opción 2, se deberá modificar el contenido de todas las listas de reproducción y de la cola de reproducción para restarle 1 a todo identificador con un valor superior al identificador borrado. De esta forma se mantiene la consistencia entre el contenido de estas estructuras y el del repositorio de canciones.

1. Considérese la siguiente función recursiva:

```
nips (1) = 1
nips (n) = nips (n-1) + nips (n-2) + nips (n-3)
```

a) (0.75 Puntos) Comente cuáles son las partes esenciales de cualquier programa recursivo, ejemplificándolas mediante la definición de `nips`.

Todo programa recursivo debe tener:

- Caso base, que puede ser uno o varios. Es aquel caso en el que los datos de entrada no necesitan ser reducidos para poder dar una respuesta. En `nips`, tenemos que la primera línea es un caso base, que se puede identificar porque el valor del parámetro de entrada es `1`. Es decir, se retorna un valor inmediato sin necesidad de más elaboración (nuevas llamadas).

- Caso recursivo, que puede ser uno o varios. Es aquel caso en el que los datos de entrada deben ser reducidos para poder dar una respuesta. Además, esta reducción debe ser estricta, es decir, la(s) llamada(s) recursiva(s) debe(n) realizarse con datos estrictamente más pequeños que los recibidos en la llamada actual en algún sentido claramente definido, observable en los datos y utilizable computacionalmente. En `nips`, tenemos que la segunda línea es un caso recursivo. En él, para cualquier valor de entrada `n` que no sea `1`, se realizan tres llamadas recursivas con valores estrictamente menores que el valor de entrada `n`.

De esta manera, los datos de las sucesivas llamadas recursivas se irán reduciendo y aproximándose a los del caso base (o a los de alguno de los casos base). También debe de haber una alternativa que cubra todos los casos (permitidos por la precondición) y además, tanto los casos recursivos como el trivial deben ser **correctos** (es decir, deben llevar al resultado esperado)

b) (0.5 Puntos) La definición de la función `nips` es incorrecta. ¿Por qué?

Esta definición es incorrecta porque para cualquier valor de entrada distinto de `1` no se puede calcular su valor de salida.

Por un lado, cualquier valor de entrada menor que `1` provocaría una cadena descendente infinita de llamadas recursivas. Por otro lado, todo valor de entrada superior a `1` provocaría en algún momento la llamada recursiva `nips(3)` ó `nips(2)`, las cuales requieren calcular el valor de `nips(0)`, que provoca esa cadena infinita de llamadas recursivas.

c) (0.75 Puntos) Complete la definición de `nips` de forma que siempre termine [Nota: hay múltiples formas de hacerlo]. ¿Cuál sería el coste asintótico temporal en el caso peor de un programa que implemente `nips` según esta definición?

Hay múltiples formas de arreglar el problema y, sin tener más información sobre los valores que debe devolver la función nips, todas ellas serían válidas siempre que cubran estos dos puntos fundamentales:

- Dar un valor explícito a `nips(3)` y `nips(2)` (de manera similar a `nips(1)`), con lo que cualquier llamada con un valor de parámetro superior a `0` ya sería calculable.

- Dar un valor explícito a cualquier llamada con un valor de entrada inferior a `1` (con lo cual también se cubriría el punto anterior), o bien indicar que no se aceptan esos valores de entrada.

A la hora de calcular el coste, establecemos en primer lugar el tamaño del problema, que no puede ser otro que el valor del parámetro recibido: `n`.

Estamos ante un programa recursivo que reduce el problema mediante substracción, por lo que la forma general de la recurrencia y las reglas prácticas para el cálculo del orden son las siguientes:

$$T(n)=\begin{cases} c_b(n) & si\ 0\le n<b \\ a\cdot T(n-b)+c_{nr}(n) & si\ n\ge b \end{cases}$$

Forma general de la recurrencia

$$T(n)\in\begin{cases} O(n\cdot c_{nr}(n)+c_b(n)) & si\ a=1 \\ O(a^{(n\ \mathbf{div}\ b)}\cdot(c_{nr}(n)+c_b(n))) & si\ a>1 \end{cases}$$

Reglas prácticas para el cálculo del orden

Podemos ver que cualquier llamada recursiva que no desemboque en un caso trivial, provoca otras tres llamadas recursivas que reducen el problema en `1`, `2` y `3` unidades respectivamente. Quedándonos con el peor valor de los tres, podemos asumir que el factor de reducción del problema es `1`.

Así pues, tenemos que `a = 3` (número de llamadas recursivas) y `b = 1` (factor de reducción del problema).

Por otro lado, el coste del caso trivial es constante con respecto al tamaño del problema, siempre que los casos triviales que añadamos para completar la definición de `nips` sean similares al que ya hay. También el coste de las operaciones no recursivas en la rama recursiva es constante con respecto al tamaño del problema (consisten en sumas y restas).

Por lo tanto, atendiendo a las reglas prácticas para el cálculo del orden, el coste de este programa estaría en $O(3^n)$

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

```
boolean isKAryTree(TreeIF<T> tree, int k)
```

que decida si el árbol dado por parámetro es k-ario.

Este problema puede resolverse de muy diversas formas, aunque aquí mostraremos una versión recursiva que implementa el siguiente algoritmo:

- Si el árbol es vacío, directamente cumple la definición de ser k-ario (ya que ninguno de sus nodos tiene más de k hijos). Por lo que se devolvería true.

- Si el árbol **no es vacío**, **se puede acceder** a la lista de sus hijos.

- Si la lista de hijos tiene un tamaño superior a k, entonces no es k-ario (y se devolvería false).
- Si hay, a lo sumo, k hijos, se ha de comprobar que **todos ellos** son árboles k-arios de forma recursiva con una llamada a la propia función. Para ello, se puede iterar sobre la lista de hijos del nodo (que son, a su vez, árboles).

El código sería el siguiente:

```
boolean isKAryTree(TreeIF<T> tree, int k) {
  if ( tree.isEmpty() ) { return true; }
  ListIF<TreeIF<T>> hijos = tree.getChildren();
  IteratorIF<TreeIF<T>> it = hijos.iterator();
  boolean allKAry = true;
  while ( it.hasNext() && allKAry ) {
    allKAry = isKAryTree(it.getNext(),k);
  }
  return allKAry;
}
```

Se podría pensar que otra forma de hacerlo sería generar un iterador de todo el árbol:

```
IteratorIF<T> iter = tree.iterator();
```

e ir consultando si existe algún nodo que tenga más de k hijos, en cuyo caso el árbol no sería k-ario. Sin embargo, esto no es correcto, ya que el iterador `iter` nos devuelve la secuencia de los valores contenidos en cada nodo (clase `T`), no la secuencia de los nodos (clase que implemente `TreeIF<T>`). Así, en el caso de un árbol de enteros, el iterador nos devolvería una secuencia de enteros, no la secuencia de los sub-árboles cuyas raíces contienen esos enteros. Por lo tanto esa solución **no sería válida**.

3. Se desea almacenar un conjunto muy grande de números en formato binario, con las siguientes características:
   - No hay números repetidos
   - No hay un tamaño máximo preestablecido ni para cada número individual ni para el conjunto de números.
   - La operación más frecuente sobre esa colección será preguntar si un determinado número está o no está incluido en ella.

   Considérense las siguientes implementaciones alternativas:

   - Los números se almacenan mediante una lista de números binarios, utilizando una clase que implemente **`ListIF`**.
   - Los números se almacenan mediante un árbol binario en el que cada nodo puede ser:
     - nodo inicio (reservado para la raíz del árbol)

- nodo conteniendo un dígito del número (1 ó 0)

Un número determinado (por ejemplo, 1101) se busca recorriendo el árbol desde el nodo inicio, buscando en cada nivel del árbol el dígito siguiente. Si existe el recorrido (en el ejemplo: inicio → 1 → 1 → 0 → 1), entonces el número está en la colección.

Se pide [Nota: no es necesario codificar en Java]:

a) (1.5 Puntos) La segunda implementación tiene un problema: en el momento de búsqueda puede encontrar números que no están en la colección. ¿Por qué? ¿Cómo se podría solucionar?

Vamos a ilustrar el problema de la segunda implementación con el mismo ejemplo del enunciado. Supongamos que en la colección sólo se ha añadido el número 1101, por lo tanto, sabemos que existe el camino:

inicio → 1 → 1 → 0 → 1

pero también existe el camino:

inicio → 1 → 1 → 0

por ejemplo. Entonces esta representación nos diría que el número 110 está en la colección, cuando no lo hemos añadido. El problema es que ese camino es un prefijo del número 1101 que sí está en la colección.

Así pues, es necesario distinguir cuándo un camino representa un número realmente presente en la colección, para distinguirlo de aquellos caminos que se corresponden con prefijos de otros números binarios presentes en la colección.

Una forma de hacerlo es modificar el contenido de los nodos con dígitos, de forma que se almacene no sólo el dígito, sino también un valor booleano que nos indique si ese nodo contiene el último dígito de un número de la colección (por ejemplo con el valor true) o es un prefijo de un número más largo (por ejemplo con el valor false).

De esta manera, ahora el camino del ejemplo sería:

inicio → (1,false) → (1,false) → (0,false) → (1,**true**)

que nos diría que el número 1101 está realmente en la colección. Mientras que el camino:

inicio → (1,false) → (1,false) → (0,**false**)

nos indicaría que el número 110 no está presente en la colección.

b) (1.5 Puntos) Justificar cuáles son las variables que intervienen en el coste de preguntar si un número está o no en la colección. Comparar el coste de esta operación para las dos opciones de implementación y discutir en qué casos debe utilizarse cada una de ellas.

En primer lugar hay que tener en cuenta lo que nos dice el enunciado:

- **No hay un tamaño máximo establecido para el conjunto de números**. Es decir, la cantidad de números binarios almacenada en la colección (a la que llamaremos n) debe ser considerada como una de las variables que van a intervenir en el coste.

- **No hay un tamaño máximo establecido para cada número**. Esto significa que el número de dígitos binarios de uno de los números almacenados en nuestra colección puede ser arbitrariamente grande, por lo que también debe considerarse a la hora de calcular el coste de comprobar si un número está o no presente. Consideraremos, pues, el número de dígitos (llamémoslo b) del número binario más largo almacenado en la colección como otra variable que interviene en el coste.

En la primera implementación, con independencia de si la lista está ordenada por algún criterio, habrá que recorrer todos los elementos de la lista y, para cada uno de ellos, comprobar si el número binario almacenado en esa posición coincide con el buscado. Esta última operación deberá realizarse comparando dígito a dígito ambos números binarios.

Por lo tanto, se deben visitar los n elementos de la lista y, para cada uno de ellos, en el peor de los casos, habrá que comparar un total de b dígitos binarios. Si consideramos que la comparación de dos dígitos binarios se puede realizar en un tiempo constante con respecto a n y a b, entonces el coste de la operación de pertenencia de un número a la colección estaría en $O(n \cdot b)$.

Pensemos ahora en la segunda implementación. ¿Cómo se comprueba si un número binario está presente o no? Nos situamos en el nodo raíz del árbol y comprobamos si éste tiene un hijo que contenga el primer dígito del número a buscar. Pasamos a dicho nodo y comprobamos si tiene un hijo que contenga el segundo dígito del número a buscar… y así sucesivamente hasta encontrar el camino que nos lleve desde la raíz hasta un nodo que nos indique que existe un número en la colección con todos los dígitos del número buscado. Por contra, si no existe un hijo con el siguiente dígito, o bien hemos completado todos los dígitos del número y el el valor booleano del último nodo es Falso, entonces el número no está en la colección.

En el peor de los casos, tendremos que recorrer el camino más largo, es decir, los dígitos del número más largo almacenado en la colección. Es decir, b nodos. Si para cada uno de ellos las operaciones realizadas (comparación de dos dígitos binarios y obtención de uno de los hijos del nodo) son de coste constante con respecto al número de elementos de la colección y a la longitud del número binario, podemos concluir que el coste de comprobar si un número pertenece o no a la colección está en $O(b)$.

Si comparamos el coste de ambas implementaciones, podemos ver que la implementación con un árbol es siempre más eficiente, con independencia de los valores que pudieran tener tanto n como b.

Se podría considerar, **erróneamente**, que la comprobación de que dos números binarios son iguales puede hacerse en un tiempo constante. En este caso, sólo se consideraría el tamaño de la colección como como magnitud a tener en cuenta para el cálculo del coste de la operación de comprobación de pertenencia de un número a la colección.

En el caso de la primera implementación, dicho coste estaría en O(n), puesto que para cada elemento de la lista habría que hacer una operación (erróneamente calculada) de tiempo constante.

¿Y qué pasa con él árbol? Un posible argumento sería considerar que al elegir el camino según la siguiente cifra del número a buscar, estaríamos descartando la mitad de los elementos de la colección, lo que nos llevaría a un coste logarítmico en $O(\log_2 n)$. Sin embargo esto no tiene por qué ser cierto y dependería no sólo de la cantidad de números almacenada, sino también de qué números estuvieran almacenados. Por ejemplo, si todos los números comparten el mismo prefijo, el árbol degenera en una lista (y el coste sería lineal y no logarítmico).

Por ello, es necesario tener en cuenta la longitud máxima de los números almacenados, ya que lo correcto en este caso, sería indicar que el coste dependería linealmente de dicha longitud máxima, es decir, estaría en O(b).

¿Cómo realizar la comparación en este caso? Pues habría que argumentar que, a la vista de estos costes, la implementación de la lista sería preferible siempre que $n < b$, es decir, que la cantidad de números almacenada fuese menor que el tamaño de dichos números. Por el contrario, la implementación del árbol sería preferible cuando $n > b$, es decir, que el tamaño de los números almacenados es menor que la cantidad de ellos presentes en la colección. Obviamente, si n y b son iguales, ambas representaciones serían igualmente costosas.

c) (1 Punto) Supongamos que los números están especificados en otra base (por ejemplo decimal o hexadecimal). ¿Cómo habría que cambiar ambas implementaciones?

En la implementación de la lista, sólo habría que cambiar el contenido de cada elemento, que ya no serían números binarios, sino números en alguna base.

En la implementación del árbol se debería prescindir del árbol binario para utilizar un árbol general, en el que cada nodo tuviera, a lo sumo, un número de hijos igual a la base (10 hijos en base decimal, 16 en hexadecimal…). Esto haría que el árbol fuera más ancho.