P1. **Pregunta sobre la práctica**. Supongamos que se desea añadir una operación **`playRandom`** en **`PlayerIF`** tal que reproduzca una canción al azar de la cola de reproducción. Salvo este detalle, su comportamiento será análogo al de **`play`** (es decir, sacará el identificador de la canción de la cola de reproducción y lo añadirá a las últimas canciones reproducidas). Para ello se apoya en una operación **`int random(int k)`** que devuelve un número entero aleatorio entre 1 y k (siendo k > 0). En base a esto:

a) (0.75 Puntos) Indique, de manera justificada, qué estructura de datos escogería para almacenar la cola de reproducción, teniendo en cuenta que se desea una mayor eficiencia (no asintótica).

En este ejercicio se dejaba abierta la operación sobre la que se pedía eficiencia, para que el estudiante razonase al respecto, eligiese una y justificase su elección.

Las operaciones que pueden realizarse en el TD PlayBackQueue son: constructor (que no se debería ver afectado por la estructura de datos elegida para representar los elementos del TD), añadir identificadores de canciones (que siempre se introducen por el final) y extraer el identificador de la siguiente canción a reproducir. Por lo tanto, debemos decidir si primamos la eficiencia de las operaciones de inserción de identificadores o de la extracción del identificador de la siguiente canción a reproducir.

Por un lado, según el funcionamiento de la cola de reproducción en la práctica, los identificadores de canciones se insertan por el final y se extraen por el principio. Esto nos indica que la estructura más adecuada para implementar la cola de reproducción sería, precisamente, una cola, ya que tanto la inserción de un identificador como la extracción de la siguiente canción a reproducir tendrían un coste constante.

Ahora tenemos una nueva operación: la siguiente canción a reproducir no necesariamente será la situada en la primera posición de la cola. Esto hace que ahora una cola no sea la estructura más adecuada, ya que no permite acceder a cualquier elemento de su secuencia sin destruir la misma (lo que nos obliga a reconstruirla). En cambio, una lista sí que nos permite hacer eso, aunque a costa de perder eficiencia en la operación de inserción, que ahora tendría un coste lineal (ya que debemos recorrer toda la lista para poder localizar la última posición de la misma).

¿Qué podemos hacer para mejorar la eficiencia de ambas operaciones?

Con las implementaciones actuales de las estructuras vistas en la asignatura no es factible mejorar la eficiencia. O bien utilizamos una cola (primando la eficiencia de la inserción sobre la de la extracción) o bien una lista (primando la eficiencia de la extracción sobre la de la inserción).

Una posible solución consistiría en enriquecer el TAD lista permitiendo que la inserción por el final de la misma fuese también de coste constante, lo que resultaría en una estructura híbrida lista-cola. Para ello se tendría que añadir una nueva operación (lo que implica modificar el TAD), cambiar la representación de

los datos y modificar los constructores. Queda como ejercicio propuesto la realización de esta tarea.

b) (0.75 Puntos) Describa (no es necesario implementar) cuál sería el funcionamiento de un método que implementase esta nueva operación. Justifique su respuesta.

Vamos a considerar dos escenarios diferentes: la utilización de una cola y la de una lista.

En primer lugar, con independencia de la estructura de datos que utilicemos, es necesario realizar una llamada a random, pasándole como parámetro el tamaño de la estructura. Esta llamada nos proporcionará un valor k entre 1 y dicho tamaño, que será la posición en la secuencia del identificador que deberemos extraer.

Supongamos ahora que estamos utilizando una cola. Necesitamos acceder al elemento k-ésimo de la secuencia. Para ello tenemos que extraer los (k-1) elementos anteriores y, para preservar el orden, volver a meterlos en la cola. Esto sitúa el elemento k-ésimo en la primera posición, por lo que podemos acceder al mismo y extraerlo para insertarlo en las últimas canciones reproducidas). Sólo nos resta repetir el proceso de extraer e introducir en la cola los siguientes (n-k) elementos, de forma que al finalizar el proceso la cola vuelva a mantener el mismo orden que al principio (salvo por el elemento extraído).

Si estuviéramos utilizando una lista, el acceso y la eliminación del elemento k-ésimo se pueden realizar con las operaciones propias del tipo, sin necesidad de operaciones extra para mantener el orden.

c) (0.5 Puntos) ¿Cuál sería el coste asintótico temporal en el caso peor del método descrito en el apartado anterior? Justifique su respuesta.

En cualquiera de los dos escenarios, en el peor de los casos deberemos recorrer completamente la secuencia de los n elementos (valor que tomaremos como tamaño del problema) almacenados en la estructura para localizar el identificador de la siguiente canción (lo que implica operaciones de coste constante). Esto supone un coste O(n).

Como no sabemos cómo está implementada la operación random, no podemos asumir que su coste sea constante con respecto al tamaño del problema, por lo que deberemos considerar también el coste de dicha operación. Análogamente, dependiendo de cómo se haya implementado RecentlyPlayed, el coste de insertar un elemento en su estructura también puede variar, por lo que no podemos tampoco olvidarnos del coste de esta operación.

Así pues, el coste del algoritmo descrito estaría en O(n) + Coste(Random) + Coste(Insertar en RP). Lo que equivale a:

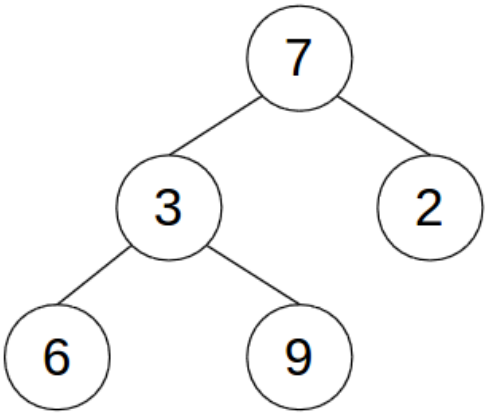
$$O(\max(n,\text{Coste(Random)},\text{Coste(Insertar en RP)})).$$

1. Responda razonadamente las siguientes cuestiones:

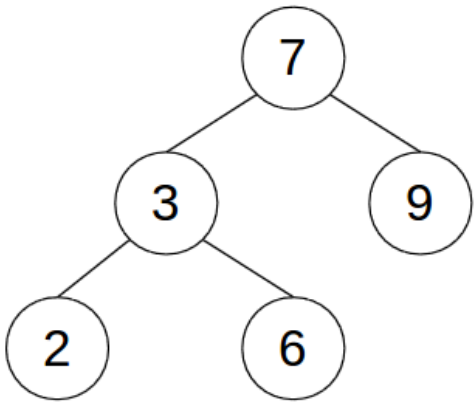
a) (0.75 Puntos) Defina la propiedad que hace que un árbol binario sea, además, un árbol binario de búsqueda. Ilustre su respuesta con dos árboles binarios, uno que cumpla la propiedad (es decir, un árbol binario de búsqueda) y otro que no.

Un árbol binario de búsqueda (o de búsqueda binaria) es un árbol binario en el que el valor almacenado en la raíz es mayor que el valor de cualquier nodo presente en su hijo izquierdo y menor que el valor de cualquier nodo presente en su hijo derecho. Además, tanto el hijo izquierdo como el hijo derecho deben ser, a su vez, árboles binarios de búsqueda.

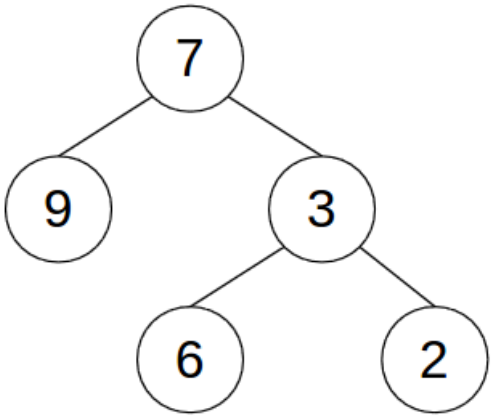
En este primer ejemplo, podemos ver un árbol binario que no es un árbol binario de búsqueda, ya que tenemos tres nodos (el 2, el 6 y el 9) que hacen que no se cumpla la condición:



Por el contrario, el siguiente árbol sí que sería un árbol binario de búsqueda:



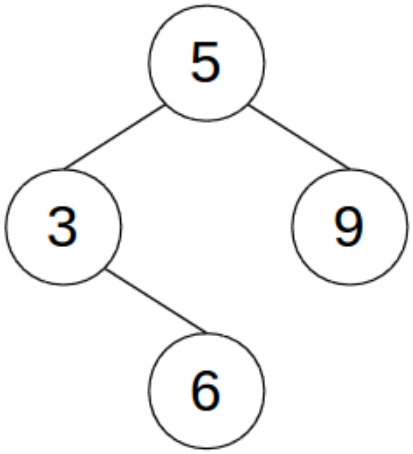
Alternativamente, el árbol puede estar ordenado al revés, siendo la raíz menor que el valor de cualquier nodo presente en su hijo izquierdo y mayor que el valor de cualquier nodo presente en su hijo derecho:



b) (0.75 Puntos) Describa (no es necesario implementar) cómo comprobaría si un árbol binario es, o no, un árbol binario de búsqueda. Justifique su respuesta.

Una forma sencilla de comprobar que un árbol binario cumple con la propiedad anterior es recorrerlo en inorden (es decir: primero el hijo izquierdo, luego el valor de la raíz y finalmente el hijo derecho). Si la secuencia de elementos así obtenida está ordenada (ascendente o descendentemente), entonces el árbol binario será un árbol binario de búsqueda.

Un error común en el examen ha sido utilizar una manera **errónea** de comprobar la propiedad: comprobar que el valor de la raíz es mayor que el valor de la raíz del hijo izquierdo y menor que el valor de la raíz del hijo derecho, repitiendo este proceso de forma recursiva para todos los nodos del árbol. Se puede ver que el siguiente árbol **no es un árbol binario de búsqueda**, ya que existe un nodo en el árbol izquierdo cuyo valor (6) es mayor que el valor de la raíz (5). Sin embargo, la comprobación descrita **arrojaría un resultado positivo**:



c) (0.5 Puntos) ¿Cuál sería el coste asintótico temporal en el caso peor del algoritmo que ha descrito en el apartado anterior? Justifique su respuesta.

El algoritmo descrito consiste en comprobar que la secuencia de elementos obtenida al recorrer el árbol en inorden esté ordenada. Tomemos como tamaño del problema el número de nodos del árbol: n. Esto supone:

- Recorrer el árbol en inorden para obtener la secuencia se realiza recursivamente, con un máximo de dos llamadas recursivas (a=2), cada una de las cuales reduciría el problema a la mitad (b=2). Substituyendo

estos valores en las fórmulas, tenemos que el coste estaría en $O(n^{\log_2(2)})$ es decir, O(n).

- Recorrer la secuencia de n elementos para comprobar que está ordenada. En el caso peor, deberemos recorrer todos los elementos una única vez, para comprobar que mantienen el orden entre sí. Esto significa que realizamos un recorrido de n elementos y hacemos una operación de coste constante con cada uno de ellos: coste en O(n).

Por lo tanto, el coste total se ha de calcular sumando el coste de las dos operaciones (obtener la secuencia y recorrerla), lo que supone calcular el máximo de ellos. Como ambos costes son lineales con respecto al tamaño del problema, tenemos que el coste del algoritmo también sería lineal con respecto al número de elementos almacenados en el árbol: O(n).

2. (2 Puntos) Prográmese en Java un método:

```
void palindromeQueue()
```

dentro del TAD `QueueIF<T>` que transforme una cola de n elementos en una cola de 2n elementos, de tal manera que los n primeros coincidan con los de la cola original (y en el mismo orden) y que, además, sea capicúa. Por ejemplo:

```
q = [1,3,4,25,2,6,3,1,6]
q.palindromeQueue();
q = [1,3,4,25,2,6,3,1,6,6,1,3,6,2,25,4,3,1]
```

El proceso que debemos realizar es el siguiente:

- Recorrer la cola (sin destruirla) para ir metiendo los elementos en una pila.
- Recorrer la pila para añadir los elementos en la cola en el orden inverso en el que fueron introducidos en la pila.

El primer paso puede hacerse de dos maneras: bien utilizando un iterador (método que seguiremos en esta solución) o bien extrayendo e insertando cada elemento hasta volver a dejar la cola en el mismo estado que al principio. Dejamos esta segunda forma como ejercicio.

El código sería el siguiente:

```
void palindromeQueue() {
  IteratorIF<T> it = this.iterator();
  StackIF<T> aux = new Stack();
  while ( it.hasNext() ) {
    aux.push( it.getNext() );
  }
  while ( ! aux.isEmpty() ) {
    this.enqueue(aux.getTop());
    aux.pop();
  }
}
```

3. Una matriz dispersa es una matriz en la que la gran mayoría de elementos son iguales a un determinado valor (al que llamaremos valor trivial) y sólo unos pocos son elementos diferentes a ese valor (a los que denominaremos valores no triviales).

   Por ejemplo, pensemos en una matriz bidimensional de 1000x1000 enteros, de los cuales sólo 50 contienen valores distintos de cero (que será el valor trivial). Si almacenamos esta matriz en un array bidimensional, estaríamos desaprovechando mucho espacio de almacenamiento, ya que sólo el 0’005% de los elementos contienen valores no triviales.

   Responda a las siguientes preguntas (no es necesario programar). Se valorará la eficiencia de la representación escogida (apartado a) en base a las operaciones que se realizarán sobre ella (apartados b, c y d):

   a) (0.75 Puntos) Proponga una representación para matrices bidimensionales dispersas que permita almacenar únicamente los valores no triviales y sus posiciones. Justifique su respuesta.

      La idea es guardar únicamente las posiciones que almacenan valores no triviales, junto con dicho valor. Una primera aproximación puede ser utilizar una lista en la que cada elemento almacene la fila, columna y valor no trivial.

      Sin más, esta representación obligaría a recorrer toda la estructura para buscar una posición determinada, lo cual no es eficiente. Si inducimos un orden en la secuencia, podremos detener la búsqueda antes de completar el recorrido. Por ejemplo, si insertamos ordenadamente los elementos de la siguiente forma:

      - Si dos elementos están situados en distintas filas, deberá aparecer primero en la secuencia el situado en una fila menor.
      - Si dos elementos están situados en la misma fila, deberá aparecer primero en la secuencia el situado en una columna menor.

      podemos detener la búsqueda si nos encontramos con un elemento que debería estar situado con posterioridad al elemento buscado.

      Sin embargo aún podemos hacerlo mejor: con la anterior representación estamos recorriendo todos los elementos de todas las filas anteriores a la fila donde se encuentra elemento buscado. Pero si sabemos que un elemento está, por ejemplo, en la fila 15, no necesitamos visitar todos los elementos de las filas 1 a 14 porque ya sabemos que ahí no estará.

      Aún utilizando listas, una mejora en la representación consistiría en utilizar una lista de filas, cada una de las cuales contendrían el índice de la fila y una lista de pares columna, valor. Así, para encontrar el elemento `(i,j)` primero recorreríamos la lista de filas buscando aquella con el índice `i` para, a continuación, buscar entre los elementos de esa fila el que represente la columna `j`. Si en alguno de estos recorridos nos pasamos del índice buscado, significará que el elemento buscado no existe.

      Cualquiera de estas dos últimas representaciones (utilizando listas en las que los elementos se insertan ordenadamente) serían adecuadas como respuesta a este apartado. Nótese que el hecho de que exista un orden entre los elementos, nos

permite (en el caso medio) abortar la búsqueda antes de recorrer toda la estructura.

Dado que lo relevante es la existencia de un orden entre las casillas, sería posible utilizar un árbol binario de búsqueda. Podríamos, para ello, emplear el orden explicado en la página anterior y utilizar una implementación de un ABB (por ejemplo un AVL) de forma que el coste asintótico de la búsqueda sería logarítmico con respecto al número de elementos almacenados en la estructura.

b) (0.75 Puntos) En base a la representación propuesta en el apartado anterior, describa cómo se accedería al valor de un elemento situado en una posición `(i,j)` de la matriz y calcule el coste asintótico temporal en el caso peor de esta operación.

Abstrayéndonos de la forma concreta de acceso estipulada por la representación escogida, es posible responder a este apartado de igual forma para cualquiera de las estructuras comentadas en el anterior (lista sin ordenación, lista ordenada por coordenadas, lista de filas o árbol binario de búsqueda). Así, el primer paso siempre será buscar si nuestra estructura contiene un elemento en la posición (i,j).

Si el resultado de la búsqueda es positivo, significará que en esa posición se está almacenando un valor no trivial, por lo que lo devolveríamos. Si el resultado fuera negativo, significa que en esa posición la matriz contiene un valor trivial, por lo que se debería devolver dicho valor.

Para calcular el coste asintótico temporal en el caso peor, lo primero que tenemos que establecer es el tamaño del problema: el número de elementos almacenados en la estructura. Llamémoslo n.

En el caso de que la estructura sea una lista, en el caso peor habrá que recorrer toda la secuencia para buscar si almacena un elemento en la posición (i,j). Dado que comprobar si la posición buscada coincide con la del elemento almacenado tiene un coste independiente del tamaño del problema, el coste de la búsqueda estará en O(n).

Si la estructura fuese un ABB, en el caso peor habrá que recorrer el árbol hasta una hoja. Dado que en un ABB (equilibrado en altura como los AVL) la altura del árbol es logarítmica con respecto al número de elementos almacenados en él, el coste de la búsqueda estaría en $O(\log_2(n))$.

Comoquiera que el coste de devolver el valor almacenado (o el valor trivial si no existiese un valor en la posición buscada) es constante con respecto al tamaño del problema, el coste total será el coste de la búsqueda del elemento.

c) (1 Punto) En base a la representación propuesta en el apartado (a), describa cómo se realizaría la modificación de un elemento situado en una posición (i,j) de la matriz.

Nuevamente, al igual que en el apartado anterior, el proceso es independiente de la estructura escogida y comienza con la búsqueda de un elemento almacenado en la estructura con la posición (i,j) buscada.

Una vez realizada la búsqueda, es necesario distinguir varios casos: el proceso que sigue diferirá según el resultado de la búsqueda y según el valor que queramos introducir en dicha posición. Un error muy común en el examen ha sido omitir este análisis por casos:

- Existe un elemento en la estructura en esa posición:
  - Si el valor a introducir es el trivial, hay que eliminar el elemento que ya existía en la estructura.
  - Si el valor a introducir es no trivial, simplemente hay que modificar el valor almacenado en su posición.
- No existe un elemento en la estructura en esa posición:
  - Si el valor a introducir es el trivial, no hay que hacer nada.
  - Si el valor a introducir es no trivial, entonces hay que crear un nuevo elemento en la estructura, para esa posición y ese valor, e introducirlo en su lugar correspondiente.

d) (1.5 Puntos) En base a la representación propuesta en el apartado (a), describa cómo se realizaría, de manera eficiente, la suma de dos matrices dispersas (del mismo tamaño) así representadas. Justifique la eficiencia de su solución calculando el coste asintótico temporal en el caso peor del algoritmo descrito.

Lo fundamental para este apartado es que los elementos almacenados en la estructura guarden un orden entre sí. Gracias a esto, podremos recorrer simultáneamente las estructuras de las dos matrices dispersas, algo que un iterador nos permite realizar de una forma homogénea independientemente de la estructura escogida.

Una vez establecida la forma de recorrer ambas estructuras, podemos encontrarnos con los siguientes casos:

- Sólo hay elemento en la posición `(i,j)` en una de las dos matrices. Esto significa que se añade ese mismo elemento en la matriz resultado.
- En ambas matrices existe un elemento en la posición `(i,j)`.
  - Si la suma resultase ser el valor trivial, no hay que añadir un elemento en la posición `(i,j)` en la matriz resultado.
  - Si la suma no es el valor trivial, entonces se añade un elemento en la posición `(i,j)` con la suma como valor en la matriz resultado.

Calculemos ahora el coste asintótico temporal en el caso peor de esta operación. Lo primero es, como siempre, establecer el tamaño del problema. En este caso tenemos que recorrer todos los elementos de las dos estructuras, llamémoslos n_1 y n_2.

En segundo lugar, hay que hacer notar que las operaciones para realizar el análisis por casos anterior tienen un coste independiente del tamaño del problema.

Por último, no olvidemos que se deberá insertar en la estructura de la matriz resultado todos los elementos que representen la matriz suma. El coste de esta operación sí que depende del tamaño del problema y, además, de la estructura elegida.

En el peor de los casos, todos los elementos de ambas matrices estarán en posiciones diferentes, es decir, no habrá ninguna posición `(i,j)` con un elemento referido a ella en ambas matrices a la vez.

Esto significa que, como mucho, habrá que insertar $(n_1 + n_2)$ elementos en una estructura, tras extraerlos del recorrido conjunto de las dos estructuras de partida (lo que supone, también, un total de $(n_1 + n_2)$ elementos analizados). Para cada elemento analizado se deberá insertar en la estructura, por lo tanto, el coste será el siguiente:

- Con una estructura de lista ordenada, cada uno de los $(n_1 + n_2)$ elementos recorridos se inserta en una lista ordenada. Como la inserción en una lista es lineal con respecto al tamaño de la misma y éste será (como máximo) $(n_1 + n_2)$, entonces tenemos que el coste estará en:

$$O\ (\ (n_1 + n_2)^2)$$

- Con una estructura de lista de filas, cada uno de los $(n_1 + n_2)$ elementos recorridos se inserta en una lista ordenada de pares columna-valor que está, a su vez, dentro de otra lista ordenada de filas. En el peor de los casos, la lista de filas contendrá un elemento por cada fila de la matriz (sea `f` el número total de filas) y la lista de pares columna-valor uno por cada columna (sea `c` el número total de columnas). En este caso, habrá que recorrer, como máximo, un total de `f` elementos en la lista de filas y un total de c elementos en la lista de pares columna-valor. Esto supone un total de `(f + c)` elementos recorridos (como máximo) por cada inserción. Así pues, el coste esté en:

$$O\ (\ (n_1 + n_2) * (f + c)\ )$$

- Con una estructura de árbol binario de búsqueda, cada uno de los $(n_1 + n_2)$ elementos recorridos se inserta en un árbol binario de búsqueda. La inserción en dicha estructura es proporcional al logaritmo en base 2 del número total de elementos almacenados, por lo que el coste estará en:

$$O\ (\ (n_1 + n_2) * \log_2(n_1 + n_2)\ )$$

Para finalizar, retomamos la idea que expusimos en la solución a la pregunta de la práctica: utilizar una estructura de lista modificada de forma que la inserción de elementos por el final se pueda realizar en tiempo constante. En este caso, la creación de la matriz resultado utilizando una estructura de lista sería mucho más rápida, ya que las inserciones se realizarían en un tiempo constante.