PRÁCTICA DE ESTRATEGIAS DE PROGRAMACIÓN Y ESTRUCTURAS DE DATOS

CURSO 2023/2024

ALEXANDRE INSUA MOREIRA
alexandreinsua@gmail.com 616 381 448 36.163. 615-W

Representación de los tipos abstractos de datos escogidos

Player

Siguiendo con el enunciado propuesto, se ha concebido la clase **Player** como una que implementa la interfaz **PlayerIF** e incorpora cuatro tipos mediante composición: **TuneCollection**, para el repositorio de canciones, **PlayListManager** para el gestor de listas de reproducción, **PlayBackQueue** para la cola de canciones a la espera de ser reproducidas y **RecentlyPlayed** para almacenar las últimas canciones reproducidas.

Se podría haber habilitado una estructura de datos, como un array, para almacenar en ella estos cuatro tipos, pero que no se obtiene ningún beneficio de ello.

Query

El tipo Query representa los criterios para realizar una búsqueda en el repositorio. Implementa la interfaz QueryIF.

Tune

Representa una canción. Implementa la interfaz TuneIF.

PlayList

El tipo **PlayList** implementa la interface **PlayListIF**. Para poder almacenar los identificadores de las canciones se ha optado por implementar una lista porque permite realizar las operaciones del enunciado de manera más óptima.

Dado que el enunciado exige que se pueda eliminar todas las ocurrencias de un identificador, la opción de utilizar un tipo **Queue** o **Stack** implicaría complicar este algoritmo. En ambos casos como las operaciones de desencolar o desapilar son destructivas, habría que ir reconstruyendo el TAD sin el identificiador que se debe eliminar. En cambio, basta con recorrer la lista e ir eliminando aquellas ocurrencias.

PlayListManager

El tipo **PlayListManager** implementa la interface **PlayListManagerIF** y se encarga de gestionar las listas de reproducción que se puede definir en el reproductor. Para establecer la relación entre el identificador y la lista de reproducción se ha creado una clase interna. Como el tipo debe implementar un método de eliminación de una lista existente, se ha optado por que implemente un tipo List para poder disponer del método remove.

Una alternativa sería implementar un tipo Set, que también dispondría y de un método para eliminar elementos y que, además, garantizaría que los elementos sean únicos.

PlayBackQueue

Este TAD representa una cola de reproducción de canciones que van a ser reproducidas y que pueden aparecer repetidas. Implementa la interface PlayBackQueueIF. Cada lista almacena un número indeterminado de canciones, que pueden ser repetidas. La primera canción que se reproduce es aquella que se ha introducido inicialmente, por lo que su funcionamiento se corresponde con una estructura de datos de tipo FIFO. **Para implementarla se ha establecido un campo de clase de tipo Queue,** que implementa la interface QueueIF que, a su vez, mantiene esta política de acceso y que mantiene sendos métodos para encolar, obtener y desencolar elementos adecuadamente.

Se han excluido una implementación con un tipo Set porque no permite la repetición de elementos y con un tipo Stack porque mantiene una política de acceso diferente. En el caso del tipo Stack se podría implementar una solución que pasaría por la destrucción de la estructura original. Sin embargo, **se puede usar un tipo List para esta implementación** usando sus métodos de inserción, obtención y eliminación de elementos para reproducir en funcionamiento FIFO.

RecentlyPlayed

Representa las últimas canciones reproducidas en el reproductor e implementa la interface **RecentlyPlayedIF**. Teniendo en cuenta que su tamaño está acotado, he determinado que cuando se alcance esa cota, se debe eliminar la canción más antigua y agregar la más reciente. Por ello se ha implementado un tipo **Queue** para gestionarla. Una alternativa muy viable es usar una lista para tal efecto.

Estudio teórico del coste de los métodos de TAD PlayerIF

addListOfTunesToPlayBackQueue(ListIF<Integer> lT)

Añade una lista de identificadores de canciones del repositorio a la cola de reproducción. Invoca el método *addTunes* de la clase **PlayBackQueue**. Este método realiza un recorrido de la lista que recibe como parámetro y encola cada elemento mediante el método *enqueue* de la clase **Queue**. Esta operación de encolado es lineal.

Tamaño del problema: **n** (tamaño de la lista de identificadores playList).

Coste asindótíco: **O(n)** (recorrido de la lista) + **O(1)** (encolado de un entero) = **O(n)**.

addListOfTunesToPlayList(String playListID, ListIF<Integer> lT)

Añade una lista de identificadores de canciones del repositorio a una lista de reproducción. Para ello invoca el método *getPlayList* de la clase **PlayListManager** que localiza la playlist en la colección mantenida en la clase **PlayListManager** mediante un iterador y un bucle *while*. Se trata de una búsqueda lineal cuyo peor caso será que el identificador sea el último de la colección.

Después realizar el encolado de cada elemento de la lista de enteros que recibe como parámetro en la playlist obtenida. El tamaño de esta operación corresponde con el tamaño de la lista *lT* y la operación de inserción cada uno de estos elementos es también una operación lineal ya que la iteración se realiza con un iterador y un bucle while y el método *insert* de la clase **List** tiene coste lineal. Por tanto, el coste global del método es lineal.

Tamaño del problema: **n** (tamaño de la colección de playList en PlayListManager) + **m** (tamaño de la lista de enteros que se recibe como parámetro).

Coste asindótíco: **O(n)** (búsqueda de la playlist) + **O(m)** (inserción de cada entero) = **O(n)**.

addPlayListToPlayBackQueue(String playListID)

Añade el contenido de una lista de reproducción a la cola de reproducción. Este método realizar tres operaciones: localiza una lista de reproducción, obtiene su contenido y lo añade a lista cola de canciones que van a ser reproducidas. La primera operación realiza una búsqueda de una lista de reproducción en la clase **PlayListManager** implementando un bucle *while* junto con un iterador, por lo que su

coste es lineal. El *getter* del contenido de la lista de reproducción realiza un retorno de un campo de clase, por lo que su coste es constante. La adición de este contenido se realizar recorriendo el contenido de la lista (cuya longitud es el tamaño de esta parte) y encolando cada uno de ellos.

Tamaño del problema: **n** (tamaño de la colección de playList en PlayListManager) + **m** (tamaño del contenido de la playList).

Coste asindótíco: **O(n)** (búsqueda de la playlist) + **O(1)** (obtención del contenido de la playlist) + **O(m)** (inserción del contenido en la cola de reproducción) = **O(n)**.

addSearchToPlayBackQueue(String playListID, String title, String author, String genre, String album, int min_year, int max_year, int min_duration, int max_duration)

Añade los identificadores de todas las canciones del repositorio que cumplan los criterios indicados a la cola de reproducción. Este método crea una instancia de la clase *List* y otra de la clase *Query* para realizar la búsqueda. Realiza una búsqueda en el repositorio de canciones mediante una iteración con un bucle *for* por que la clase la clase **TuneCollection** no implementa un iterador. Para cada canción se ejecuta el método match cuyo coste es lineal y, en caso de que satisfaga los criterios de búsqueda, se inserta al final de una lista habilitado como almacén temporal. La inserción de estas canciones se realiza mediante el método *insert* y la propiedad *size* de la clase List, polo lo que el tamaño será en número de canciones que coinciden y el coste es lineal.

La segunda parte del método encola los elementos coincidentes en la cola de reproducción. En este caso, el tamaño de problema coincide con el tamaño de la lista de elementos coincidentes porque se itera por ella usando un iterador y encolado de cada elemento.

Tamaño del problema: **n** (tamaño del repositorio) + **m** (número de canciones coincidentes)

Coste asindótíco: **O(1)** (instanciaciación de List) + **O(1)** (instanciaciación de Query) + **O(n)** (búsqueda en el repositorio + **O(1)** (ejecución del método *match*) + **O(n)** (encolado de elementos coincidentes = **O(n)**.

addSearchToPlayList(String playListID, String title, String author, String genre, String album, int min_year, int max_year, int min_duration, int max_duration)

Añade los identificadores de todas las canciones del repositorio que cumplan los criterios indicados a una lista de reproducción. La operación de búsqueda de las canciones coincidentes se ha comentado en el método anterior. La siguiente operación se delega en el método *getPlayList* de la clase **PlayListManager** que busca la playlist en la colección de playlist cuyo tamaño es el tamaño del problema y coste es lineal como se ha comentado anteriormente. El paso final es insertar cada elemento de la lista auxiliar en la playlist obtenida. Esta operación ya se ha comentado anteriormente, el tamaño será el tamaño de la lista auxiliar y su coste lineal.

Tamaño del problema: **n** (tamaño del repositorio) + **m** (número de canciones coincidentes) + **o** (tamaño de la colección de playlist)

Coste asindótíco: **O(1)** (instanciaciación de List) + **O(1)** (instanciaciación de Query) + **O(n)** (búsqueda en el repositorio + **O(1)** (ejecución del método *match*) + **O(n**) (inserción de la lista coincidente en la playlist).

ClearPlayBackQueue()

Vacía la cola de reproducción. Para ello delega en el método *clear* de la clase **PlayBackQueue**, que a su vez invoca el método *clear* de la clase **Queue**. Este realizar una asignación de null a su atributo *lastNode* e invoca el método de *clear* de la clase **Sequence**, que asigna *null* a su atributo *firstNode* e invoca el método *clear* de la clase **Collection**, que, finalmente, asigna un valor 0 a su atributo *size*. Por lo tanto, se trata de un conjunto de asignaciones que tienen un conste lineal.

Tamaño del problema: no aplica porque las tres asignaciones son constantes.

Coste asindótíco: **O(1)**

createPlayList(String playListID)

Crea una nueva lista de reproducción a partir de su identificador. Delega en el método *createPlayLIst* de la clase **PlayListManager**, que inserta una nueva playlist al final de la colección de *playLists* implementada en aquella clase. El coste total es linal.

Tamaño del problema: 1 (la lista de reproducción que se va crear).

Coste asindótíco: **O(1)** (insertar la nueva lista de reproducción) + **O(1)** (instanciación de la nueva playList) = **O(1)**.

GetPlayBackQueue()

Devuelve los identificadores de las canciones contenidas en la cola de reproducción. Delega en el método *getContent* de la clase **PlayBackQueue**. Este método crea una lista auxiliar y recorre la cola mediante un iterador, inserta cada elemento en la lista auxiliar y retorna esa lista auxiliar. El tamaño del problema se corresponde con el tamaño de la cola de reproducción.

Tamaño del problema: **n** (tamaño de la cola de reproducción).

Coste asindótíco: **O(1)** (instanciación de la lista auxiliar) + **O(n)** (recorrido de la lista) + **O(1)** (inserción de cada elemento) = **O(n)**.

getPlayListContent(String playListID)

Devuelve el contenido de una lista de reproducción. Para ello primero busca la *playList* en la colección mantenida en la clase **PlayListManager** a través del método *getPlayList*, que ya se ha comentado. Después asigna una instancia de la *playList* para comprobar su valor. Finalmente devuelve el contenido de la playList usando un *getter*.

Tamaño del problema: **n** (tamaño de la colección de playLists)

Coste asindótíco: **O(n)** (búsqueda de la playList en la colección) + **O(1)** (asignación de la playList) + **O(1)** (uso getter y retorno del contenido) = **O(n)**.

GetPlayListIDs()

Devuelve los identificadores de todas las listas de reproducción existentes. Para ello crea una lista auxiliar y e invoca el método *getIDs* de la clase **PlayListManager**. Este método realizar un recorrido con un iterador sobre la colección mantenida en la clase para obtener los ids. El tamaño de problema corresponde con el tamaño de la colección.

Tamaño del problema: **n** (tamaño de la colección en PlayListManager).

Coste asindótíco: **O(1)** (creación de la lista auxiliar) + **O(n)** (recorrido de la colección + **O(1)** inserción de los ids en la lista auxilar + **O(1)** (comprobación de contenido de la lista auxiliar) = **O(n)**.

GetRecentlyPlayed()

Devuelve los identificadores de las últimas canciones reproducidas que están almacenadas en **RecentlyPlayed**. Para ello delega en el método *getContent* de la clase **RecentlyPlayed**. Este método crea una lista auxiliar e itera con un iterador la cola de ids de canciones reproducidas recientemente. Sin embargo, el tamaño de esta cola es fijo, en función de un parámetro introducido al inicio de la aplicación. El coste del método es constante.

Tamaño del problema: no aplicar porque la longitud de la cola depende de un parámetro global al arrancar la aplicación, por lo que es constante.

Coste asindótíco: **O(1)**.

play()

Reproduce la siguiente canción en la cola de reproducción. Primero comprueba si la cola de reproducción no está vacía. En caso de que no este vacía, obtiene de la cola de reproducción el primer elemento mediante el método *getFirstTune* e **PlayBackQueue**. Como esa clase implementa una **Queue**, ese método usa *getFirst* que está obteniendo el valor de *firstNode*, por lo que su coste en constante. Después añade la canción obtenida a la instancia de la clase **RecentlyPlayed** mediante el método *addTune*. Como esta clase también implementa una **Queue** para mantener su colección está usando el método *enqueue* cuyo coste es constante. También se comprueba si el número de canciones reproducidas ha llegado a su máximo, en cuyo caso se realiza un desencolado. En ambos casos el coste es también constante. Finalmente, se extrae de la cola de reproducción el primer elemento de la cola de reproducción. El método *extractFirstTune* de la clase **PlayBackQueue** también invoca un metodo *dequeue* de la **Queue** que implementa, por lo que su coste también es constante. Por lo tanto, es coste el coste total del método *play* es constante.

Tamaño del problema: 1 (la canción que se va a reproducir).

Coste asindótíco: **O(1)** (obtener el primer elemento de la cola de reproducción) + **O(1)** (agregar esa canción a la cola de reproducidas recientemente + **O(1)** (desencolar esa canción de la cola de reproducción) = **O(1)**.

removeTuneFromPlayList(String playListID, int tuneID)

Elimina una canción de una lista de reproducción. Para ello realiza dos operaciones. Primero busca la lista de reproducción en la colección mantenida en el **PlayListManager** con un iterator por lo que el tamaño del problema depende del tamaño de la colección su conste es lineal. Después crea una instancia de la playList.

A continuación, invoca el método *removeTune* de la clase **PlayList**. Este realizar una búsqueda en su contenido usando un iterador y cuando encuentra una coincidencia, invoca el método *remove* de la clase *List*. Este método, en caso de que el índice sea diferente de 1, utiliza el método *getNode* de la clase Secuencia, que a su vez realiza una iteración por sus todos con un bucle *for*. **Esto implica que para eliminar una canción primero se itera para localizarla y después se vuelve a iterar para eliminarla, por lo que el tamaño del problema será el cuadrado del tamaño del contenido** de la lista de reproducción y su coste será cuadrático, con lo que el coste global también será cuadrático.

Tamaño del problema: **n** (tamaño de la colección de listas de reproducción) + **m** (contenido de la lista de reproducción) * **m** (contenido de la lista de reproducción).

Coste asindótíco: **O(n)** (búsqueda de la lista de reproducción + O(1) (asignación de la lista de reproducción) + **O(n^2)** (borrado de la canción) = **O(n^2)**.