Documentación de la práctica de EPED

Pablo Cumpián Díaz

Abril, 2017

Representación de los TAD

Para cada uno de los TAD que deben ser implementados, justificar la elección de la Estructura de Datos escogida para representar los elementos del TAD y comentar, al menos, otra posible Estructura (de las estudiadas en la asignatura) que pudiera usarse indicando los pros y los contras de dicha elección.

Para la clase PlayBackQueue he usado una cola como estructura de datos, de forma que las canciones que se fueran añadiendo simplemente se iban encolando al final de la lista con un coste lineal (la Cola guarda siempre el último nodo al que se añaden los nuevos). De la misma forma cada vez que Player ejecuta el método Play simplemente extrae el primer elemento (al que corresponde reproducirse), que también tiene un coste lineal.

Para RecentlyPlayed he utilizado también una cola, ya que no he encontrado otra opción que me permitiese invertir la propia estructura sin tener que usar otra estructura auxiliar.

En cuanto a PlayBackQueue no encuentro una alternativa a la cola (de hecho es intuitivamente la estructura más conveniente). En lo que respecta a RecentlyPlayed al principio de implementar la práctica plantee utilizar una pila que fuera apilando las canciones según se iban reproduciendo. El problema vino cuando hubo que realizar la inversión de la estructura: para ello es necesario utilizar otra pila en la cual ir volcando los elementos de la primera para devolver las canciones en orden inverso. Además de la penalización en el coste computacional, esto supondría utilizar una estructura auxiliar, lo que incurriría en un costo adicional de memoria.

Estudio teórico del coste

- El tamaño de todos los métodos C.R.U.D., salvo borrar toda la cola de reproducción, el método play() y las llamadas a getters, es N en el peor de los casos, ya que con una iteración debería ser suficiente para crear, consultar, modificar o eliminar cualquier nodo. Ahora bien, la implementación con abstracciones de clases puede provocar alteraciones en alguno de estos métodos.

- Para limpiar la cola de reproducción y reproducir una canción siempre se hacen operaciones de coste constante O(1). Limpiar la cola de reproducción es simplemente declarar el primer nodo como null. De igual forma el método play() realiza operaciones de coste constante (encolar una nueva canción en RecentlyPlayed y extraer la primera canción de la cola de reproducción).

public ListIF<String> getPlayListIDs();

El coste de este método es constante (O(1)) ya que funciona como un get de la lista de identificadores de PlayListManager.

```
public ListIF<String> getPlayListIDs() {
        return this.playListManager.getIDs();          O(1)
}
```

public ListIF<Integer> getPlayListContent(String playListID);

El coste peor es O(n) ya que necesita de una iteración para buscar la lista de reproducción en PlayListManager.

```
public ListIF<Integer> getPlayListContent(String playListID) {
   if(this.playListManager.contains(playListID)){         O(n)
     return(ListIF<Integer>)this.playListManager.getPlayList(playListID).getPlayL
     ist();                                               O(n)
   } else {
      return new List<Integer>();                         O(1)
   }
}
```

public ListIF<Integer> getPlayBackQueue();

public ListIF<Integer> getRecentlyPlayed();

Ambos métodos, a pesar de funcionar como getters, al devolver una lista, necesita hacer la conversión haciendo una iteración para rellenar la lista a devolver con cada elemento de la estructura de datos.

```
  public ListIF<Integer> getPlayBackQueue() {
            return this.playBackQueue.getContent();        O(n)
  }

public ListIF<Integer> getRecentlyPlayed() {
            return this.recentlyPlayed.getContent();       O(n)
}
```

public void createPlayList(String playListID);

Este método tiene un coste de O(n), ya que tiene que recorrer toda la lista hasta llegar al último lugar e insertar una nueva PlayList.

```
public void createPlayList(String playListID) {
        if(!(this.playListManager.contains(playListID))){          O(n)
            this.playListManager.createPlayList(playListID);      O(n)
        }
}
```

public void removePlayList(String playListID);

Para eliminar una lista primero ha de hacerse una búsqueda por iteración hasta encontrarla y después borrarla, con lo que el problema en sí son dos costes lineales suvesivos. Si se eliminan todas las ocurrencias de la lista a borrar el coste pesimista y el real en cada caso será el mismo (el número de elementos a procesar): O(n).

```
public void removePlayList(String playListID) {
    if(this.playListManager.contains(playListID)){            O(n)
        this.playListManager.removePlayList(playListID);      O(n)
    }
}
```

public void addListOfTunesToPlayList(String playListID, ListIF<Integer> lT);

El coste de este método es O(n) ya que han de hacerse dos iteraciones sucesivas (no anidadas): para buscar la lista y para insertar los nuevos elementos.

```
public void addListOfTunesToPlayList(String playListID, ListIF<Integer> lT) {
   if(this.playListManager.contains(playListID)){                        O(n)
      this.playListManager.getPlayList(playListID).addListOfTunes(lT); O(n)+O(n)
   }
}
```

public void addSearchToPlayList(String playListID, String t, String a, String g, String al, int min_y, int max_y, int min_d, int max_d);

Debemos buscar la lista a partir del ID e iterar por toda la colección de TuneCollection para encontrar los matches y posteriormente añadir las canciones a la playList, con lo que el coste de estas tres iteraciones es O(3n).

```
public void addSearchToPlayBackQueue(String t, String a, String g, String al,
  int min_y, int max_y, int min_d, int max_d) {

    Query query = new Query(t,a,g,al,min_y,max_y, min_d, max_d);      O(1)

    List<Integer> lista = new List<Integer>();                        O(1)

    int contador = 0;                                                 O(1)


    for(int i = 0; i < collection.size()-1; i++){        O(n)

        if(collection.getTune(i).match(query)){          O(1)

            lista.insert(i, ++contador);                 O(1)

        }

    }

    this.playBackQueue.addTunes(lista);                  O(n)

}
```

public void removeTuneFromPlayList(String playListID, int tuneID);

Dos iteraciones: encontrar la playList, y dentro de ella buscar la canción a borrar: coste O(n).

```
public void removeTuneFromPlayList(String playListID, int tuneID) {
 if(this.playListManager.contains(playListID)){
   this.playListManager.getPlayList(playListID).removeTune(tuneID); O(n) + O(n)
 }
}
```

public void addListOfTunesToPlayBackQueue(ListIF<Integer> lT);

Una única iteración para añadir las canciones a la lista de reproducción: O(n).

```
public void addListOfTunesToPlayBackQueue(ListIF<Integer> lT) {
    this.playBackQueue.addTunes((List<Integer>) lT);                O(n)
}
```

public void addSearchToPlayBackQueue(String t, String a, String g, String al, int min_y, int max_y, int min_d, int max_d);

Una única iteración para añadir las canciones a la lista de reproducción: O(n).

```
public void addSearchToPlayBackQueue(String t, String a, String g, String al,
 int min_y, int max_y, int min_d, int max_d) {

  Query query = new Query(t,a,g,al,min_y,max_y, min_d, max_d);      O(1)

  List<Integer> lista = new List<Integer>();                       O(1)

  int contador = 0;                                                O(1)


  for(int i = 0; i < collection.size()-1; i++){                    O(n)

      if(collection.getTune(i).match(query)){                      O(1)

          lista.insert(i, ++contador);                             O(1)

      }

  }

  this.playBackQueue.addTunes(lista);                              O(n)

}
```

public void addPlayListToPlayBackQueue(String playListID);

O(n): una iteración para buscar la playList y después otra para encolarla a la lista de reproducción. Al no estar anidadas el coste sigue siendo lineal.

```
public void addSearchToPlayBackQueue(String t, String a, String g, String al,
  int min_y, int max_y, int min_d, int max_d) {

    Query query = new Query(t,a,g,al,min_y,max_y, min_d, max_d);      O(1)

    List<Integer> lista = new List<Integer>();                        O(1)

    int contador = 0;                                                 O(1)


    for(int i = 0; i < collection.size()-1; i++){                     O(n)

        if(collection.getTune(i).match(query)){                       O(1)

            lista.insert(i, ++contador);                              O(1)

        }

    }

    this.playBackQueue.addTunes(lista);                               O(n)

}
```

public void clearPlayBackQueue();

Este método tiene coste O(1), dado que el coste siempre es el mismo: el de asignar al primer nodo el valor null.

```
public void clearPlayBackQueue() {

    this.playBackQueue.clear();                          O(1)

}
```

public void play();

Realiza operaciones de coste constante (encolar una nueva canción en RecentlyPlayed y extraer la primera canción de la cola de reproducción). O(1).

```
public void play() {

    if(!this.playBackQueue.isEmpty()){

       recentlyPlayed.addTune(this.playBackQueue.getFirstTune());     O(1)+O(1)

       this.playBackQueue.extractFirstTune();                         O(1)

    } else {    //caso de que la pila esté vacía

       return;                                                        O(1)

    }

}
```

Supuestos adicionales

a) Supongamos que se añade una operación en PlayListIF que permita insertar una lista de identificadores de canciones en una posición determinada de la lista de reproducción. ¿Es necesario modificar la elección de la Estructura de Datos escogida para representar los elementos de este TAD?

No, únicamente habrá que llamar iterativamente al método insert del objeto de clase List para que inserte cada elemento en una posición que parte del inicio y va aumentando con cada iteración.

b) Supongamos que se añade una operación en PlayBackQueueIF que permita eliminar todas las apariciones de una canción (representada mediante su identificador) de la cola de reproducción. ¿Sería aconsejable cambiar la Estructura de Datos empleada para representar los elementos de este TAD?

No, la cola permite recorrer igualmente todos los nodos, únicamente el coste sería de O(n) al tener que recorrer todos los elementos. En su implementación actual, solamente tiene que borrar el primer elemento que encuentre, con lo que el coste máximo (peor caso) es de O(n) si tiene que llegar hasta el último nodo.