Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-2692 - Laboratorio de Algoritmos y Estructuras de Datos II
Trimestre Enero-Marzo 2020

Administrador de Música

El objetivo del proyecto es la implementación de una aplicación en Python para administrar y reproducir archivos de audio. Los archivos de audio con que va a trabajar esta aplicación son canciones. Se requerirá que implemente cuatro módulos, tres de ellos consisten en tipos abstractos de datos (TADs) y uno integra los TADs para hacer una aplicación que sirva para administrar y reproducir música.

1. Tipos de datos

1.1. TAD Canción

El objetivo de este TAD es el representar la información relacionada con un archivo de audio que se asume contiene una canción. Para cada canción se debe tener su *título*, su *intérprete*, y la *ubicación* del archivo de audio con la música. La *ubicación* debe corresponder a un dirección absoluta desde la raíz del sistema de archivos hasta el archivo de audio. Teniendo como restricción que el sistema de archivo es el que usa los sistemas de operación basados en Unix. Por ejemplo, si un archivo de música se llama "Beethoven.9na-Sinfonia.mp3", su título podría ser "9na Sinfonia de Beethoven", el intérprete la "Filarmónica de Berlín" y una ubicación válida es "/home/gpalma/MiMusica/Beethoven.9na-Sinfonia.mp3".

Se define la función `esUbicacionValida`, como la función que recibe como argumento como String u y retorna `True` si u corresponde a la dirección absoluta a un archivo de audio que existe en el sistema de archivos, y el archivo de audio posee la extensión ".mp3" o ".wav". Retorna `False` en caso contrario.

La Figura 1 presenta la especificación del TAD Canción.

1.2. TAD Reproductor

Este TAD es el encargado de proporcionar las facilidades para la reproducción de canciones. La idea es hacer transparente a un módulo cliente o usuario, las operaciones de un reproductor de música. Para poder lograr este objetivo el módulo debe hacer uso de alguna librería externa que permita la reproducción de archivos de sonidos, en específico, los archivos de sonido con extensiones "mp3" y "wav", los cuales son los tipos de archivo que maneja los objetos de tipo Canción. Los detalles de la librería de sonido a utilizar en la implementación de este TAD se indicarán posteriormente. El reproductor debe tocar una canción, debido a esto en el modelo de representación el reproductor contiene a un objeto de tipo `Canción` que va a ser la canción cargada en reproductor para su reproducción. Cuando se crea un reproductor se debe realizar al menos dos operaciones. Primero se debe inicializar la librería de audio a utilizar y segundo se debe cargar una canción para que este lista para su reproducción. Las operaciones de este TAD son las comunes de un reproductor de música. El método `cargarCancion` permite a un reproductor cargar una nueva canción a reproducir. El método `estaTocandoCancion` detecta si en el momento actual la librería de audio está reproduciendo un archivo de sonido o no. Para determinar esto se debe hacer uso de la facilidades de la librería de audio. La Figura 2 muestra el TAD.

Especificación del TAD Canción

Modelo de Representación

const *titulo* : String .

const *interprete* : String .

const *ubicacion* : String .

Invariante de Representación

$titulo \neq \text{NULL} \ \wedge \ interprete \neq \text{NULL} \ \wedge \ ubicacion \neq \text{NULL} \ \wedge \ esUbicacionValida(ubicacion)$

Operaciones

constructor *crearCancion* (**in** t : String; **in** i : String; **in** u : String) → Canción

{ **Pre:** $t \neq \text{NULL} \wedge i \neq \text{NULL} \wedge u \neq \text{NULL} \wedge esUbicacionValida(u)$ }

{ **Post:** $self.titulo = t \ \wedge \ self.interprete = i \ \wedge \ self.ubicacion = u$ }

fmeth *obtenerTitulo* ( ) → String

{ **Pre:** True }

{ **Post:** $obtenerTitulo = self.titulo$ }

fmeth *obtenerInterprete* ( ) → String

{ **Pre:** True }

{ **Post:** $obtenerInterprete = self.interprete$ }

fmeth *obtenerUbicacion* ( ) → String

{ **Pre:** True }

{ **Post:** $obtenerUbicacion = self.ubicacion$ }

fmeth *aString* ( ) → String

{ **Pre:** True }

{ **Post:** $aString =$ String que muestra el $self.titulo$ y el $self.interprete$ de la canción }

Fin TAD

Figura 1: Especificación del TAD Canción

Especificación del TAD Reproductor

Modelo de Representación

var *actual* : Canción .

Invariante de Representación

$actual \neq$ NULL

Operaciones

constructor *crearReproductor* (**in** c : Canción) → Reproductor
{ **Pre:** True }
{ **Post:** $self.actual = c \wedge$ se inicializa la librería de reproducción de audio $\wedge$ se carga en la librería de audio el archivo indicado en $c.ubicacion$ }

meth *cargarCancion* (**in** c : Canción)
{ **Pre:** True }
{ **Post:** $self.actual = c \wedge$ Se carga en la librería de audio el archivo $c.ubicacion$ }

meth *reproducir* ( )
{ **Pre:** True }
{ **Post:** Se reproduce el archivo de audio $self.actual.ubicacion$ }

meth *parar* ( )
{ **Pre:** True }
{ **Post:** Se detiene la reproducción del archivo de audio $self.actual.ubicacion$ }

meth *pausa* ( )
{ **Pre:** True }
{ **Post:** Se pausa la reproducción del archivo de audio $self.actual.ubicacion$ }

fmeth *estaTocandoCancion* ( ) → Boolean
{ **Pre:** True }
{ **Post:** *estaTocandoCancion* ≡ **Si** el archivo de audio $self.actual.ubicacion$ se está reproduciendo **entonces** True **de lo contrario** False }

Fin TAD

Figura 2: Especificación del TAD Reproductor

1.3. TAD Lista de Reproducción (LR)

Un administrador de música no está completo sin una lista de canciones a reproducir. El objetivo es tener una estructura dinámica que contenga las canciones que después van a ser reproducidas. La estructura a utilizar es un *árbol binario de búsqueda*. En específico se usará en el modelo de representación un árbol binario de nodo, es decir un árbol con información en los nodos internos más no en las hojas, en los que cada nodo contiene un elemento de tipo `Canción`. Para el modelo concreto se define el tipo de datos `ArbolDeCanciones` al que corresponde el siguiente tipo algebraico libre [1] que se muestra en la Figura 3.

$$\begin{aligned}\textbf{freeType}\ \ \mathit{ArbolDeCanciones} \ &= \ \underline{\text{avac}} \\ &\mid \underline{\text{nodo}}\,(\mathit{ArbolDeCanciones}\,,\ \text{Canción}\,,\ \mathit{ArbolDeCanciones}) \quad .\end{aligned}$$

Figura 3: Tipo de dato concreto `ArbolDeCanciones`

Para ordenamiento de los objetos en el árbol se va hacer uso de dos claves. La clave primaria es el intérprete de la canción y la secundaria es el título de la canción. Es decir, si hay dos nodos con canciones que poseen intérpretes diferentes, entonces el intérprete es la clave usada para el ordenamiento. Si por el contrario dos canciones tienen el mismo intérprete, entonces se usa el título para determinar la posición de la canción en el árbol. El invariante de representación se exige que la variable de tipo `ArbolDeCanciones` sea un árbol binario de búsqueda y que todos los elementos de tipo canción, que se encuentran almacenados en los nodos, sigan el orden deseado usando la claves intérprete y título. Para verificar estas condiciones se define la función `esArbolDeBusqCancion` inductivamente sobre la estructura `ArbolDeCanciones` como indica la Figura 4

$$\begin{aligned}&\mathit{esArbolDeBusqCancion} : \mathit{ArbolDeCanciones} \to \text{Boolean} \\ &\quad \text{con cláusulas} \\ &\quad \left[\begin{array}{ll} \mathit{esArbolDeBusqCancion}(\underline{\text{avac}}) \ = \ \text{True} \quad , \\ \mathit{esArbolDeBusqCancion}(\underline{\text{nodo}}(\mathit{izq}, c, \mathit{der})) = & (c.\mathit{interprete} < \mathit{minInterprete}(\mathit{der}) \vee \\ & \quad (c.\mathit{interprete} = \mathit{minInterprete}(\mathit{der}) \wedge \\ & \quad c.\mathit{titulo} < \mathit{minTitulo}(\mathit{der}))) \\ & \wedge\ \mathit{esArbolDeBusqCancion}(\mathit{der}) \\ & \wedge\ (c.\mathit{interprete} > \mathit{maxInterprete}(\mathit{izq}) \vee \\ & \quad (c.\mathit{interprete} = \mathit{maxInterprete}(\mathit{izq}) \wedge \\ & \quad c.\mathit{titulo} > \mathit{maxTitulo}(\mathit{izq}))) \\ & \wedge\ \mathit{esArbolDeBusqCancion}(\mathit{izq}) \end{array}\right.\end{aligned}$$

Figura 4: Definición de la función *esArbolDeBusqCancion*

En la función `ArbolDeCanciones` se tiene que c es un elemento de tipo `Canción`. Las funciones *minInterprete* y *maxInterprete* determinan el menor y mayor valor del string que representa a un intérprete que se encuentran en el árbol de búsqueda binaria, usando ordenamiento lexicográfico. De la misma manera las funciones minTitulo y maxTitulo tienen como fin encontrar el menor y mayor valor de un título de una canción del árbol binario de búsqueda.

La primera operación del TAD LR es *agregarLista*, recibe como entrada el nombre de un archivo con una lista de canciones, las cuales carga en la lista de reproducción. El formato del archivo con la lista de canciones es el siguiente. Cada línea corresponde a una canción y la misma posee tres campos, el primero es el intérprete de la canción, el segundo es el título de la canción y el tercero es la dirección absoluta en el sistema de archivos del archivo de audio. Los campos están separados por ";". La Figura 5 muestra un ejemplo de un archivo con un formato válido, con una lista de cuatro canciones. Por cada línea del archivo

con la lista de canciones, el método *agregarLista* crea un elemento de tipo `Canción` para luego agregarlo en la estructura árbol de canciones. La segunda operación del TAD LR es *eliminarCancion*, que remueve una canción del árbol de canciones, dado el intérprete y el título de la canción. La tercera operación es *obtenerLR* que retorna una secuencia con todas las canciones contenidas en el árbol binario de búsqueda de canciones. Para construir la secuencia de elementos de tipo `Canción` debe usar la función `deArbolASecuencia`, la cual se define inductivamente como se muestra en la Figura 6. La cuarta y última operación del TAD LR es *mostrarLR*, que muestra por la salida estándar cada canción c contenida en el árbol de canciones. El árbol se debe recorrerse in-orden y cada elemento c de tipo `Canción` debe mostrar su contenido haciendo aplicando método $c.aString()$. En la Figura 7 se muestra la especificación completa del TAD LR.

```
Katy Perry;Dark Horse;/home/gpalma/MiMusica/katy_perry-dark_horse.mp3
Pharrell Williams;Happy;/home/gpalma/MiMusica/Pharrell HAPPY.mp3
OneRepublic;Counting Stars;/home/gpalma/MiMusica/Counting Stars.mp3
Katy Perry;Roar;/home/gpalma/MiMusica/Katy Perry - Roar.mp3
```

Figura 5: Ejemplo de un archivo que contiene la información de cuatro canciones

$$
\begin{array}{l}
\textit{deArbolASecuencia} : \text{ArbolDeCanciones} \rightarrow \text{seq} \\
\quad \text{con cláusulas} \\
\quad \left[\begin{array}{l}
\textit{deArbolASecuencia}(\underline{\text{avac}}) = <> \quad , \\
\textit{deArbolASecuencia}(\underline{\text{nodo}}(izq, c, der)) = \textit{deArbolASecuencia}(izq) + < c > + \textit{deArbolASecuencia}(der)
\end{array} \right.
\end{array}
$$

Figura 6: Definición de la función *deArbolASecuencia*

2. Módulo de administración de música

Una vez definidos los TADs podemos crear una aplicación que administre archivos de música. En este caso vamos a describir un módulo cliente que hace uso de los TADs como librerías, en términos de la implementación concreta en `Python`. A esta aplicación la vamos a llamar *Administrador de Música* (ADM). El ADM interactúa con los usuarios por medio de menú iterativo que presenta las siguientes opciones:

1. Cargar lista de reproducción
2. Mostrar lista de reproducción
3. Eliminar canción
4. Reproducir
5. Pausar
6. Parar
7. Próxima canción
8. Salir del administrador de música

A continuación se explican cada uno de las opciones.

Especificación del TAD LR

Modelo de Representación

var *contenido* : ArbolDeCanciones .

Invariante de Representación

esArbolDeBusqCancion(*contenido*)

Operaciones

constructor *crearLR* ( ) → LR
{ **Pre:** True }
{ **Post:** *self*.*contenido* = NULL }

meth *agregarLista* (**in** *na* : String)
{ **Pre:** *na* es el nombre de un archivo que posee un formato válido con una lista de canciones }
{ **Post:** Con cada linea de *na* se crea un elemento de tipo Canción
el cual es agregado a *self*.*contenido* en el orden en que aparece en *na* }

meth *eliminarCancion* (**in** *i* : String; **in** *t* : String)
{ **Pre:** True }
{ **Post:** Elimina de *self*.*contenido* la canción que tenga como intérprete a *i* y como título a *t* }

fmeth *obtenerLR* ( ) → seq
{ **Pre:** True }
{ **Post:** *obtenerLR* = *deArbolASecuencia*(*contenido*) }

meth *mostrarLR* ( )
{ **Pre:** True }
{ **Post:** Para cada canción *c* en *self*.*contenido*, se muestra por la salida estándar
el String *c*.*aString*(), recorriendo el árbol de búsqueda *self*.*contenido* in-orden }

Fin TAD

Figura 7: Especificación del TAD Lista de Reproducción

Cargar lista de reproducción: Solicita al usuario el nombre del archivo de datos con la lista de canciones con el formato válido explicado anteriormente. Se debe crear un objeto que corresponda a la `Lista de Reproducción` y almacenar en él las canciones especificadas en el archivo. Un usuario puede usar esta opción para cargar varios archivos de datos de canciones.

Mostrar lista de reproducción: Muestra por la salida estándar las canciones almacenadas hasta ahora en la `Lista de Reproducción` del administrador de música.

Eliminar canción: Debe mostrar la lista de reproducción y permitir al usuario seleccionar al usuario la canción que quiere remover y luego esta es eliminada de la lista de reproducción.

Reproducir: Antes de reproducir cualquier archivo de audio es necesario que se hayan cumplido varios pasos. Se debe haber cargado la librería de audio y se debe tener la lista de las canciones a reproducir. La lista de las canciones a reproducir se obtiene mediante la ejecución del método `obtenerLR` de un objeto `LR`. Una vez que empieza a reproducir un archivo de audio se debe mostrar al usuario el nombre del intérprete y el título de la canción.

Pausa: Hace una pausa en la reproducción del archivo de audio. Se debe mostrar al usuario el el nombre del intérprete y el título de la canción que esta en pausa.

Parar: Si en el momento de escoger esta opción hay una canción reproduciéndose, entonces se debe parar su reproducción. En la lista de reproducción esta canción que se paró es la próxima a tocarse. Si luego de parar una canción, el usuario selecciona la opción de **Reproducir**, entonces la canción comienza desde el principio.

Próxima canción: Al seleccionar esta opción se debe comenzar a reproducir la próxima canción en la lista de reproducción, ya sea que en ese momento se este tocando una canción o no. En caso de que haya una canción reproduciéndose, entonces de debe parar su reproducción y se procede a reproducir la próxima en la lista de reproducción. Igual que para reproducir, se debe mostrar al usuario el nombre del intérprete y el título de la canción que se esta reproduciendo.

Salir del administrador de música: Se termina la ejecución de la aplicación.

Es responsabilidad de este módulo verificar las precondiciones antes de llamar a las métodos de los TADs, en este caso se le debe indicar al usuario que la operación no pudo ser efectuada porque no se cumple la precondición, indicando cual es la precondición

3. Requerimientos de la implementación

Cada uno de los TADs debe ser implementado como una clase de `Python` y el módulo ADM debe ser implementado en un archivo aparte. Por lo tanto debe hacer entrega de por lo menos cuatro archivos con los siguientes nombres: `cancion.py`, `reproductor.py`, `lr.py` y `adm.py`.

Para la implementación de TAD Reproductor se requiere el uso de una librería de audio para poder tener las funcionalidades de un reproductor con un archivo de audio. Para este proyecto vamos hacer uso de la librería Pygame, disponible en `http://pygame.org`. Pygame es una librería madura y está bien documentada. En el sitio web se encuentran las instrucciones para su instalación además de la documentación. Su programa debe funcionar en la plataforma Linux. Si su aplicación no se ejecuta en Linux la nota del proyecto será *cero*. El código debe estar debidamente documentado y cada uno de los métodos de los TADs deben tener los siguientes elementos: descripción, parámetros, precondición y postcondición. Además debe hacer uso de la guía de estilo de `Python`.

4. Condiciones de entrega

El trabajo es por equipos de laboratorio. Debe entregar los códigos fuentes de sus programas en un archivo comprimido llamado `Proy2ci2692em20-X-Y.tar.xz` donde X y Y son los números de carné de los integrantes del grupo. La entrega del archivo *Proy2ci2692em20-X-Y.tar.xz*, debe hacerse al profesor del laboratorio por email, antes de las 6:00 pm del día domingo 29 de marzo de 2020.

Referencias

[1] Ravelo, J y Fernández, K. Tipos algebraico-libres. `https://ldc.usb.ve/~jravelo/docencia/algoritmos/material/talibres.pdf`, 2011.

Guillermo Palma / gvpalma@usb.ve / Marzo 2020