Trabajo Práctico 3
Listas enlazadas y Árboles binarios de búsqueda

Estructuras de datos
Comision A (Turno Mañana)
Segundo cuatrimestre de 2020

Docentes:	Ariel Clocchiatti Sergio Gonzalez Román García
Alumnos:	Kener, Sebastian

Introducción

En este trabajo nos centrándonos en las estructuras de datos Lista enlazada y Árboles binarios de búsqueda. Se realizo el modelamiento en Phyton y se hizo funcionar respetando es script de prueba y los datos provistos por los docentes.

TDA Arbol de Canciones

```
class ArbolDeCanciones:
  def __init__(self):
    self.raiz = None

  def estaVacio(self):
    return self.raiz == None
```

La primera funcion __init__(self), es el constructor de la clase. Crea una variable que es la raíz del árbol.
La segunda función, estaVacio(self), nos indica si la raíz esta o no vacia.

```
def insertarCanciones(self, listaCanciones, nombreInterprete):
  nuevoNodo = NodoArbolDeCanciones(nombreInterprete,listaCanciones)
  if self.estaVacio():
    self.raiz = nuevoNodo
  else:
    self.raiz.insertarInterprete(nuevoNodo)
```

Inserta cada una de las canciones de la lista en el árbol. Si el intérprete ya existe en el árbol, agrega cada canción a la lista de canciones (se debe verificar previamente si la canción está en la lista, para no duplicar información). Si el intérprete no existe en el árbol, agrega un nuevo nodo con el nuevo intérprete y las canciones en el lugar correspondiente.

```
def interpretesDeCancion(self, nombreCancion):
      listaInterpretes = Lista()
      listaInterpretes = self.raiz.interpretesDeCanciones(nombreCancion)
      return listaInterpretes
```

Recibe el nombre de una canción y retorna una lista de todos los interpretes que tienen una canción con ese nombre (pueden ser uno o más intérpretes). Si no hay ninguna canción con ese nombre almacenada en el árbol, retorna una lista vacía.

```
def buscarCanciones(self, listaInterpretes):
  listaDeCancionesCompartidas = Lista()
  if self.raiz != None:
    listaDeCancionesCompartidas = self.raiz.buscarCanciones(listaInterpretes)
  return listaDeCancionesCompartidas
```

Recibe una lista con los intérpretes buscados. Retorna una lista con los nombres de las canciones compartidas por todos los intérpretes de la lista de entrada. Si no hay ninguna canción compatida por todos los intérpretes, debe retornar una lista vacía.

```
def eliminarInterprete(self, nombreInterprete):
  if not self.estaVacio():
    if nombreInterprete == self.raiz.interprete:
      if self.raiz.grado() == 2:
        nodoPred = self.raiz.predecesor()
        self.eliminarInterprete(nodoPred.interprete)
        nodoPred.izquierdo = self.raiz.izquierdo
        nodoPred.derecho = self.raiz.derecho
        self.raiz = nodoPred
      elif self.raiz.tieneIzquierdo():
        self.raiz = self.raiz.izquierdo
      elif self.raiz.tieneDerecho():
        self.raiz = self.raiz.derecho
      else:
        self.raiz = None
    else:
      self.raiz.eliminarInterprete(nombreInterprete)
```

Elimina del árbol el intérprete que recibe por parámetros.

```
def eliminarCancion(self,nombreCancion):
      if not self.estaVacio():
        self.raiz.eliminaCancion(nombreCancion)
```

Elimina del árbol la canción que recibe por parámetro. Debe eliminarla de todos los nodos del árbol donde se encuentre.

```
def cantidadTotalInterpretes(self, Palabra):
  listaInterprete = Lista()
  if not self.estaVacio():
    resultado = self.raiz.cantidadTotalInterpretes(Palabra, listaInterprete)
  return resultado.len()
```

Recibe una palabra por parámetro y retorna la cantidad total de intérpretes almacenados en el árbol que tienen esa palabra formando parte de su nombre (completa o como parte de otra. Ej: La palabra jugando forma parte de la palabra conjugando).

```
def raizBalanceada(self):
        salida = False
        if self.raiz.grado() == 2:
          if self.raiz.diferenciaDeAltura() <=1:
            salida = True
        return salida
```

Retorna True si la diferencia de altura entre los subárboles hijos de la raiz es menor o igual a 1 y False en caso contrario.

```
def cancionesEnNivel(self,nivel):
        lista=Lista()
        if not self.estaVacio():
            self.raiz.cancionesEnNivel(lista,nivel,0)
        return lista
```

Recibe un nivel y retorna una lista con las canciones de todos los intérpretes que están en ese nivel del árbol. La lista no debe tener canciones repetidas. Si en ese nivel no hay nada, retorna una lista vacía.

```
def interpretesConMasCanciones(self,cantidadCancionesMinima):
        masCanciones=0
        lista=Lista()
        if not self.estaVacio():
          self.raiz.interpretesConMasCanciones(cantidadCancionesMinima,lista)
          masCanciones=lista.len()
        return masCanciones
```

Recibe una cantidad de canciones por parámetro y retorna la cantidad de intérpretes que tienen esa cantidad de canciones o más almacenadas en el árbol.

```
def internosAlfabetico(self):
        lista=Lista()
        if not self.estaVacio():
          self.raiz.internosAlfabetico(lista)
        return lista
```

Retorna una lista con los intérpretes que se encuentran en un nodo interno del árbol (no hojas). La lista debe contener los nombres de los intérpretes en orden alfabético.

TDA Nodo Arbol de Canciones

```
def __init__(self, interprete, listaDeCanciones):
  self.interprete = interprete
  self.canciones = listaDeCanciones
  self.izquierdo = None
  self.derecho = None
```

Constructor de la clase NodoArbolDeCanciones, inicia una variable interprete, una lista de canciones (canciones) y dos punteros hacia los hijos del árbol.

```
def tieneIzquierdo(self):
  return self.izquierdo != None
```

Devuelve true o False, si tiene o no nodo izquierdo.

```
def tieneDerecho(self):
  return self.derecho != None
```

Devuelve true o False, si tiene o no nodo derecho.

```
def insertarCanciones(self, listaNueva):
  if not self.canciones.estaLaCancion(listaNueva.primero.dato):
    self.canciones.append(listaNueva.primero.dato)
  else:
    print("La cancion ya esta en la lista")
```

Recibe una lista con canciones y si no esta en la lista del nodo actual la agrega.

```
def insertarInterprete(self, nuevoNodo):
  if nuevoNodo.interprete < self.interprete:
    if not self.tieneIzquierdo():
      self.izquierdo = nuevoNodo
    else:
      self.izquierdo.insertarInterprete(nuevoNodo)
  elif nuevoNodo.interprete > self.interprete:
    if not self.tieneDerecho():
      self.derecho = nuevoNodo
    else:
      self.derecho.insertarInterprete(nuevoNodo)
  elif nuevoNodo.interprete == self.interprete:
    self.insertarCanciones(nuevoNodo.canciones)
```

Recibe un nodo y verifica si el interprete esta o no, si no esta lo agrega, y si esta agrega las canciones a la lista.

```
def interpretesDeCanciones(self, unaCancion):
  listaInterpretes = Lista()
  if self.canciones.estaLaCancion(unaCancion):
    listaInterpretes.append(self.interprete)
  if self.tieneIzquierdo():
    self.izquierdo.interpretesDeCanciones(unaCancion)
  if self.tieneDerecho():
    self.derecho.interpretesDeCanciones(unaCancion)
  return listaInterpretes
```

Recibe una cancion, revisa todas las listas de canciones del arbol y devuelve una lista de interpretes en la que se encuentra la canción.

```
def buscarInterprete(self, interprete):
  nodoDato = None
  #print(nodoDato)
  if self.interprete == interprete:
    nodoDato = self
  elif self.tieneIzquierdo():
      nodoDato = self.izquierdo.buscarInterprete(interprete)
  elif self.tieneDerecho():
      nodoDato = self.derecho.buscarInterprete(interprete)
  return nodoDato
```

Recibe un interprete y devuelve el nodo en el que esta.

```
def buscarCanciones(self, listaInterpretes):
  pos = 0
  cantDeInterpretesEnLista = listaInterpretes.len()
  listaDeCanciones = Lista()
  cantidadCanciones = 0
  posicionCanciones = 0
  nodoUno = None
  nodoDos = None
  if cantDeInterpretesEnLista > 1:
    while pos < cantDeInterpretesEnLista:
      interpretePosMasUno = listaInterpretes.get(pos+1)
      if interpretePosMasUno != None:
        nodoUno = self.buscarInterprete(listaInterpretes.get(pos))
        nodoDos = self.buscarInterprete(listaInterpretes.get(pos+1))
        if nodoUno != None and nodoDos != None:
          cantidadCanciones = nodoUno.canciones.len()
          while posicionCanciones < cantidadCanciones:

            if nodoDos.canciones.estaLaCancion(nodoUno.canciones.pop(posicionCanciones)):
              listaDeCanciones.append(nodoUno.canciones.pop(posicionCanciones))
            else:
              posicionCanciones+=1
      else:
        print("Al menos un interprete no esta en el arbol")
      pos = pos + 1

  else:
    return listaDeCanciones
```

Recibe una lista de interpretes, tiene que ser mas de uno, y devuelve una lista con las canciones que comparten entre si.

```
def buscaPadre(self, dato):
  nodoHijo = None
  nodoPadre = None
  lado = None
  if dato < self.interprete:
    if self.tieneIzquierdo():
      if self.izquierdo.interprete == dato:
        nodoHijo = self.izquierdo
        nodoPadre = self
        lado = "izq"
      else:
        nodoHijo, nodoPadre, lado = self.izquierdo.buscaPadre(dato)
  else:
    if self.tieneDerecho():
      if self.derecho.interprete == dato:
        nodoHijo = self.derecho
        nodoPadre = self
        lado = "der"
      else:
        nodoHijo, nodoPadre, lado = self.derecho.buscaPadre(dato)
  return nodoHijo, nodoPadre, lado
```

Recibe un interprete(dato), y devuelve el padre de este nodo y a que lado esta.

```
def eliminarInterprete(self, nombreInterprete):
  nodoEliminar, nodoPadre, lado = self.buscaPadre(nombreInterprete)
  if nodoEliminar != None:
    if nodoEliminar.grado() == 2:
      nodoPred = nodoEliminar.predecesor()
      self.eliminar(nodoPred.nombreInterprete)
      nodoPred.izquierdo = nodoEliminar.izquierdo
      nodoPred.derecho = nodoEliminar.derecho
      if lado == "izq":
        nodoPadre.izquierdo = nodoPred
      else:
        nodoPadre.derecho = nodoPred
    elif nodoEliminar.tieneIzquierdo():
      if lado == "izq":
        nodoPadre.izquierdo = nodoEliminar.izquierdo
      else:
        nodoPadre.derecho = nodoEliminar.izquierdo
    elif nodoEliminar.tieneDerecho():
      if lado == "izq":
        nodoPadre.izquierdo = nodoEliminar.derecho
      else:
        nodoPadre.derecho = nodoEliminar.derecho
    else:
      if lado == "izq":
        nodoPadre.izquierdo = None
      else:
        nodoPadre.derecho = None
```

Recibe el nombre de un interprete y elimina el nodo en el que se encuentra el interprete.

```
def altura(self):
  alt=0
  if self.grado()==2:
    alt=1+max(self.izquierdo.altura(),self.derecho.altura())
  elif self.grado()==1:
    if self.tieneIzquierdo():
      alt=1+(self.izquierdo.altura())
    else:
      alt=1+(self.derecho.altura())
  return alt
```

Indica la altura del arbol que se encuentra el nodo actual.

```
def diferenciaDeAltura(self):
  return abs(self.izquierdo.altura() - self.derecho.altura())
```

Devuelve la diferencia de altura entre el nodo izquierdo y el derecho del nodo actual.

```
def grado(self):
    grado=0
    if self.tieneDerecho():
      grado+=1
    if self.tieneIzquierdo():
      grado+=1
    return grado
```

Devuelve el grado del nodo actual

```
def esHoja(self):
    return not self.tieneDerecho() and not self.tieneIzquierdo()
```

Devuelve True o False, revisa si el nodo actual es hoja(el ultimo de la rama del arbol) o no.

```
def cancionesEnNivel(self,lista,nivel,nivelActual):
      if nivelActual==nivel:
          self.guardarListaEnLista(lista)
      if self.tieneIzquierdo():
          self.izquierdo.cancionesEnNivel(lista, nivel,nivelActual+1)
      if self.tieneDerecho():
          self.derecho.cancionesEnNivel(lista, nivel,nivelActual+1)
```

Recibe una lista, un nivel y el nivel del nodo actual. Guarda las canciones del nivel ingresado en la lista.

```
def guardarListaEnLista(self,lista):
    largoDeLista=self.canciones.len()
    elementoNuevo=None
    pos=0
    while pos<largoDeLista:
        elementoNuevo=self.canciones.get(pos)
        #print(elementoNuevo)
        if lista.estaVacia():
            lista.append(elementoNuevo)
        elif not lista.estaLaCancion(elementoNuevo):
                lista.append(elementoNuevo)
        pos+=1
```

Recibe una lista y guarda las canciones del nodo actual en ella.

```
def interpretesConMasCanciones(self,cantidadCancionesMinima,lista):
        if self.canciones.len()>=cantidadCancionesMinima:
            lista.append(self.interprete)
        if self.tieneIzquierdo() :
            self.izquierdo.interpretesConMasCanciones(cantidadCancionesMinima,lista)
        if self.tieneDerecho() :
            self.derecho.interpretesConMasCanciones(cantidadCancionesMinima,lista)
```

Recibe un numero (cantidadCancionesMinima) y una lista, si la cantidad de canciones del nodo actual es mayor o igual al numero ingresado, se agrega el interprete a la lista.

```
def internosAlfabetico(self,lista):
        if self.tieneIzquierdo():
            self.izquierdo.internosAlfabetico(lista)
        if self.canciones.primero.dato[0].isupper() and not self.esHoja():
            lista.append(self.canciones.primero.dato)
        if self.tieneDerecho() :
            self.derecho.internosAlfabetico(lista)
```

Recibe una lista de interpretes y los ordena alfabéticamente.

```
def eliminaCancion(self,nombreCancion):
    posicion=None
    if self.canciones.estaLaCancion(nombreCancion):
            posicion = self.canciones.buscarCancionYRetornarPosicion(nombreCancion)
            self.canciones.pop(posicion)

    else:
        if self.tieneIzquierdo() :
                self.izquierdo.eliminaCancion(nombreCancion)

        if self.tieneDerecho() :
                self.derecho.eliminaCancion(nombreCancion)
```

Recibe el nombre de una cancion y la elimina de la lista.

```
def estaPalabraEnInterprete(self, palabra):
  resultado = self.interprete.find(palabra)
  booleano = resultado >= 0
  return booleano
```

Se ingresa una palabra y revisa si esta en el nombre del interprete actual. Devuelve un booleano.

```
def cantidadTotalInterpretes(self, Palabra, lista):
    if self.estaPalabraEnInterprete(Palabra):
      print(self.interprete)
      lista.append(self.interprete)
    if self.tieneIzquierdo():
        self.izquierdo.cantidadTotalInterpretes(Palabra, lista)
    if self.tieneDerecho():
        self.derecho.cantidadTotalInterpretes(Palabra, lista)

    return  lista
```

Recibe una palabra y una lista. Revisa los interpretes y pone en la lista los que tienen en su nombre la palabra ingresada.

TDA Lista

El TDA Lista es una lista enlazada con las funciones estándar, salvo por las siguientes:

```
def estaLaCancion(self, cancion):
  respuesta = False
  if cancion == self.primero.dato:
      respuesta = True
  elif self.primero.tieneSiguiente():
      respuesta = self.primero.siguiente.estaLaCancion(cancion)
  return respuesta
```

Recibe una cancion y devuelve True o False. Si esta o no la cancion en la lista.

```
def buscarCancionYRetornarPosicion (self,unaCancion):
    posAux = 0
    posicion = None
    nodoAux = self.primero
    while nodoAux != None:
        if nodoAux.dato == unaCancion:
            posicion = posAux
        posAux+=1
        nodoAux = nodoAux.siguiente
    return posicion
```

Recibe una cancion, la busca en la lista, y retorna la posicion en la que se encuentra.