Laboratorio No.3: Arreglos y listas encadenadas

Mariana Díaz Arenas 202020993 & Nicole Murillo Fonseca 202025521
Universidad de Los Andes
11 de Febrero 2021

Paso 3: Estudiar el ejemplo en VS Code

1. ¿Cuáles son los mecanismos de interacción (I/O: Input/Output) que tiene el view.py con el usuario?

   El primer mecanismo de interacción presente en el view.py es el menú principal, desde el cual el usuario puede elegir la función que desee que ejecute el programa. Lo que el usuario ve es la segunda imagen, una variedad de opciones a ejecutar con su respectivo número; ahora bien, el usuario puede escribir el número que desea porque hay un condicional while que imprime el menú con un input mientras esté en True o la opción "0- Salir" no sea elegida. Una vez que el usuario digita un número en el input el while se encarga de buscar la función correspondiente y efectuarla.

```
"""
Menu principal
"""
while True:
    printMenu()
    inputs = input('Seleccione una opción para continuar\n')
    if int(inputs[0]) == 1:
```

Imagen 1. Input del menú principal

```
def printMenu():
    print("Bienvenido")
    print("1- Cargar información en el catálogo")
    print("2- Consultar los Top x libros por promedio")
    print("3- Consultar los libros de un autor")
    print("4- Libros por género")
    print("0- Salir")
```

Imagen 2. Print del menú principal

2. ¿Cómo se almacenan los datos de GoodReads en el model.py?

En el model.py para almacenar los datos, primero se implementa la función newCatalog() que retorna un catálogo con listas vacías para guardar posterior y respectivamente los libros, los autores, los géneros (tags) y finalmente la asociación géneros y libros (booktags).

```
def newCatalog():

    catalog = {'books': None,
               'authors': None,
               'tags': None,
               'book_tags': None}

    catalog['books'] = lt.newList()
    catalog['authors'] = lt.newList('ARRAY_LIST',
                                    cmpfunction=compareauthors)
    catalog['tags'] = lt.newList('ARRAY_LIST',
                                 cmpfunction=comparetagnames)
    catalog['book_tags'] = lt.newList('ARRAY_LIST')

    return catalog
```

Imagen 3. Función newCtalog()

Una vez que se creó el catálogo vacío se empieza a agregar la información en las listas respectivas por medio de las funciones addBook() para agregar los libros, addBookAuthor() para agregar los autores, addTag() para agregar los géneros (tags) y por último la función addBookTag() para agregara la asociación de géneros y libros (booktags).

En la función addBook() se agrega el nombre del libro en la lista vacía de libros y los autores se dividen en otra lista denominada 'authors' con el método Split. Luego ese autor es agregado al catalogo con la siguiente función definida en el model.py addBookAuthor().

```
def addBook(catalog, book):
    # Se adiciona el libro a la lista de libros
    lt.addLast(catalog['books'], book)
    # Se obtienen los autores del libro
    authors = book['authors'].split(",")
    # Cada autor, se crea en la lista de libros del catalogo, y se
    # crea un libro en la lista de dicho autor (apuntador al libro)
    for author in authors:
        addBookAuthor(catalog, author.strip(), book)
```

Imagen 4. Función addBook()

En la siguiente función, addBookAuthor(), implementada dentro de addBook(), se revisa si el nombre del autor que se quiere agregar, ya está es la lista de autores del catálogo, si está se usa lt.getElement para agregar ese autor a la lista pero si no está se usa otra función, que será explicada más adelante, para crear un nuevo autor.

```python
def addBookAuthor(catalog, authorname, book):
    """
    Adiciona un autor a lista de autores, la cual guarda referencias
    a los libros de dicho autor
    """
    authors = catalog['authors']
    posauthor = lt.isPresent(authors, authorname)
    if posauthor > 0:
        author = lt.getElement(authors, posauthor)
    else:
        author = newAuthor(authorname)
        lt.addLast(authors, author)
    lt.addLast(author['books'], book)
```

Imagen 5. Función addBookAuthor()

Las siguientes dos funcionas addTag() y addBookTag() agregan respectivamente un género o tag a la lista de tags del catálogo y la asociación géneros y libros (booktags) a la lista de book_tags en el catálogo, utilizando el método addlast(). Además utilizan las funciones newTag() y newBookTag que serán explicadas a continuación.

```python
def addTag(catalog, tag):
    """
    Adiciona un tag a la lista de tags
    """
    t = newTag(tag['tag_name'], tag['tag_id'])
    lt.addLast(catalog['tags'], t)


def addBookTag(catalog, booktag):
    """
    Adiciona un tag a la lista de tags
    """
    t = newBookTag(booktag['tag_id'], booktag['goodreads_book_id'])
    lt.addLast(catalog['book_tags'], t)
```

Imagen 6. Funciones addTag() y addBookTag()

Las funciones newAuthor(), newTag() y newBookTag sirven para crear datos y retornar un diccionario con la información específica y requerida para cada categoría: autor, género (tags) y la asociación género y libro (booktags). En la imagen 7 se pueden observar los datos de cada diccionario. Estas funciones son implementadas por las mencionadas anteriormente si el nombre del autor, género (tags) o la asociación género y libro (booktag) aún no existen en sus respectivas listas.

```python
def newAuthor(name):
    """
    Crea una nueva estructura para modelar los libros de
    un autor y su promedio de ratings
    """
    author = {'name': "", "books": None,  "average_rating": 0}
    author['name'] = name
    author['books'] = lt.newList('ARRAY_LIST')
    return author


def newTag(name, id):
    """
    Esta estructura almancena los tags utilizados para marcar libros.
    """
    tag = {'name': '', 'tag_id': ''}
    tag['name'] = name
    tag['tag_id'] = id
    return tag


def newBookTag(tag_id, book_id):
    """
    Esta estructura crea una relación entre un tag y
    los libros que han sido marcados con dicho tag.
    """
    booktag = {'tag_id': tag_id, 'book_id': book_id}
    return booktag
```

Imagen 7. Funciones de creación (diccionarios)

Finalmente, hay unas funciones definidas de consulta; compareauthors() y comparetagnames(), estas las podemos observar en la función newCatalog() en la creación de listas de 'autores' y 'tags', como parámetro del método newlist().

```python
def compareauthors(authorname1, author):
    if (authorname1.lower() in author['name'].lower()):
        return 0
    return -1


def comparetagnames(name, tag):
    return (name == tag['name'])
```

Imagen 8. Funciones para comparar elementos utilizados en la función newCatalog()

3. ¿Cuáles son las funciones que comunican el view.py y el model.py?

   Las funciones que comunican el view.py y el model.py son las funciones de consulta: getBookByAuthor(), getBestBooks() y CountBooksByTag().

```python
def getBooksByAuthor(catalog, authorname):
    """
    Retorna un autor con sus libros a partir del nombre del autor
    """
    posauthor = lt.isPresent(catalog['authors'], authorname)
    if posauthor > 0:
        author = lt.getElement(catalog['authors'], posauthor)
        return author
    return None


def getBestBooks(catalog, number):
    """
    Retorna los mejores libros
    """
    books = catalog['books']
    bestbooks = lt.newList()
    for cont in range(1, number+1):
        book = lt.getElement(books, cont)
        lt.addLast(bestbooks, book)
    return bestbooks


def countBooksByTag(catalog, tag):
    """
    Retorna los libros que fueron etiquetados con el tag
    """
    tags = catalog['tags']
    bookcount = 0
    pos = lt.isPresent(tags, tag)
    if pos > 0:
        tag_element = lt.getElement(tags, pos)
        if tag_element is not None:
            for book_tag in lt.iterator(catalog['book_tags']):
                if tag_element['tag_id'] == book_tag['tag_id']:
                    bookcount += 1
    return bookcount
```

Imagen 9. Funciones que conectan al model.py con el view.py

Paso 4: Estudiar el uso de listas

1. ¿Cómo se crea una lista?

   Para crear una lista con el ATD se debe crear una función utilizando el comando newList(), al cual se le deben agregar cinco parámetros, donde cuatro son opcionales y uno es de tipo obligatorio. En este caso, se pueden crear dos tipos de listas, un ARRAY_LIST o una SINGLE_LINKED correspondientes a un arreglo o una lista encadenada, respectivamente. Así mismo, los argumentos de cmpfunction, key, filename y delimiter son opciones que permiten personalizar adecuadamente una lista pero que no es necesarias definirlas para la creación de la lista.

```
def newList(datastructure='SINGLE_LINKED',
            cmpfunction=None,
            key=None,
            filename=None,
            delimiter=","):
    """Crea una lista vacia

    Args:
        datastructure:  Tipo de estructura de datos a utilizar para implementar
        la lista. Los tipos posibles pueden ser: ARRAY_LIST y SINGLE_LINKED.

        cmpfunction: Función de comparación para los elementos de la lista.
        Si no se provee función de comparación se utiliza la función
        por defecto pero se debe proveer un valor para key.
        Si se provee una función de comparación el valor de Key debe ser None.

        Key:  Identificador utilizado para comparar dos elementos de la lista
        con la función de comaparación por defecto.

        filename: Si se provee este valor, se crea una lista a partir
        de los elementos encontrados en el archivo.
        Se espera que sea un archivo CSV UTF8.

        delimiter: Si se pasa un archivo en el parámetro filename, se utiliza
        este valor para separar los campos. El valor por defecto es una coma.

    Returns:
        Una nueva lista
    Raises:
        Exception
    """
```

Imagen 10. newList()

2. ¿Qué hace el parámetro cmpfunction = None en la función newList()?

   Teniendo en cuenta que el parámetro cmpfunction() funciona para realizar comparaciones de tipo (definidas por el usuario) dentro de la lista, si este está en None, False etc. significa que no se va a agregar a la lista.

3. ¿Qué hace la función addLast()?

   La función addLast () permite agregar elementos en la última posición de la lista, por lo cual, pide como parámetros la lista donde se desea agregar el elemento (lst) y el elemento que se quiere agregar a la lista (element). Igualmente, la lista automáticamente incrementa el size en uno para ajustarlo a la modificación.

```
def addLast(lst, element):
    """ Agrega un elemento en la última posición de la lista.

    Se adiciona un elemento en la última posición de la lista y se actualiza
    el apuntador a la útima posición. Se incrementa el tamaño de la lista en 1

    Args:
        lst: La lista en la que se inserta el elemento
        element: El elemento a insertar

    Raises:
        Exception
    """
```

Imagen 11. addLast()

4. ¿Qué hace la función getElement()?

   Esta función se encarga de recorrer la lista elemento por elemento para localizar un determinado elemento que se pase por parámetro, por lo cual, la función recibe la lista (lst) de la cual se quiere obtener el elemento y también se recibe la posición del elemento que se desea buscar (pos), el parámetro de posición debe ser mayor o igual a cero y menor o igual al tamaño de la lista. Se pide que la lista no esté vacía porque en ese caso, la función no funcionaría correctamente, así mismo, al obtener el elemento, no lo elimina de la lista principal, simplemente lo muestra.

```
def getElement(lst, pos):
    """ Retorna el elemento en la posición pos de la lista.

    Se recorre la lista hasta el elemento pos, el cual  debe ser mayor
    que cero y menor o igual al tamaño de la lista.
    Se retorna el elemento en dicha posición sin eleminarlo.
    La lista no puede ser vacia.

    Args:
        lst: La lista a examinar
        pos: Posición del elemento a retornar

    Raises:
        Exception
    """
```

Imagen 12. getElement()

5. ¿Qué hace la función subList()?

   El propósito de esta función es crear una nueva lista a partir de una ya existente que es indicada por parámetro, por lo cual, recibe una lista (lst), una posición(pos) y el número de elementos que se desea copiar (numelem). La función recorre la lista hasta llegar a la posición que se indica y cuando llega a esta, inicia una nueva lista con todos los elementos

hasta que haya recorrido el número de elementos especificados en el parámetro numelem, cuando finaliza esto crea una copia con todos los elementos en el rango creado y retorna esta nueva lista sin modificar la original.

```
def subList(lst, pos, numelem):
    """ Retorna una sublista de la lista lst.

    Se retorna una lista que contiene los elementos a partir de la
    posicion pos, con una longitud de numelem elementos.
    Se crea una copia de dichos elementos y se retorna una lista nueva.

    Args:
        lst: La lista a examinar
        pos: Posición a partir de la que se desea obtener la sublista
        numelem: Numero de elementos a copiar en la sublista

    Raises:
        Exception
    """
```

Imagen 13. subList()

Paso 5: Cambios en el uso de TAD lista (ADT list)

1. ¿Observó algún cambio en el comportamiento del programa al cambiar la implementación del parámetro “ARRAY_LIST” a “SINGLE_LINKED”?

   Cuando se implementó el programa con ARRAY_LIST el tiempo transcurrido fue de varios minutos.

```
catalog['books'] = lt.newList()
catalog['authors'] = lt.newList('ARRAY_LIST',
                                cmpfunction=compareauthors)
catalog['tags'] = lt.newList('ARRAY_LIST',
                             cmpfunction=comparetagnames)
catalog['book_tags'] = lt.newList('ARRAY_LIST')

return catalog
```

Imagen 14. Model.py con ARRAY_LIST

Sin embargo, cuando se cambió en el model.py a SINGLE_LINKED las listas asociadas con el manejo de los tags y los booktags, parecía el tiempo de ejecución del programa era más extenso que al usar ARRAY_LIST.

```
catalog['books'] = lt.newList()
catalog['authors'] = lt.newList('ARRAY_LIST',
                                cmpfunction=compareauthors)
catalog['tags'] = lt.newList('SINGLE_LINKED',
                             cmpfunction=comparetagnames)
catalog['book_tags'] = lt.newList('SINGLE_LINKED')
```

Imagen 15. Model.py con SINGLE_LINKED

Por esta razón, decidimos usar la función time() para verificar nuestra hipótesis de que el cambio a SINGLE_LINKED hizo que el tiempo de ejecución fuera mayor.

```
while True:
    printMenu()
    inputs = input('Seleccione una opción para continuar\n')
    st = time.time()
    if int(inputs[0]) == 1:
        print("Cargando información de los archivos ....")
        catalog = initCatalog()
        loadData(catalog)
        print('Libros cargados: ' + str(lt.size(catalog['books'])))
        print('Autores cargados: ' + str(lt.size(catalog['authors'])))
        print('Géneros cargados: ' + str(lt.size(catalog['tags'])))
        print('Asociación de Géneros a Libros cargados: ' +
              str(lt.size(catalog['book_tags'])))
        et = time.time()
        print("tiempo: " + str(et - st))
```

Imagen 16. Función time()

Cuando volvimos a ejecutar los programas para ARRAY_LIST y para SINGLE_LINKED los tiempos arrojados fueron respectivamente: 350.20 segundos y 403.44 segundos.

```
1
Cargando información de los archivos ....
Libros cargados: 10000
Autores cargados: 5833
Géneros cargados: 34252
Asociación de Géneros a Libros cargados: 999912
tiempo: 350.2020969390869
Bienvenido
```

Imagen 17. Tiempo para ARRAY_LIST

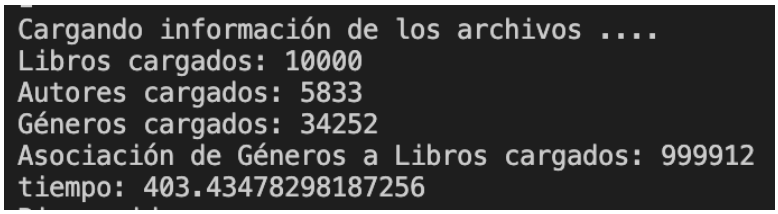

Imagen 18. Tiempo para SINGLE_LINKED

Como lo sospechábamos, al usar SINGLE_LINKED o una lista encadenada en vez de un arreglo, pudimos observar que el tiempo de ejecución fue mayor por casi 54 segundos más.