Implementación de una Lista Doblemente Enlazada

Elaborado Por: Arnaldo Quintero Segura.

Decisiones de Diseño

Para la implementación de esta lista enlazada. Se procederá a desarrollar cada una de las operaciones que se puede realizar utilizándola. Y una ligera explicación de por qué es de esta manera.

Antes, veamos la estructura de los nodos que componen cada elemento presente en la lista. Esta sería la clase `Node`, llamada así por la palabra en inglés para Nodo:

Node

Esta clase posee tres atributos: El valor del nodo, de tipo `String`, y los apuntadores a los Nodos anterior y siguiente.

Estos atributos de la clase son: `value`, `prev` y `next` respectivamente. Haciendo referencia a las palabras en inglés para valor, y las abreviaturas de anterior y siguiente. Estos tres atributos se encuentran encapsulados dentro de la clase, donde cada uno posee correspondientemente su `getter` y su `setter`.

Esta clase posee un único inicializador, que dado un parámetro de tipo `String`, inicializa un nodo con el valor introducido. Al cual luego se podrá acceder mediante su `getter`: `getValue()`. Al igual que los nodos anterior y siguiente, que se encuentran inicialmente con un valor `null`.

Propiedades de la Lista

Esta clase de lista doblemente enlazada llamada `DoublyLinkedList`, haciendo referencia a las palabras en inglés para ListaDoblementeEnlazada, posee también tres atributos: La cantidad elementos presentes en la lista, un apuntador al primer nodo de la lista, y un apuntador al último nodo de la lista. Siendo cada uno de estos atributos `count`, `head`, y `tail` correspondientemente, haciendo referencia a las palabras en inglés para cantidad, cabeza, y cola.

Los tres atributos de la clase se encuentran encapsulados, dado que para efectos de el consumidor final de la clase, no debe ser accesible como un objeto de tipo `Node`, sino como una lista de `String`. Por lo tanto, los `getters` de la cabeza y la cola: `getHead()` y `getTail()` correspondientemente, devuelven el valor contenido dentro de los nodos que se encuentran en dichas posiciones en la lista; en caso de ser una lista vacía, devuelve un valor `null` al usuario.

Listemos ahora cada una de las operaciones que soporta la clase `DoublyLinkedList`:

Inicializar

Para inicializar una instancia de `DoublyLinkedList`, se utiliza el constructor de la clase. El cual inicializa la cantidad de elementos presentes en la lista a cero. Los punteros `head` y `tail` se mantienen en `null` dado que no hay nada dentro de la lista.

Insertar

Esta implementación posee tres tipos distintos de inserción: insertar al comienzo de la lista, insertar al final de la lista e insertar en una posición específica de la lista. Siendo los métodos `insert`, `append` e `insertAtPosition` correspondientemente. Se decidió utilizar distintos modos de inserción dado que le da más versatilidad a la lista. Y además, como ya se posee un nodo apuntando al final de la lista, se puede aprovechar dándole al usuario un método de agregar al final.

Cada uno de estos métodos recibe un `String` como parámetro, ya que la implementación de los Nodos es transparente para el usuario final. Veamos cada uno de los métodos de inserción:

Insertar al comienzo

Se inserta directamente a la cabeza de la lista.

Insertar al final

Se inserta directamente al final de la lista.

Insertar en una posición indicada

Se revisa de que punta de la lista se encuentra más cerca el elemento a insertar, y se procede a recorrer desde esa punta hasta la posición indicada. Para luego insertar el nuevo nodo deseado. Este método tiene la particularidad de que puede lanzar excepciones, y que tiene varios casos borde.

Si el usuario introduce una posición menor que cero, el método devuelve una excepción de tipo `IllegalArgumentException`, diciéndole al usuario que la posición que introdujo no es valida.

Por el contrario, si el usuario introduce una posición mayor o igual a la cantidad de elementos que posee la lista, en vez de lanzar una excepción, simplemente se inserta el nuevo elemento al final de la lista.

Estas decisiones fueron tomadas, porque así no se permiten posiciones negativas en la lista, pero permite al usuario hacer una inserción al final en cualquier caso de exceso de conteo al final.

Recuperar elementos de la lista

Para recuperar elementos de la lista, al igual que para insertar, tenemos tres métodos. Dos de ellos ya mencionados anteriormente, siendo estos `getHead` y `getTail` para obtener respectivamente el primer y el último elemento de la lista.

Luego tenemos un método que recupera el elemento presente en la posición deseada de la lista, este método es `getElementAtPosition`, siendo esto la traducción al inglés para "obtener elemento en la posición". Este método recibe un entero `int` como parámetro, que es la posición a buscar en la lista. Si el número es menor que cero, al igual que al insertar, se lanza una excepción de tipo `IllegalArgumentException`.

En este caso, como intentamos recuperar elementos, si el usuario introduce una posición mayor o igual a la cantidad de elementos que hay en la lista, también recibirá una excepción del mismo tipo. Dado que en estas posiciones realmente no existe nada que mostrar.

Para el caso restante se recorre la lista desde el nodo extremo más cercano, bien sea desde la cabeza o desde la cola, hasta llegar al nodo indicado. Devolviendo al usuario el valor `String` contenido en ese nodo.

Para esta implementación, la clase posee un par de métodos privados: `getNodeByValue` y `getNodeAtPosition`, los cuales facilitan recuperar el nodo completo de la lista correspondiente utilizando un valor introducido o una posición específica. Para facilitar para futuros métodos, la extracción de los mismos para su manejo interno dentro de la implementación de la lista. Devolviendo siempre al usuario final valores de tipo `String`.

Revisar si un elemento se encuentra en la lista

Para revisar si un elemento se encuentra en la lista, se tiene el método `contains`, haciendo referencia a la palabra en inglés para contener. Este método revisa cada uno de los elementos dentro de la lista, y los compara con el valor que se está buscando. En caso de contenerlo, devuelve un valor `true`, de lo contrario `false`.

Reemplazar

Esta operación nos permite reemplazar el valor de el nodo en una posición concreta especificada. En este caso, se sigue el mismo criterio que en el método de recuperación de elementos con respecto a qué posiciones se consideran inválidas y la excepción correspondiente.

Para reemplazar el valor contenido en el nodo, utilizamos el método anteriormente especificado `getNodeAtPosition`, que devuelve el nodo asociado a esa posición. Y asumiendo que el nodo es existente, cambia el valor del nodo, utilizando el método `setValue` de la clase `Node`, por el valor introducido por el usuario.

Eliminar

Para eliminar de la lista, tenemos dos opciones: eliminar un valor, o eliminar el elemento en una posición concreta.

Eliminar un valor

Para eliminar un valor, se utiliza el método privado anteriormente mencionado `getNodeByValue`, además de otro método privado, `removeNode`, que se encarga de modificar los apuntadores correspondientes de la lista para eliminar un nodo existente.

Si el valor a eliminar no se encuentra en la lista, el método retorna `false`, de lo contrario `true`. Para que así el usuario pueda saber si el elemento deseado fue eliminado o no de la lista.

Eliminar una posición

Para eliminar una posición, se utilizan los métodos privados `getNodeAtPosition` y `removeNode` ya mencionados anteriormente. Y el método devuelve el valor del nodo eliminado de la lista. Para que de esta manera el usuario pueda confirmar qué fue eliminado de la lista.

Para este método se utiliza el mismo criterio de posiciones invalidas que los métodos de recuperación y reemplazo de valores de la lista.

Concatenar listas

Para concatenar dos listas, se tiene el método `concat`, que recibe una lista, y añade la lista suministrada al final de la lista que está invocando el método. En caso de ser una lista vacía, no hace nada.

Al concatenar dos listas, la lista a la a la que se concatena, se le modifica la cola, y pasa a apuntar al último elemento de la lista concatenada. Mientras que para la lista concatenada no se tocan los punteros. Esto asumiendo que el usuario dejará de usar la lista a concatenar y usará unicamente la lista concatenada.

Imprimir los valores de la lista

Para imprimir los valores de la lista, tenemos una función `print` que ya nos imprime a la salida estándar, `stdOut`, el contenido de la lista, dentro de corchetes, los elementos separados por coma, cada uno dentro de comillas.

En caso de que la lista se encuentre vacía, se imprime el mensaje `Empty List`, que traduce del inglés a lista vacía.

Este método utiliza la sobreescritura del método `toString` de Java. Haciendo por lo tanto posible el uso de los métodos normales de Java para imprimir valores por consola, tales como `println` u otros.

Prueba de funcionalidad

Se incluye además una serie de pruebas, tanto para la clase `Node` como para la clase `DoublyLinkedList`, en donde se prueba el funcionamiento de cada uno de los métodos públicos que posee la clase.

Dichas pruebas deben ser corridas utilizando la librería JUnit 5.