2014 Big Java Early Objets

**LINKED LIST**

* Una lista consiste en un número de nodos, de los cuales cada uno tiene una referencia al siguiente o anterior nodo.
* Es una estructura de datos utilizada para coleccionar una secuencia de objetos que permiten agregar y eliminar elementos de manera eficiente (en medio de la secuencia).
* Para comprender la ineficiencia de los arreglos y la necesidad de una estructura de datos más inteligente: imagina que tienes un programa que contiene una secuencia de nombres de empleados.
* Si un empleado abandona la compañía, el nombre debe ser removido.
* En un arreglo, el hueco en la secuencia debe cerrarse moviendo todos los objetos que vienen después del hueco.
* Por el contrario, si un empleado nuevo se agrega en el medio de la secuencia, todos los nombres que siguen deben moverse hacia adelante/final.
* Mover una gran cantidad de elementos puede involucrar una cantidad sustancial de tiempo de procesamiento. Una lista enlazada evita este movimiento.
* Una lista enlazada utiliza una secuencia de nodos, un nodo es un objeto que almacena un elemento y referencias a los nodos vecinos (nodo anterior y nodo siguiente) de la secuencia.

|  |
| --- |
|  |

* Cuando insertamos un nuevo nodo en la lista enlazada, solamente las referencias de los nodos vecinos son actualizadas.

|  |
| --- |
|  |

* Lo mismo ocurre cuando se remueve o elimina un nodo.

|  |
| --- |
|  |

* ¿Cuál es el truco entonces? Las listas enlazadas permiten la **inserción y eliminación eficiente, pero el acceso a los elementos puede ser ineficiente.**
* **Agregar y eliminar elementos en una determinada ubicación es eficiente en una lista enlazada.**
* **Visitar los elementos en orden secuencial es eficiente, pero el acceso aleatorio no lo es.**
* Por ejemplo, suponiendo que queremos localizar el quinto elemento, primero se deben atravesar los primeros cuatro elementos. Este es un problema si se necesita acceder a los elementos en un orden arbitrario.
* Si visitamos todos los elementos en secuencia (mostrar o imprimir los elementos) la ineficiencia del acceso aleatorio no es un problema.
* Se utilizan las listas enlazadas cuando nos preocupa la eficacia de insertar o eliminar elementos y raramente necesitaremos el acceso a elementos en orden aleatorio.

**LIST ITERATORS (Iteradores de listas)**

* Un iterador de lista se utiliza para acceder a los elementos que están dentro de una lista.
* Un iterador encapsula una posición en cualquier lugar dentro de la lista enlazada.
* Un iterador está entre dos elementos, como el cursor que está entre dos letras en un procesador de textos.

|  |
| --- |
|  |

* Cada elemento es como una letra en el procesador de textos y el iterador es como el cursor parpadeando entre dos letras.
* Para obtener un iterador, se utiliza el método **lisIterator()** de la clase LinkedList.

|  |
| --- |
| LinkedList<String> nombres = **new** LinkedList<>();  LinkedList<Empleado> empleados = **new** LinkedList<>();    ListIterator<String> iteradorNombres = nombres.listIterator();  ListIterator<Empleado> iteradorEmpleados = empleados.listIterator(); |

* Inicialmente, el iterador apunta antes del primer elemento:

|  |
| --- |
|  |

* Podemos mover la posición del iterador con el método **next()**

|  |
| --- |
| iteradorNombres.next();  iteradorEmpleados.next(); |
|  |

* El método **next()** lanza la excepción **NoSuchElementException** si hemos pasado el final de la lista.
* Siempre se debe llamar al método **hasNext()** del iterador antes de llamar al método **next()**

|  |  |
| --- | --- |
| **if**(iteradorNombres.hasNext()) { iteradorNombres.next();  } | **if**(iteradorEmpleados.hasNext()) {  iteradorEmpleados.next();  } |

* El método **next()** del iterador retorna el tipo de variable que está utilizando entre los operadores <>.

|  |  |
| --- | --- |
| **while** (iteradorNombres.hasNext()) {  String nombre = iteradorNombres.next();  // hacer algo con nombre  } | **while** (iteradorEmpleados.hasNext()) {  Empleado empleado = iteradorEmpleados.next();  // hacer algo con empleado  } |

* PARA EVITAR EL USO DE ITERADORES, se puede utilizar el ciclo **for each**.

|  |  |
| --- | --- |
| **for** (String nombre : nombres) {  // hacer algo con nombre  } | **for** (Empleado empleado : empleados) {  // hacer algo con empleado  } |

* Con esto ya no tenemos que preocuparnos por los iteradores. Detrás de la escena, este ciclo utiliza un iterador para visitar todos los elementos de una lista enlazada.
* Los nodos de la clase LinkedList almacenan dos enlaces, uno para el elemento anterior y otro para el elemento siguiente.
* El método **add()** agrega un elemento después del iterador, luego mueve la posición del iterador delante del nuevo elemento.

|  |
| --- |
|  |

* Podemos visualizar la agregación como algo parecido a escribir texto en un procesador de textos. Cada carácter es insertado después del cursor, luego el curso se mueve delante del carácter insertado.
* A continuación, se muestran algunos ejemplos que insertan Strings en una lista y luego iteran a través de la lista, agregando y removiendo elementos. Los comentarios indican la posición del iterador.

|  |
| --- |
| **package** ejemplo\_01;  **import** java.util.LinkedList;  **import** java.util.ListIterator;  **public** **class** Nombres {  **public** **static** **void** main(String[] args) {  LinkedList<String> nombres = **new** LinkedList<>();    nombres.addLast("Diana");  nombres.addLast("Harry");  nombres.addLast("Romeo");  nombres.addLast("Tom");    // | este simbolo indica la posición del iterador    ListIterator<String> iterador = nombres.listIterator(); // |DHRT      iterador.next(); // D|HRT    iterador.next(); // DH|RT    // Agregar mas elementos depues del segundo elemento    iterador.add("Juliet"); // DHJ|RT  iterador.add("Nina"); // DHJN|RT    iterador.next(); // DHJNR|T    // remover el ultimo elemento atravesado (R)    iterador.remove(); // DHJN|T    // imprimir todos los elementos  System.***out***.println(nombres);  }  } |

|  |
| --- |
| **import** java.util.LinkedList;  **import** java.util.ListIterator;  **public** **class** Letras {  **public** **static** **void** main(String[] args) {  LinkedList<String> letras = **new** LinkedList<>();  letras.add("A");  letras.add("B");  letras.add("C");  letras.add("D");  System.***out***.println(letras); // [A, B, C, D]  ListIterator<String> iterador = letras.listIterator();  iterador.next(); // A|BCD  iterador.next(); // AB|CD  iterador.remove(); // A|CD  System.***out***.println(letras); // [A, C, D]  iterador.next(); // AC|D  iterador.add("E"); // ACE|D  System.***out***.println(letras);    iterador.next(); // ACED|    iterador.add("F"); // ACED|F    System.***out***.println(letras); // ACED|F  }  } |

* El método **remove()** elimina el objeto que fue retornado por la última llamada al método **next()** o **previous().**

|  |
| --- |
| LinkedList<String> colores = **new** LinkedList<>();    colores.add("rojo");  colores.add("verde");  colores.add("azul");  colores.add("gris");    ListIterator<String> iterador = colores.listIterator(); // |RVAG    iterador.remove(); |
| * La posición del iterador está al principio. Si queremos remover algún elemento no se podrá, porque no existe ningún elemento que este atrás del iterador. Esto lanza una excepción de tipo **IllegalStateException**. |

* Se debe tener cuidado cuando se llama al método **remove().** Solo puede ser llamando una vez después de llamar al método **next()** o **previous()**, y no se puede llamar inmediatamente después de llamar al método **add().** Si llamamos a éste método de manera inapropiada, se puede lanzar una excepción de tipo **IllegalStateException.**