|  |
| --- |
| UNIVERSIDAD nACIONAL DE LA MATANZA |
| Documentación |
| “Jerarquías” |
|  |
| **GRUPO 12 – Turno Noche** |
|  |

|  |
| --- |
| Programación Avanzada (2013) |

# Conclusiones extraídas en la implementación de PilaHL/PilaCL y ColaHL/ColaCL

Estos ejercicios sirvieron para marcar la diferencia entre entre composición de clases (relación has-a) y herencia de clases (relación is-a). Es decir, mediante composición de clases se establece una relación entre clases donde una clase es parte de o contiene a otra. Este es el caso, en la vida real, de un Auto y un Motor, por citar un ejemplo. Por otro lado, en la herencia de clase se establece una relación entre clases donde una es una especialización o una generalización de la otra (dependiendo de que lado de la relación se lo mire).

En nuestro caso, al utilizar la relación de composición entre Lista y PilaCL/ColaCL, la implementación se redujo a traducir los métodos requeridos por la interfaz Pila/Cola correspondiente. Mediante manipulaciones internas de la lista contenida dentro de nuestra clase PilaCL/ColaCL, se obtienen métodos concisos y fáciles de seguir a primera vista. Para nosotros implicó menos código porque el comportamiento de PilaCL/ColaCL está limitado por la interfaz correspondiente, y existen métodos de sobra en la Lista como para respetar el comportamiento de una u otra.

Por otro lado, usando la relación de herencia entre Lista y PilaHL/ColaHL, la implementación se traduce en no solamente utilizar la estructura interna de nodos de Lista para respetar el contrato de la interfaz Pila/Cola correspondiente, sino a limitar el uso indebido de ciertos métodos que son exclusivos de Lista. Lo cual se traduce en mucho más código que las implementaciones anteriores y, por ende, mayor probabilidad de error al implementar algo sencillo como es el comportamiento de una Pila o una Cola. Por otro lado, tiene la ventaja de aportar polimorfismo, es decir, con PilaHL/ColaHL podríamos tener un método o una clase que posean una referencia a Lista, e intercambiar PilaHL/ColaHL en forma transparente, pero no tiene mucho sentido ya que son dos estructuras bien diferenciadas a la hora de elegir una u otra.

Si bien la “solución” propuesta por la cátedra es “ocultar” los métodos detrás del uso de interfaces, mediante, por ejemplo

Cola**<**T**>** cola **= new** ColaHL**<**T**>();**

para que solamente sean visibles aquéllos métodos de la interfaz Cola, y no todos los disponibles de ColaHL (que son aquéllos de la interfaz Cola más los de Lista, por ser heredada); nada le impide al programador utilizar

ColaHL**<**T**>** cola **= new** ColaHL**<**T**>();**

y en este caso, muchos métodos son incompatibles con una Cola.

En conclusión, en nuestro caso, el uso de herencia estaría sumando trabajo por realizar y no simplificando las cosas, lo cual es propenso a errores y suma mayor cantidad de horas de trabajo. En nuestra opinión Pila/Cola no son una Lista, y por ende, no tiene demasiado sentido utilizar herencia; es mucho más entendible que una Pila/Cole use una Lista o un Array (y eso queda expresado mejor por la relación de composición).

Quizá si existiese una clase/interfaz de mayor jerarquía como lo es en Java la interfaz Collections, se podría aprovechar las ventajas de ambos ejercicios (el polimorfismo y el código sencillo y al punto). Pero no es nuestro caso.

# Rendimiento de las versiones estáticas contra las dinámicas

Se nos pidió, por otro lado, analizar el rendimiento de insertar y quitar un millón de elementos en los TDA creados para comparar los tiempos y extraer conclusiones. Los valores obtenidos fueron:

|  |  |  |
| --- | --- | --- |
| Clase | Offer/Push (ms) | Poll/Pop (ms) |
| ColaEstatica | 40 | 15 |
| ColaDinamica | 239 | 18 |
| PilaEstatica | 35 | 13 |
| PilaDinamica | 419 | 15 |
| ColaCL | 211 | 20 |
| ColaHL | 208 | 20 |
| PilaCL | 207 | 21 |
| PilaHL | 211 | 20 |

De las clases analizadas, solamente ColaEstatica, PilaEstatica y ColaPrioridad utilizan internamente un array (redimensionable en tiempo de ejecución para inserción y eliminación), y el resto utilizan listas doblemente enlazadas.

El criterio de redimensión para las clases usando arrays fue: duplicar el tamaño del array cuando este llega al 100% de su capacidad (chequeo realizado en cada inserción) y dividir el tamaño del array a la mitad cuando este llega al 25% de su capacidad (chequeo realizado en cada eliminación).

Si reordenamos la tabla de acuerdo a esta información, tomando para las versiones usando una lista el valor medio de las tres implementaciones, tenemos la siguiente tabla:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TDA | Array | | Lista | |
| Método | **Offer/Push (ms)** | **Poll/Pop (ms)** | **Offer/Push (ms)** | **Poll/Pop (ms)** |
| Cola | 40 | 15 | 219 | 19 |
| Pila | 35 | 13 | 279 | 19 |

Salta a la vista que las implementaciones usando listas enlazadas son mucho más lentas durante el proceso de inserción que las mismas clases usando arrays.

La conclusión arrojada en clase fue que una lista enlazada es **siempre** mucho más lenta que una lista usando arrays (por una cuestión de enlazar nodos, recorrer la lista a diferencia de acceder por índices, etc). A primera vista, con nuestros resultados y los resultados de los demás grupos en clase, pareciera ser que es así. Sin embargo, si bien es cierto para estos casos, ¿es **siempre** cierto? ¿Siempre deberíamos elegir una implementación de una lista basada en arrays sobre aquélla implementada mediante una lista doblemente enlazada de nodos?

Para responder a esta pregunta decidimos poner a prueba las versiones dadas por el JDK de listas, es decir ArrayList vs LinkedList. Para que el análisis sea justo no se utilizaron dichas estructuras como un TDA en particular sino como meras listas secuenciales. Para que sea más justo todavía se utilizó un array aleatorio de índices al eliminar. Se compararon ambas versiones de eliminación: el método que elimina el objeto en un índice particular y el método que elimina un objeto en particular. Los resultados obtenidos fueron:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ArrayList | | LinkedList | |
| Método | **add (ms)** | **remove (ms)** | **add (ms)** | **remove (ms)** |
| Por índice | 19 | 177.575 | 22 | 1.287.972 |
| Por objeto | 306 | 435.023 | 21 | 21 |

Los resultados son los esperados: la afirmación de que una lista doblemente enlazada es **siempre** más lenta que una utilizando arrays es **falsa**.

La realidad se ajusta a lo que uno esperaría de cada estructura, si necesitásemos una lista con acceso posicional utilizaríamos un ArrayList puesto que provee esta funcionalidad y es mucho más rápida para insertar y eliminar elementos mediante su posición, que una lista doblemente enlazada como LinkedList.

Sin embargo, si no necesitásemos acceso posicional a la lista y la eliminación se reduce a eliminar un objeto en particular (más que un objeto cualquiera en una posición en particular), la lista doblemente enlazada gana con creces a la lista mediante arrays en velocidad.

La decisión final de una u otra estructura subyacente para una lista depende en última instancia del uso que se le dará a dicha lista.

# Rendimiento de Cola de Prioridad

Se nos pidió, finalmente, analizar el rendimiento de insertar y quitar un millón de elementos en la Cola de Prioridad creada para comparar los tiempos y extraer conclusiones. Los valores obtenidos fueron:

|  |  |  |
| --- | --- | --- |
| Clase | Offer/Push (ms) | Poll/Pop (ms) |
| ColaPrioridadMaxima | 114 | 1153 |
| ColaPrioridadMinima | 146 | 1142 |

Ambas fueron creadas utilizando montículos con arrays. Se esperaba un comportamiento teórico de ***O[n\*log(n)]*** tanto para la inserción como la eliminación; sin embargo, tarda 10 veces menos en la inserción que en la eliminación, comportándose linealmente en la inserción (aproximadamente 100ns por cada elemento) y comportándose en forma lineal logarítmica en la eliminación.