Proyecto #1: Colecciones en Java
Estructura de datos y algoritmos II

López,Ricardo Argüello, Dante Sánchez, Marco

1 de noviembre de 2020

1. Introducción

1.1. ¿Qué es una interfaz?

Una interface es una lista de métodos (solamente cabeceras de métodos, sin implementación) que define un comportamiento específico para un conjunto de objetos. Cualquier clase que declare que implementa una determinada interface, debe comprometerse a implementar todos y cada uno de los métodos que ese interfaz define. Esa clase, además, pasará a pertenecer a un nuevo tipo de dato extra que es el tipo de la interface que implementa. Los interfaces actúan, por tanto, como tipos de clases. Se pueden declarar variables que pertenezcan al tipo de la interface, se pueden declarar métodos cuyos argumentos sean del tipo de la interface, asimismo se pueden declarar métodos cuyo retorno sea el tipo de una interface.

Una interface es un conjunto de constantes y de métodos abstractos.

1.1.1. Clases

Cuando se define una clase, se especifica como serán los objetos de dicha clase, esto quiere decir, de que variables y de que métodos estará constituida. Las clases proporciona una especie de plantilla o molde para los objetos van a tener como atributos (características), y como acciones o métodos(acciones).

1.2. Hashing

El hashing es un concepto muy importante en computación ya que permite recuperar elementos de una colección con una complejidad temporal O(1). Para acceder a cualquier localización de una lista debemos saber su índice. Un índice sirve como la dirección de un elemento una vez que ha sido

almacenado en la lista. Para lograr acceso a un índice en tiempo O(1), debemos recordar el índice del elemento almacenado, de otra manera el elemento se tendrá que buscar, operación con complejidad temporal O(n), no O(1).

Entonces, si queremos recuperar los elementos de una lista en tiempo O(1), de alguna manera hay que recordar los índices de todos los elementos. A primera vista esto parece imposible, pero utilizando a los mismos elementos a almacenar como base para encontrar la dirección, se pueden recuperar elementos en O(1). Entonces, a través de funciones que transforman alguna característica del elemento a almacenar, se calculan las direcciones de los elementos. Este es el principio del hashing que permite que se puedan recuperar datos en O(1).

Estructuras de datos como los mapas y los conjuntos aplican este concepto para almacenar y recuperar elementos.

1.3. Collection Framework de Java

La plataforma Java incluye un marco de colecciones (Collection Framework). Un marco de colecciones es una arquitectura unificada para representar y manipular colecciones, lo que permite manipular colecciones independientemente de los detalles de implementación.

2. Colecciones

Una colección representa un grupo de objetos. Algunas colecciones permiten elementos duplicados y otras no. Algunos están ordenados y otros desordenados. El JDK no proporciona ninguna implementación directa de esta interfaz: proporciona implementaciones de subinterfaces más específicas como Set y List. Esta interfaz se utiliza normalmente para pasar colecciones y manipularlas donde se desea la máxima generalidad.

2.1. Iterable

Un Iterable es una interface que hace referencia a una colección de elementos que se puede recorrer.

La interface solo necesita que implementemos un método para poder funcionar de forma correcta, este método es iterator().

Esto significa que una clase que implementa la interfaz Java Iterable puede tener sus elementos iterados.

2.2. Interfaz Collection

La interfaz raíz en la jerarquía de colecciones.Todas las clases de implementación de Collection de propósito general (que normalmente implementan Collection indirectamente a través de una de sus subinterfaces) deben proporcionar dos constructores "estándar": un constructor vacío (sin argumentos), que crea una colección vacía, y un constructor con un solo argumento de tipo Colección, que crea una nueva colección con los mismos elementos que su argumento.

Esta interfaz es miembro de Java Collections Framework.

3. Conjuntos

3.1. Interfaz Set

Una colección que no contiene elementos duplicados. Más formalmente, los conjuntos no contienen un par de elementos e1 y e2 tales que e1.equals (e2), y como máximo un elemento nulo. Como lo implica su nombre, esta interfaz modela la abstracción de conjuntos matemáticos.

Todos los constructores deben crear un conjunto que no contenga elementos duplicados (como se definió anteriormente). No está permitido que un conjunto se contenga a sí mismo como un elemento. Algunas implementaciones de conjuntos tienen restricciones sobre los elementos que pueden contener. Por ejemplo, algunas implementaciones prohíben los elementos nulos y algunas tienen restricciones sobre los tipos de sus elementos.

Esta interfaz es miembro de Java Collections Framework.

3.1.1. Clase HashSet

Esta clase implementa la interfaz Set, respaldada por una tabla hash (en realidad, una instancia de HashMap). No ofrece ninguna garantía en cuanto al orden de iteración del conjunto; en particular, no garantiza que el pedido se mantenga constante en el tiempo. Esta clase permite el elemento nulo.

Esta clase ofrece un rendimiento de tiempo constante para las operaciones básicas (add, remove, contains and size), asumiendo que la función hash dispersa los elementos correctamente entre los depósitos. La iteración sobre este conjunto requiere un tiempo proporcional a la suma del tamaño de la instancia de HashSet (el número de elementos) más la çapacidad"de la instancia de HashMap de respaldo (el número de depósitos). Por lo tanto, es muy importante no establecer la capacidad inicial demasiado alta (o el factor de carga demasiado bajo) si el rendimiento de la iteración es importante.

Esta clase es miembro de Java Collections Framework.

3.1.2. Clase LinkedHashSet

Implementación de tabla hash y lista vinculada de la interfaz Set, con orden de iteración predecible. Esta implementación se diferencia de HashSet en que mantiene una lista doblemente enlazada que se ejecuta en todas sus entradas. Esta lista enlazada define el orden de iteración, que es el orden en el que se insertaron los elementos en el conjunto (orden de inserción). Se puede utilizar para producir una copia de un conjunto que tiene el mismo orden que el original, independientemente de la implementación del conjunto original.

Esta clase proporciona todas las operaciones Set opcionales y permite elementos nulos. Al igual que HashSet, proporciona un rendimiento en tiempo constante para las operaciones básicas (add, contains and remove), asumiendo que la función hash dispersa los elementos correctamente entre los depósitos. Es probable que el rendimiento sea ligeramente inferior al de HashSet, debido al gasto adicional de mantener la lista vinculada.

Esta clase es miembro de Java Collections Framework.

3.2. Interfaz SortedSet

Un conjunto que además proporciona un ordenamiento total de sus elementos. Los elementos se ordenan utilizando su orden natural o mediante un comparador que normalmente se proporciona en el momento de la creación del conjunto ordenado. El iterador del conjunto atravesará el conjunto en orden ascendente de elementos. Se proporcionan varias operaciones adicionales para aprovechar el pedido. (Esta interfaz es el análogo establecido de SortedMap).

Todos los elementos insertados en un conjunto ordenado deben implementar la interfaz Comparable (o ser aceptados por el comparador especificado). Además, todos estos elementos deben ser mutuamente comparables.

Todas las clases de implementación de conjuntos ordenados de propósito general deben proporcionar cuatro constructores .estándar": 1) Un constructor vacío (sin argumentos), que crea un conjunto ordenado vacío ordenado según el orden natural de sus elementos. 2) Un constructor con un solo argumento de tipo Comparator, que crea un conjunto ordenado vacío ordenado según el comparador especificado. 3) Un constructor con un solo argumento de tipo Colección, que crea un nuevo conjunto ordenado con los mismos elementos que su argumento, ordenados según el orden natural de

los elementos. 4) Un constructor con un solo argumento de tipo SortedSet, que crea un nuevo conjunto ordenado con los mismos elementos y el mismo orden que el conjunto ordenado de entrada.

Esta interfaz es miembro de Java Collections Framework.

4. Listas

4.1. Interfaz List

La interfaz List, también conocida como "secuencia" es la implementación de la estructura de datos List, y normalmente acepta elementos repetidos o duplicados según sus implementaciones, y donde cada uno de los elementos se encuentra indexado, el primer elemento se encuentra en la posición cero. Hereda todos los métodos de la interfaz Collection.

Esta interfaz proporciona uso un iterador especial llamada ListIterator que extiende a la interfaz Iterator. La interfaz List incluye métodos para acceder posicionalmente a los elementos (por medio del índice), para realizar búsqueda secuencial, y hacer u obtener operaciones con sublistas, En Java existen dos implementaciones muy importantes de la interfaz List: ArrayList y LinkedList.

4.1.1. Clase ArrayList

ArrayList es una de las implementaciones más usadas de la Interfaz List, pues representa el uso de un Arreglo Dinámico que puede modificar su tamaño en tiempo de ejecución. Esta clase incluye métodos para manipular el tamaño del arreglo que internamente contiene los elementos de una List: size, isEmpty, get, set, add, Iterator y ListIterator.

Cada objeto de tipo ArrayList tiene una capacidad, que es el tamaño del arreglo usado para almacenar los elementos de la lista, siempre de que se agregue un nuevo elemento se aumentará la capacidad automáticamente. Es importante saber que esta implementación no está sincronizada, por lo que no es seguro manipular un ArrayList con varios subprocesos a la vez a menos que se sincronice de manera externa.

Esta implementación es preferida ante un Array convencional dado su dinamismo, sin embargo, la clase Vector es equivalente a un ArrayList, excepto que el Vector sí es seguro para manejar hilos.

4.1.2. Clase LinkedList

La clase LinkedList representa una implementación de la estructura de datos lineal lista ligada, donde cada uno de los nodos contiene una referencia a su nodo contiguo, esto no significa que estén contiguos físicamente en memoria, pueden estar separados pero enlazados por medio de referencias. Esta es otra de las implementaciones más usadas de la Interfaz List, sin embargo, tienen algunas desventajas con respecto a ArrayList, pues LinkedList toma mayor cantidad de tiempo para encontrar un elemento ya que debe recorrer todos siguiendo la secuencia indicada por las referencias hasta hallar el valor buscado, y ArrayList sólo necesita saber el índice del elemento y accede directamente a él.

La clase LinkedList contiene todos los métodos que tiene un ArrayList, pero su principal diferencia radica en cómo están construidas: Mientras que ArrayList en esencia tiene un arreglo que contiene sus elementos, un LinkedList almacena sus elementos en nodos referenciados. Los nuevos métodos que contiene son: addFirst, addLast, removeFirst, removeLast, getFirst y getLast.

4.1.3. Clase Vector

Un Vector es en esencia igual a un ArrayList, pues ambas implementaciones tienen un Array internamente como estructura de datos, ambos pueden modificar el tamaño del arreglo de manera dinámica, pero su diferencia es cómo modifican su capacidad. Por defecto un Vector duplica el tamaño de su arreglo cuando un elemento se agrega, mientras que ArrayList aumenta el tamaño agregando la mitad de su tamaño actual por cada elemento ingresado. Además de que Vector fue incluido en la primera versión del JDK, mientras que ArrayList fue incluido en la versión 1.2.

Pero su principal diferencia es que un Vector es sincronizado, esto implica que puede ser modificado seguramente por varios subprocesos, mientras que ArrayList no. Que un Vector sea sincronizado significa un mayor coste de recursos, y sólo lo debemos usar cuando tengamos que hacer operaciones "thread-safe".

4.1.4. Clase Stack

La clase Stack extiende a la clase Vector para implementar una estructura de datos de tipo pila. Una pila es una estructura de tipo LIFO, donde el último elemento ingresado es el primero que sale, e incluye los métodos push (apilar), pop (desapilar), peek (copiar el elemento de la cima), search (busca

la posición de un elemento) y empty (verificar si es vacía). Dado que incluye todos los métodos de Vector, podemos usar el método add para añadir en cualquier posición, sin embargo, no es recomendable puesto que se perdería la naturaleza de la Pila.

Un conjunto más completo y consistente para operaciones de Pila es el de la interfaz Deque y sus implementaciones.

5. Colas

5.1. Interfaz Queue

Es una colección diseñada para almacenar elementos antes de ser procesados. Esta estructura generalmente pero no necesariamente ordena los elementos de manera FIFO (first-in-first-out). Una excepción de esta esto son las colas LIFO o stacks. La mayor parte de las implementaciones no tienen un limite de elementos, pero algunas otras sí.

Las implementaciones de queue generalmente no permiten elementos nulos ya que muchos métodos devuelven null al realizar sus operaciones.

Algunos de los métodos más importantes de esta interfaz son:

- Add y offer permiten añadir elementos a una cola si es posible. El primero devuelve una excepción si no hay espacio disponible y el segundo devuelve verdadero o falso dependiendo de si se pudo añadir un elemento o no.
- Peek y element devuelven, pero no remueven, el primero elemento de una cola. El primero devuelve null si la cola está vacía y el segundo devuelve una excepción.
- Poll y remove devuelven y eliminan al primer elemento de una cola. El primero devuelve null si la cola está vacía y el segundo genera una excepción.

5.1.1. Clase priorityQueue

Esta es una implementación de una cola de prioridad basada en un heap de prioridad. En esta estructura, los elementos están ordenados de acuerdo con su orden natural o por un comparador provisto durante la creación de la estructura. Esta estructura no permite la inserción de elementos no comparables si se esta ordenando a los elementos de acuerdo con su orden natural.

Esta implementación no está sincronizada, lo que significa que diferentes procesos pueden acceder a ella al mismo tiempo.

Además, esta clase ofrece complejidad O(log(n)) para las operaciones de offer, add, poll y remove ya explicadas en la interfaz Queue; complejidad lineal para las operaciones de remove(Object) y contains(Object) y tiempo constante para las operaciones de peek, element, y size.

5.2. Deque

Es una interfaz que representa una cola doblemente enlazada, que es una estructura de datos lineal que permite insertar y eliminar elementos por ambos extremos, que engloba las propiedades LIFO y FIFO en una misma estructura. Las clases ArrayDeque y LinkedList implementan a esta interfaz.

5.2.1. ArrayDeque

La clase ArrayDeque implementa dos interfaces, la interfaz Queue y Deque y permite implementar la estructura de datos Cola Doblemente Enlazada, pero aplicado sobre un Array, no funciona como las LinkedList por medio de nodos enlazados, pues aquí todos los nodos son contiguos en memoria. Un ArrayDeque tiene las siguientes consideraciones:

- No hay restricciones de capacidad, excepto por la memoria del sistema donde se ejecutan.
- No son sincronizadas, esto significa que no son seguras para ser manipuladas por varios subprocesos
- No se permite el uso de elementos Null

ArrayDeque es más consistente y rápida que un Stack cuando se usa como pila, de igual forma es mejor que LinkedList cuando se utiliza como cola.

6. Mapas

Los mapas en el contexto de la programación no tienen que ver con lo primero que se nos ocurre al escuchar su nombre. El nombre refiere a un concepto más matemático que se refiere a una función que mapea un dominio y un recorrido. Los mapas son conocidos por muchos nombres,

entre los que se encuentran diccionario, tabla hash y mapas hash. Todos los nombres anteriores refieren a la misma estructura de datos.

Un mapa mapea o relaciona un conjunto de claves con sus valores asociados de una manera muy parecida a como una función matemática mapea un valor en el dominio al rango. La clave es lo que se debe proporcionar al mapa para buscar un par llave/valor. Las claves en un mapa deben ser únicas, lo que significa que solo puede haber una copia de cada clave en un mapa al mismo tiempo.

Los mapas de cierta manera son parecidos a los conjuntos (set), en el sentido de que el mapa contiene un conjunto de llaves únicas. Como en los conjuntos, se pude buscar una clave, y por lo tanto su valor asociado, en un tiempo cercano a O(1). Esto es debido a que los mapas al igual que los sets utilizan una función hash para almacenar los elementos.

Los mapas, a diferencia de los conjuntos, no almacenan únicamente el valor de clave, sino que el valor asociado también está dentro de la estructura de datos.

Los mapas son especialmente útiles en dos situaciones, cuando se requieren memorizar y cuando se requiere asociar datos. Un ejemplo muy sencillo de primer caso es utilizar un mapa para mejorar la versión recursiva de la sucesión de Fibonacci. La versión original es muy ineficiente ya que requiere calcular múltiples veces la sucesión de Fibonacci de los números. Si se implementa un mapa en el que las claves sean los números y los valores el resultado de calcular la serie de Fibonacci del número, se puede mejorar considerablemente el rendimiento del algoritmo.

Un ejemplo de segundo caso es una relación de ciudades y códigos postales. Si se tiene un programa en el que se debe buscar o almacenar el código postal de diferentes ciudades, se puede crear un mapa en el que las claves sean el nombre de las ciudades y el valor el código postal. De esa manera al ingresar el nombre de una ciudad se podría obtener su código postal en tiempo constante.

6.1. Interfaz Map

Una interfaz es un conjunto de métodos relacionados y vacíos que las clases pueden implementar. Cuando una clase implementa un método, está obligada a implementar los métodos de la interfaz. Las interfaces sirven para formalizar como un objeto interactuará con el exterior y para saber el comportamiento esperado del mismo.

La interfaz Map define objetos que mapean claves y valores. Esta interfaz ofrece tres maneras de verla: como un conjunto de claves, como una colección

de valores o como un conjunto de relaciones clave- valor.

El comportamiento de un mapa no está especificado en caso de que se utilice un objeto mutable como clave o que se realice cualquier operación que afecte al resultado del método equals del objeto. Por esta razón, no está permitido que un mapa se contenga como clave a sí mismo.

En general, las clases que implementan la interfaz equals deben tener dos constructores, uno vacío que genera un mapa vacío u otro que recibe un argumento de tipo Map que genera un mapa con las mismas relaciones clave-valor.

Algunos de los métodos más importantes de los mapas son:

- clear, que elimina todas las relaciones del mapa.
- get, que devuelve el valor asociado a una clave.
- put, cuya función es asociar un valor especificado con una llave del mapa.

6.2. Interfaz SortedMap

Esta interfaz provee mapas que están ordenados. El ordenamiento sigue el orden natural de sus claves o a un Comparator suministrado al momento de creación del mapa. Las llaves deben implementar la interfaz comparable y deben ser consistente en el método equals.

Cualquier clase que implemente la interfaz SortedMap debe tener al menos cuatro constructores estándar. Dos de ellos son los que solicita la interfaz Map (que es extendida por esta interfaz) y otros dos son propios de esta interfaz. Uno debe recibir un solo argumento de tipo comparator, que crea un mapa ordenado de acuerdo con el comparador especificado vacío y otro con un argumento de tipo SortedMap, que genera un mapa ordenado con el mismo orden, elementos y comparador del mapa suministrado.

Algunos métodos importantes son:

- firstKey, que devuelve la primera llave del mapa. Hay otro método parecido llamado lastKey que devuelve la última llave.
- subMap, que devuelve un mapa ordenado desde la primera llave suministrada de manera abierta, hasta la segunda suministrada de manera cerrada.

- Values, que devuelve una colección de los valores contenidos en el mapa.

6.3. Interfaz navigableMap

Esta es una interfaz que extiende a SortedMap con métodos de navegación que devuelven al elemento más cercano siguiendo un determinado criterio de búsqueda. Por ejemplo, está el método lowerEntry que devuelve la clave más cercana menor a una clave dada. Los métodos de este mapa deben ser utilizados solo para localizar otros elementos, ya que no están diseñados para permitir modificar las entradas.

Algunos métodos tienen nombre y funcionalidad parecidos a los de la interfaz sortedMap, pero difieren en que permiten elegir si las fronteras o criterios de búsqueda son inclusivos o no. Un ejemplo es el método subMap. Los submapas de un mapa navegable deben implementar la interfaz navigableMap. Los métodos que devuelven entradas no devuelven el valor almacenado, sino que devuelven pares Map.Entry que son una copia de una relación dentro del mapa en el momento de la consulta. Estás entradas deben ser utilizadas para consulta únicamente, ya que las implementaciones no suelen permitir su modificación.

Algunos métodos importantes son:

- pollFirtEntry y pollLastEntry devuelven el mapeo de la primero y última clave respectivamente, y lo eliminan del mapa.
- lowerEntry y higherEntry devuelven el valor más grande estrictamente menor y el valor más pequeño estrictamente mayor a la clave dada respectivamente.
- floorEntry y ceilingEntry hacen lo mismo que los métodos mencionados anteriormente, solo que incluyen a valores iguales a la clave suministrada.

6.3.1. TreeMap

Esta es una implementación de la interfaz NavigableMap que utiliza un árbol rojo-negro. Las claves son ordenadas por su orden natural o por un comparador proporcionado al mapa al momento de creación.

Gracias al árbol rojo-negro, esta estructura puede garantizar tiempo O(log (n)) en las operaciones de búsqueda de una clave, obtención de un valor o clave y de eliminación. Esta implementación no está sincronizada, lo que significa que varios procesos pueden acceder al objeto al mismo tiempo.

Como esta es una implementación de la interfaz NavigableMap, sus métodos son los mismo a los de esa interfaz.