Estructura de datos convencionales de Java

Montecinos Gómez Juan Pablo
Facultad de Ciencias y Tecnología, Universidad Mayor de San Simón
Licenciatura en Ingeniería Informática
barkion00@gmail.com
Cochabamba-Bolivia

Resumen – El documento contiene un análisis e interpretación de las estructuras de datos que cuenta internamente el lenguaje de programación java y como interactúa el almacenamiento de datos.

I. INTRODUCCIÓN

Java collections es una colección que representa un grupo de objetos. Estos objetos son conocidos como elementos.

El lenguaje java se emplea el concepto de interfaz genérica Collection para el almacén de elementos. Esta interfaz nos ayuda a almacenar cualquier tipo de objeto y podemos usar distintos métodos sobre estos como ser: insertar, eliminar, modificar, la longitud, etc.

Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

II. ESTRUCTURAS DE DATOS

A. Vector

Este concepto no se encuentra en Collection de java, pero es una estructura básica, los vectores se asignan dinámicamente. No se declara que contengan un tipo de variable; en cambio, cada Vector contiene una lista dinámica de referencias a otros objetos. La clase Vector se encuentra en el paquete java.util y extiende java.util.Abstractlist.

La gran ventaja de usar Vectores es que el tamaño del vector puede cambiar según sea necesario. Los vectores manejan estos cambios a través de los campos "capacidad" y "capacidad Incremento".

Cuando se crea una instancia de un Vector, declara una matriz de objetos de tamaño initialCapacity. Cada vez que se llena esta matriz, el vector aumenta el tamaño total del búfer en el valor valueIncrement. Por lo tanto, un Vector representa un compromiso. No asigna cada objeto dinámicamente, lo que haría que el acceso a los elementos de la lista central fuera extremadamente lento; pero sí permite el crecimiento en el tamaño de la lista. El constructor predeterminado, Vector (), crea un Vector vacío de capacidad inicial cero que duplica su tamaño cada vez que se llena.

TABLE I
METODOS DE LA CLASE VECTOR

Metodo y tipo	Descripcion
boolean add (E e)	Agrega un elemento al final del vector,
void add (int index, elemento E)	Agrega un elemento en el índice especificado y mueve los elementos completos un paso hacia adelante
void addElement(E obj)	Agrega el elemento especificado al final del vector y también aumenta su tamaño en 1
int capacity()	Proporciona la capacidad del vector,
void clear()	Elimina todos los elementos en el vector
clone clone()	Proporciona una copia duplicada de todo el vector
boolean contains(Object o)	Indica si el vector contiene el elemento especificado en el vector, devolverá verdadero si ese elemento está presente o falso si no está presente
boolean containsAll(Collection c)	Retorna verdadero si todos los elementos de una colección están presentes en la colección de llamada, o falso si no

B. Array

Al igual que vector este no pertenece a Collection de java, pero es una estructura básica, Los arrays se pueden definir como objetos en los que podemos guardar más de una variable, es decir, al tener un único arreglo, este puede

guardar múltiples variables de acuerdo a su tamaño o capacidad, esta estructura soporta variables de un solo tipo. También se dividen en arreglos unidimensionales y multidimensionales.

Existen 2 tipos de arreglos, los unidimensionales, y los multidimensionales (generalmente 2 dimensiones y se les denomina matrices).

TABLE II
MÉTODOS DE LA CLASE ARRAY

Metodo y tipo	Descripcion
.length	Retorna la dimensión del array
binarySearch	Compruebe si una matriz contiene un cierto valor
.clone()	Se realiza una "copia profunda" con el nuevo array que contiene copias de los elementos del array original en lugar de referencias
.contains()	Verifica si existe e
	Concatena dos arrays
reverse(Array);	Invierte un array

C. Pila

La pila es una secuencia de elementos un mismo tipo en la que el acceso a la misma se realiza por el cabezal de la esta.

Una Pila tiene una filosofía de entrada y salida de datos, esta filosofía es la LIFO (Last In First Out, en español, ultimo en entrar, primero en salir).

TABLE III
METODOS DE LA CLASE PILA

Metodo y tipo	Descripcion
boolean empty()	Verifica si esta pila está vacía.
E peek()	Mira el objeto en la parte superior de esta pila sin quitarlo de la pila.
E pop()	elimina el objeto en la parte superior de esta pila y devuelve ese objeto como el valor de esta función.
E push(E item):	Inserta e empuja un elemento sobre la parte superior de esta pila.
int search(Object o)	devuelve la posición basada en 1 donde un objeto está en esta pila.

D. Cola

Una cola es una estructura de datos similar a la pila, donde los elementos se insertan una detrás de otra y para extraer siempre se lo hace por adelante de la cola donde se encuentra el primer elemento.

TABLE IV
METODOS DE LA CLASE COLA

Metodo y tipo	Descripcion
add()	Inserta el elemento especificado en la cola. Si la tarea es exitosa, add()regresa true, si no, arroja una excepción.
offer()	Inserta el elemento especificado en la cola. Si la tarea es exitosa, offer()regresa true, si no regresa false.
element()	devuelve el encabezado de la cola. Lanza una excepción si la cola está vacía.
peek()	Devuelve el encabezado de la cola. Devuelve null si la cola está vacía.
remove()	devuelve y elimina el encabezado de la cola. Lanza una excepción si la cola está vacía.
poll()	Devuelve y elimina el encabezado de la cola. Devuelve nullsi la cola está vacía.

E. Set

La interfaz Set define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode.

Dentro de la interfaz Set existen varios tipos de implementaciones realizadas dentro de la plataforma Java entre ellas tenemos:

1) HashSet:

Esta implementación almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
Es mejor que un Linkendlist

2) TreeSet:

Esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que HashSet. Los elementos almacenados deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

3) LinkedHashSet:

Esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que HashSet.

TABLE V
METODOS DE LA CLASE SET

Metodo y tipo	Descripcion
boolean add(E e)	Agrega el elemento especificado a este conjunto si aún no está presente (operación opcional). boolean
Boolean addAll(Collection<? extends E> c)	Agrega todos los elementos de la colección especificada a este conjunto si aún no están presentes (operación opcional).
void clear()	Elimina todos los elementos de este conjunto (operación opcional).
boolean contains(Object o)	Devuelve verdadero si este conjunto contiene el elemento especificado.
Boolean containsAll(Collection<?> c)	Devuelve verdadero si este conjunto contiene todos los elementos de la colección especificada.
boolean equals(Object o)	Compara el objeto especificado con este conjunto para la igualdad.
Int hashCode()	Devuelve el valor del código hash para este conjunto.
boolean isEmpty()	Devuelve verdadero si este conjunto no contiene elementos.
Iterator<E>iterator()	Devuelve un iterador sobre los elementos de este conjunto.
boolean remove(Object o)	Elimina el elemento especificado de este conjunto si está presente (operación opcional).
boolean removeAll(Collection<?> c)	Elimina de este conjunto todos sus elementos contenidos en la colección especificada (operación opcional).
Boolean retainAll(Collection<?> c)	Retiene solo los elementos de este conjunto que están contenidos en la colección especificada (operación opcional).
int size()	Devuelve el número de elementos en este conjunto (su cardinalidad).
Object[] toArray()	Devuelve una matriz que contiene todos los elementos de este conjunto.
<T> T[] toArray(T[] a)	Devuelve una matriz que contiene todos los elementos de este conjunto; El tipo de tiempo de ejecución de la matriz devuelta es el de la matriz especificada.

F. List

La interfaz List define una sucesión de elementos. A diferencia de la interfaz Set, la interfaz List sí admite elementos duplicados. A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
Iteración sobre elementos: mejora el Iterator por defecto.
Rango de operación: permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.
Dentro de la interfaz List existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

1) ArrayList:

Se basa en un array redimensionable que aumenta su tamaño según va insertándose elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.

2) LinkedList:

Esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento

TABLE VI
METODOS DE LA CLASE LIST

Metodo y tipo	Descripcion
int size():	Obtiene el número de elementos en la lista.
boolean isEmpty()	Verificar si la lista está vacía o no.

boolean contains(Object o)	Devuelve verdadero si esta lista contiene el elemento especificado.
Iterator<E> iterator()	devuelve un iterador sobre los elementos de esta lista en la secuencia adecuada.
Object[] toArray()	devuelve una matriz que contiene todos los elementos de esta lista en la secuencia adecuada
boolean add(E e)	agrega el elemento especificado al final de esta lista.
boolean remove(Object o)	elimina la primera aparición del elemento especificado de esta lista.
boolean retainAll(Collection<?> c)	retiene solo los elementos de esta lista que están contenidos en la colección especificada.
void clear()	elimina todos los elementos de la lista.
E get(int index)	devuelve el elemento en la posición especificada en la lista.
E set(int index, E element)	reemplaza el elemento en la posición especificada en la lista con el elemento especificado.
ListIterator<E> listIterator()	Devuelve un iterador de lista sobre los elementos en la lista.
List<E> subList(int fromIndex, int toIndex)	devuelve una vista de la parte de esta lista entre el fromIndex especificado, inclusive, y toIndex, exclusivo. La lista devuelta está respaldada por esta lista, por lo que los cambios no estructurales en la lista devuelta se reflejan en esta lista, y viceversa.

G. Map

La interfaz Map asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.

1) 1hashMap:

Esta implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

2) TreeMap

Esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que HashMap. Las claves almacenadas deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

3) LinkedHashMap

Esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.

TABLE VII
METODOS DE LA CLASE LIST

Metodo y tipo	Descripcion
put(Object key, Object value)	este método se utiliza para insertar el mapa especificado en este mapa.
remove(Object key)	este método se utiliza para eliminar una entrada para la clave especificada.
boolean containsKey(Object key)	este método se utiliza para buscar la clave especificada en este mapa.
void putAll(Map map):	este método se utiliza para insertar el mapa especificado en este mapa.
get(Object key):	este método se utiliza para devolver el valor de la clave especificada.
Set keySet():	este método se utiliza para devolver la vista Set que contiene todas las claves.
Set entrySet():	este método se utiliza para devolver la vista Set que contiene todas las claves y valores.

III. CONCLUSIÓN

Las estructuras de datos de Collection java, nos ayudan de gran manera en simplificar muchas tares en cuanto a complejidad y costo. Es importante mencionar que no todas estas son usadas esto por un problema de decisión que se debe basar en un diagrama de decisión para el uso de colecciones java Fig. 1

Fig.1 diagrama de desiciones para uso de colecciones en java

IV. REFERENCES

[1] *Algoritmos y Estructuras de Datos*. (s. f.). Centro de Investigación de Métodos Computacionales. Recuperado 7 de octubre de 2020, de https://www.academia.edu/4653959/ESTRUCTURA_DE_DATOS_EN_JAVA

[2] Joyanes Aguilar, L. (s. f.). *Estructura de datos en java*. ACADEMIA. Recuperado 7 de octubre de 2020, de https://www.academia.edu/4653959/ESTRUCTURA_DE_DATOS_EN_JAVA

[3] *Estructura de datos*. (s. f.). Centro de computo. Recuperado 7 de octubre de 2020, de https://www.cec.uchile.cl/~luvasque/edo/java/manuales/Estructuras%20de%20Datos%20en%20Lenguaje%20Java%20(CCG).pdf

[4] Oracle. (s. f.). *Https://docs.oracle.com/javase/8/docs/api/*. Documentation. Recuperado 7 de octubre de 2020, de https://docs.oracle.com/javase/8/docs/api/