

Apéndice 1

Cómo implementa Java las estructuras de datos vistas en la materia

Java tiene sus propias implementaciones de algunas de las estructuras vistas en esta materia. A continuación se muestran algunas de estas implementaciones y sus métodos básicos. Sin embargo, Java ofrece más implementaciones y métodos que los listados a continuación. Para saber más se puede acceder a la documentación en el sitio web de Oracle¹.

1.1. Operaciones comunes a todas las estructuras Java

Las operaciones listadas a continuación son heredadas de la clase `Collection`, por lo tanto son comunes en todas las implementaciones a continuación. Algunas implementaciones heredan los métodos `equals` y `toString` de super-clases abstractas, por lo tanto se debe chequear en la documentación de cada clase.

Clase <code>Collection</code> y sus sub-clases				
Signatura	Resultado	Descripción Oracle	Orden	Equivale a
<code>clear()</code>	<code>void</code>	Quita todos los elementos de la estructura	$O(n)$	vaciar
<code>clone()</code>	<code>Object</code>	Retorna una copia superficial de la estructura (Copia no profunda, a nivel de estructura. No duplica los elementos)	$O(n)$	clonar
<code>isEmpty()</code>	<code>boolean</code>	Retorna verdadero si la estructura no tiene elementos	$O(1)$	<code>esVacia</code>

1.2. TDAs Lineales

Los TDAs lineales pueden encontrarse en Java implementados para permitir y no permitir acceso concurrente, y también de forma dinámica y estática. A continuación se presenta un resumen de las operaciones básicas de la clase `LinkedList`, que puede ser utilizada en lugar de los TDA Lista, Pila y Cola vistos en la materia, y al final se presentarán alternativas que permiten acceso concurrente.

1.2.1. Lista

El TDA Lista está implementado en Java de distintas maneras, cuyo nombre da un indicio sobre su implementación. `LinkedList` corresponde a la implementación dinámica, similar a la vista en la materia, pero doblemente enlazada y con un atributo que le permite obtener la longitud (*size*) en orden $O(1)$. La primera posición de la lista se considera posición 0.

Existe otra implementación, `ArrayList`, que provee las mismas operaciones pero con implementación estática (sobre un array). La clase `Vector` de Java también implementa el TDA lista de forma estática, y se diferencia de `ArrayList` en que permite acceso sincronizado.

Todas las implementaciones permiten usar elemento de tipo `Object` o parametrizar al tipo `E`. En la tabla a continuación se listan los métodos básicos. Otros métodos se pueden encontrar en la página de documentación de Oracle².

¹<https://docs.oracle.com/javase/7/docs/api/allclasses-noframe.html>

²Disponible en <https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>

TDA Lista con Clase LinkedList o ArrayList				
Signatura	Return	Descripción Oracle	Orden	Equivale a
size()	int	Retorna la cantidad de elementos de la lista	O(1)	longitud
add (int index, E element)	void	Inserta el elemento dado en la posición index. Si la posición no es válida se produce <code>IndexOutOfBoundsException</code>	O(n)	insertar en pos dada
addFirst(E e)	void	Inserta el elemento al principio de la lista (posición 0)	O(1)	insertar en pos 1
addLast(E e)	void	Agrega el elemento en la última posición	O(1)	insertar en longitud() + 1
get(int index)	E	Retorna el elemento de la posición dada	O(n)	recuperar
getFirst()	E	Retorna el primer elemento de la lista (posición 0)	O(1)	recuperar de pos 1
getLast()	E	Retorna el último elemento de la lista	O(1)	recuperar de longitud()
indexOf(Object o)	int	Retorna la posición del elemento dentro de la lista. Si no está retorna -1	O(n)	localizar
remove(int index)	E	Elimina el elemento de la posición dada, y también lo retorna	O(n)	combina eliminar y recuperar
removeFirst()	E	Elimina el elemento de la primera posición (0)	O(1)	eliminar en pos 1
removeLast()	E	Elimina el elemento de la última posición	O(1)	eliminar en pos longitud()
remove(Object o)	boolean	Busca y elimina la primera aparición del elemento pasado por parámetro	O(n)	combina localizar y eliminar
set(int index, E element)	E	Reemplaza el elemento que está en la posición index por un nuevo elemento	O(n)	no se vio en la materia pero puede ser útil
contains(Object o)	boolean	Retorna true si el elemento está en la lista (pertenece)	O(n)	localizar(o) < 0

1.2.2. Cola

Para usar Cola sin permitir acceso concurrente, la documentación Oracle recomienda utilizar la clase *LinkedList*. En esta implementación se supone que el frente está en la posición 0. Los métodos básicos del TDA visto en la materia, por cuáles se pueden reemplazar al usar esta estructura son:

TDA Cola con Clase LinkedList				
Signatura	Return	Descripción Oracle	Orden	Equivale a
peek()	E	Retorna el elemento del frente = posición 0. A diferencia de <code>poll()</code> , no lo remueve	O(1)	obtener frente
add(E e)	boolean	Agrega el elemento recibido por parámetro al final de la lista	O(1)	poner
poll()	E	Elimina y retorna el elemento en el frente = posición 0	O(1)	combina sacar + obtener frente

1.2.3. Pila

De forma similar a Cola, los desarrolladores de Java recomiendan utilizar la clase *LinkedList* para utilizar Pila sin acceso concurrente.

TDA Pila con Clase LinkedList				
Signatura	Return	Descripción Oracle	Orden	Equivale a
peek()	E	Retorna el elemento del tope A diferencia de <code>pop()</code> , no lo remueve	O(1)	obtener tope
push(E item)	E	Pone el elemento en el tope de la pila	O(1)	apilar
pop()	E	Elimina y a la vez retorna el elemento del tope	O(1)	combina desapilar + obtener tope

1.2.4. Cola de doble entrada-salida

La Cola de doble entrada-salida, es una estructura lineal que puede resultar muy útil en algunos contextos. Su comportamiento es una mezcla entre Pila y Cola. Se puede imaginar como un tubo con una abertura en cada extremo que, a diferencia de la cola común (que los elementos ingresan por un extremo y salen por el otro), en la cola de doble entrada-salida se pueden mover en ambos sentidos, es decir que pueden entrar y salir por cualquiera de los dos extremos (nunca por el medio). Para usarla, se puede usar *LinkedList* con los métodos que agregan, quitan y obtienen cada uno de los extremos, como se muestra en la tabla a continuación:

TDA Cola de doble entrada-salida con Clase <i>LinkedList</i>				
Signatura	Return	Descripción Oracle	Orden	Equivale a
<code>addFirst(E e)</code>	void	Pone el elemento en la primera posición	$O(1)$	poner por izquierda
<code>addLast(E e)</code>	void	Pone el elemento en la última posición	$O(1)$	poner por derecha
<code>getFirst()</code>	E	Retorna el primer elemento	$O(1)$	obtener frente izquierda
<code>getLast()</code>	E	Retorna el último elemento	$O(1)$	obtener frente derecha
<code>removeFirst()</code>	E	Remueve y retorna el primer elemento	$O(1)$	quitar por izquierda
<code>removeLast()</code>	E	Remueve y retorna el ultimo elemento	$O(1)$	quitar por derecha

1.3. TDA Conjunto

Como se vio en el material de la Unidad de TDAs de uso específico, el TDA Conjunto funciona de manera análoga a un conjunto matemático, con operaciones para insertar, preguntar el elemento pertenece al conjunto y eliminarlo. A diferencia de Lista, no se debe indicar posición ni orden dentro del conjunto.

1.3.1. TreeSet

Es un conjunto ordenado implementado con árbol auto-balanceado Red-Black - equivalente a un árbol B de orden 4 - sin elementos repetidos.

Respecto a las estructuras vistas en la materia, se puede pensar en un comportamiento equivalente a un árbol binario de búsqueda AVL, pero de mejor eficiencia porque el árbol es más chato (menos niveles). Para poder usarlo, el tipo de elemento a almacenar debe ser Comparable, igual que se vio para AVL.

TDA Conjunto con Clase <i>TreeSet</i> (para elementos Comparable)				
Signatura	Return	Descripción Oracle	Orden	Equivale a
<code>add(E e)</code>	boolean	Si el elemento no está en el conjunto lo agrega	$O(\log n)$	Insertar
<code>remove(Object o)</code>	boolean	Si el elemento estaba en el conjunto, lo quita	$O(\log n)$	Eliminar
<code>contains(Object o)</code>	boolean	Retorna true si el elemento está en el conjunto	$O(\log n)$	Existe o Pertenece
<code>first()</code>	E	Recupera el primer elemento (menor) del conjunto	$O(\log n)$	Recuperar elem mínimo
<code>last()</code>	E	Retorna el último elemento (mayor) del conjunto	$O(\log n)$	Recuperar elem máximo
<code>ceiling(E e)</code>	E	Retorna el elemento más pequeño que sea mayor o igual al pasado por parámetro. Si no existe retorna null.		
<code>floor(E e)</code>	E	Retorna el elemento más grande que sea menor o igual al pasado por parámetro. Si no existe retorna null.		
<code>higher(E e)</code>	E	Retorna el elemento más pequeño que sea estrictamente mayor al pasado por parámetro. Si no existe retorna null.		
<code>lower(E e)</code>	E	Retorna el elemento estrictamente más pequeño que sea mayor que el pasado por parámetro. Si no existe retorna null.		

Ejemplo: Sea la lista $\langle 1, 3, 5, 7, 9, 11 \rangle$

- `first()` retorna 1
- `last()` retorna 11
- `ceiling(6)` retorna 7
- `floor(4)` retorna 3

- `higher(5)` retorna 7 y `ceiling(5)` retorna 5, la diferencia es que `higher` busca estrictamente mayor, y `ceiling` mayor o igual
- `lower(5)` retorna 3 y `floor(5)` retorna 5 (por motivo similar a la diferencia entre `higher` y `ceiling`)

1.3.2. HashSet

Implementa un conjunto no ordenado, sin elementos repetidos. Basado en estrategia de hashing según lo visto en la unidad de Tabla Hash. *HashSet* está implementado con la estrategia de hash abierto. Para poder utilizarlo, el elemento de tipo *E* debe tener implementado eficientemente el método *hashCode()*, tal cual está especificado para *Object*. En la siguiente tabla se puede ver que al no ser ordenado, no tiene operaciones como *ceiling*, *floor*, etc. que sí tiene *TreeSet*.

TDA Conjunto con Clase HashSet (para elementos Object con hashCode)				
Signatura	Return	Descripción Oracle	Orden	Equivale a
<code>add(E e)</code>	boolean	Si el elemento no está en el conjunto lo agrega	$O(1)$ *	insertar
<code>contains(Object o)</code>	boolean	Retorna true si el elemento está en el conjunto	$O(1)$ *	existe
<code>remove(Object o)</code>	boolean	Si el elemento estaba en el conjunto, lo quita	$O(1)$ *	eliminar

(*) La eficiencia de los métodos es orden $O(1)$ siempre y cuando la función *hashCode* tenga una distribución uniforme sobre la tabla y la longitud promedio de las listas de cada bucket sea no mayor a 3.

1.4. Cola de prioridad

La clase que implementa cola de prioridad en Java se llama *PriorityQueue*. Si bien la documentación de Oracle no especifica si la implementación es un árbol Heap binario, indica que la eficiencia de las operaciones de inserción y eliminación son de $O(\log n)$; acceder al elemento del frente (*peek*) o saber la cantidad de elementos en la estructura (*size*) es $O(1)$ y eliminar o preguntar si un elemento cualquiera existe en la cola, la operación es de $O(n)$. A partir de estos indicios se puede imaginar que la estructura interna es similar al comportamiento esperado en un árbol Heap, más un atributo numérico que guarda la cantidad de elementos.

El orden que considera entre los elementos es el orden natural, por lo tanto los elementos deben cumplir la interfaz *Comparable*. A diferencia de lo visto en la materia, la comparación se hace solamente por el elemento (no ingresa la prioridad por separado), por lo tanto la prioridad debe ser parte del elemento a ingresar y, si se desea que además respete el orden de llegada, debería ingresarse junto con la prioridad y el elemento a ser tratado.

TDA Cola de Prioridad con Clase PriorityQueue				
Signatura	Return	Descripción Oracle	Orden	Equivale a
<code>add(E e)</code>	boolean	Insertar el elemento según el orden establecido en el <i>compareTo</i>	$O(\log n)$	insertar
<code>contains(Object o)</code>	boolean	Retorna true si el elemento está en la estructura	$O(n)$	no se vio en la materia (equivaldría a pertenece)
<code>peek()</code>	E	Retorna el elemento del frente de la cola (sin sacarlo). Retorna null si la estructura está vacía	$O(1)$	recuperar mínimo
<code>poll()</code>	E	Retorna el elemento del frente y lo elimina de la cola. Retorna null si la estructura está vacía	$O(\log n)$	eliminar mínimo + recuperar mínimo
<code>remove(Object o)</code>	boolean	Busca y elimina la primera aparición que encuentre del elemento buscado	$O(n)$	no se vio en la materia

1.5. Mapeo

1.5.1. TreeMap

Se trata de un TDA Mapeo que permite relacionar elementos de un conjunto dominio con un único elemento rango. Los elementos del dominio se guardan ordenados, y no permiten dominio repetido. La estructura interna es la misma que la de *TreeSet* (árbol Red-Black) en base a un árbol autobalanceado.

Para *mapeo uno a muchos* se utiliza la misma estructura guardando una lista u otra estructura similar, que permita almacenar más de un elemento en el rango. El orden de las operaciones será respecto a la cantidad de pares (dominio, rango) almacenados.

- (**) Se debe cuidar que al obtener el conjunto de rangos en realidad se va a obtener una colección de las estructuras (listas o lo que sea) y no elementos sueltos.
- (***) Para recorrer el conjunto de dominios (Set) o de Rangos (Collection) se deberá utilizar un Iterator (ver en Sección 1.7).

TDA Mapeo con Clase TreeMap				
Signatura	Return	Descripción Oracle	Orden	Equivalencia
put(K key, V value)	V	Carga la dupla dominio (key) - rango (value)	$O(\log n)$	asociar
get(Object key)	V	Dado el dominio (key) retorna el valor asociado (value) o null si el dominio no estaba en el mapeo	$O(\log n)$	valor
remove(Object key)	V	Quita el par que tenga al dominio (key), si existe	$O(\log n)$	desasociar
containsKey(Object key)	boolean	Retorna true si hay algún par cuyo dominio sea igual a key	$O(\log n)$	existe dominio
keySet()	Set<K> (***)	Retorna un conjunto (set) con todos los valores de key. Set puede ser TreeSet o HashSet vistos anteriormente	$O(n)$	obtener conjunto dominio
values()	Collection<V>	Retorna todos los valores del rango en una Collection	$O(n)$	obtener conjunto rango (**)
firstKey()	K	Retorna el dominio (key) más pequeño		dominio más pequeño
lastKey()	K	Retorna el dominio (key) más grande		dominio más grande

1.5.2. HashMap

De manera similar a TreeMap, implementa el TDA Mapeo pero los elementos del dominio se guardan en una tabla hash de tipo abierto. Las operaciones que no mantiene son aquellas relacionadas a orden entre los elementos del dominio, ya que serían operaciones muy costosas.

TDA Mapeo con Clase HashMap				
Signatura	Return	Descripción Oracle	Orden	Equivalencia
put(K key, V value)	V	Carga la dupla dominio (key) - rango (value)	$O(1)$	asociar
get(Object key)	V	Dado el dominio (key) retorna el valor asociado (value) o null si el dominio no estaba en el mapeo	$O(1)$	valor
remove(Object key)	V	Quita el par que tenga al dominio (key), si existe	$O(1)$	desasociar
containsKey(Object key)	boolean	Retorna true si hay algún par cuyo dominio sea igual a key	$O(1)$	existe dominio
keySet()	Set<K> (***)	Retorna un conjunto (set) con todos los valores de key. Set puede ser TreeSet o HashSet vistos anteriormente	$O(n)$	obtener conjunto dominio
values()	Collection<V>	Retorna todos los valores del rango en una Collection	$O(n)$	obtener conjunto rango (**)

1.6. Estructuras que no tienen implementación específica

1.6.1. Tabla de búsqueda

Dado que una tabla de búsqueda es básicamente una relación entre dominio-rango, donde el dominio es la clave del elemento y el rango es el elemento en sí, para hacer uso de tabla de búsqueda en Java se pueden utilizar las mismas estructuras que para TDA Mapeo:

- TreeMap: para elementos que permiten ordenarse, con tiempos de orden logarítmicos para las operaciones de inserción, eliminación y búsqueda exacta;

- **HashMap:** para elementos que no se requiera acceder a ellos de forma ordenada. Se recomienda generar automáticamente los métodos *equals* y *hashCode* sobre los atributos que conforman la clave.

1.6.2. Grafos

Al implementar grafos, hay que considerar la información necesaria de los vértices y los arcos. Si es un tipo vértice complejo (como información de un aeropuerto), se puede implementar con una tabla de búsqueda (mapeo uno a uno) para la información de los vértices (clave, información). Por ejemplo, como dominio la clave (código único del aeropuerto); y como dato asociado su información (nombre real, coordenadas, ciudad, país, cantidad de pistas, etc.). Por separado se guardarán los arcos en otro mapeo, que en este caso será uno a muchos (código aeropuerto origen, lista de códigos de aeropuertos destino).

- **TreeMap:** para elementos que permiten ordenarse, con tiempos de orden logarítmicos para las operaciones de inserción, eliminación y búsqueda exacta;
- **HashMap:** para elementos que no se requiera acceder a ellos de forma ordenada. Se recomienda generar automáticamente los métodos *equals* y *hashCode* sobre los atributos que conforman la clave.

1.6.3. Árboles genéricos

Un árbol genérico es un tipo de grafo dirigido, así que su implementación puede hacerse de la misma manera descripta antes.

1.7. Recorrer una colección Java con Iterator

Un iterador (clase *Iterator*) es un mecanismo que provee Java para acceder secuencialmente a elementos de una colección. Lo que permite es acceder a los objetos almacenados en la estructura sin exponer la parte interna de la estructura, de forma similar a lo que realizamos en la materia con los métodos *listar* que retornaban todos los elementos de una estructura, o una parte de ellos, en una lista; para que luego pueda ser recorrida secuencialmente o consumida recuperando el elemento de la primera posición y luego eliminarlo.

Iterator puede utilizarse sobre cualquiera de las estructuras vistas de tipo conjunto o colecciones. La diferencia entre estos grupos es que en un conjunto no puede haber elementos repetidos (como en un hash o un árbol binario de búsqueda), mientras que en una colección sí (como una lista o un arreglo).

A continuación se muestra como ejemplo el método *mostrarElementos* (se muestra en Algoritmo 1.1) que recibe una estructura *Iterable* y la recorre mediante un *Iterator* para mostrar elemento a elemento por pantalla.

Como se puede observar, el *Iterator* puede crearse sobre cualquier estructura de tipo map o set, sean implementados con hash o árbol. Además, en el caso del mapeo, puede crearse el *Iterator* sobre el conjunto del dominio (key) o de los rangos (values) o sobre los pares (*entrySet*) lo que da la posibilidad de recorrerlos con distintos propósitos, aunque lo más común en estas estructuras es la consulta de obtener el rango o los rangos asociados a un dominio particular.

Luego se presenta un fragmento de código que utiliza el método *mostrarElementos* anterior para mostrar el contenido de un mapeo implementado con *HashMap* y de un conjunto implementado con *TreeSet* (Algoritmo 1.2).

Los resultados de la ejecución del primer código se muestran en la Figura 1.1.

Algoritmo 1.1 Código que crea un iterador y lo recorre

```
public static void mostrarElementos(Iterable s) {
    Iterator it = s.iterator();
    while (it.hasNext()) {
        System.out.println(" - " + it.next());
    }
}
```

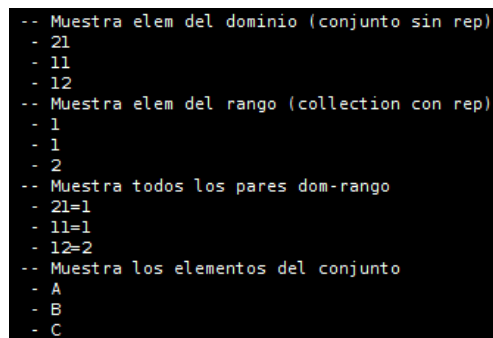
Algoritmo 1.2 Código que crea mapeo y conjunto y los muestra

```
HashMap map = new HashMap();
map.put(11, 1);
map.put(12, 2);
map.put(21, 1);

System.out.println("-- Muestra elem del dominio (conjunto sin rep)");
mostrarElementos(map.keySet());
System.out.println("-- Muestra elem del rango (collection con rep)");
mostrarElementos(map.values());
System.out.println("-- Muestra todos los pares dom-rango");
mostrarElementos(map.entrySet());

TreeSet set = new TreeSet();
set.add("A");
set.add("B");
set.add("C");
// los siguientes no los agrega pq son repetidos
set.add("A");
set.add("B");

System.out.println("-- Muestra los elementos del conjunto");
mostrarElementos(set);
```



```
-- Muestra elem del dominio (conjunto sin rep)
- 21
- 11
- 12
-- Muestra elem del rango (collection con rep)
- 1
- 1
- 2
-- Muestra todos los pares dom-rango
- 21=1
- 11=1
- 12=2
-- Muestra los elementos del conjunto
- A
- B
- C
```

Figura 1.1: Resultados obtenidos al ejecutar el código anterior