**Framework**

Un *framework* es una agrupación de clases e interfaces que proporcionan una arquitectura lista para desarrollar software. Por ende, si se quiere implementar una nueva característica o clase, no es necesario definir un nuevo *framework* si ya existe uno. No obstante, una buena práctica en el paradigma orientado a objetos es incluir un *framework* con una colección de clases tal que todas las clases realicen el mismo tipo de operaciones.

**Colecciones en Java**

Cualquier grupo de objetos individuales que se representa como una sola unidad se le conoce como una colección de objetos. En el lenguaje de programación Java, en Java Development Kit 1.2 se definió un modelo denominado *Collection Framework* que contiene todas las clases de colección con sus respectivas interfaces. De ahí que la interfaz de colección **java.util.Collection** y la interfaz de mapa **java.util.Map** son las dos interfaces principales de las clases de colección en Java.

**Collection Framework en Java**

Antes de que existiera el *Collection Framework*, es decir, antes de JDK 1.2, los métodos estándar para agrupar objetos en Java, o colecciones, eran *Arrays*, *Vectors* o *Hashtables*. Todas estas colecciones no tenían una interfaz en común. Por consiguiente, aunque el objetivo principal de todas las colecciones es el mismo, la implementación de todas estas colecciones se definió de forma independiente y en consecuencia no había ninguna relación entre ellas. Además, era muy difícil para los programadores recordar los diferentes métodos, sintaxis y constructores existentes para cada clase de colección.

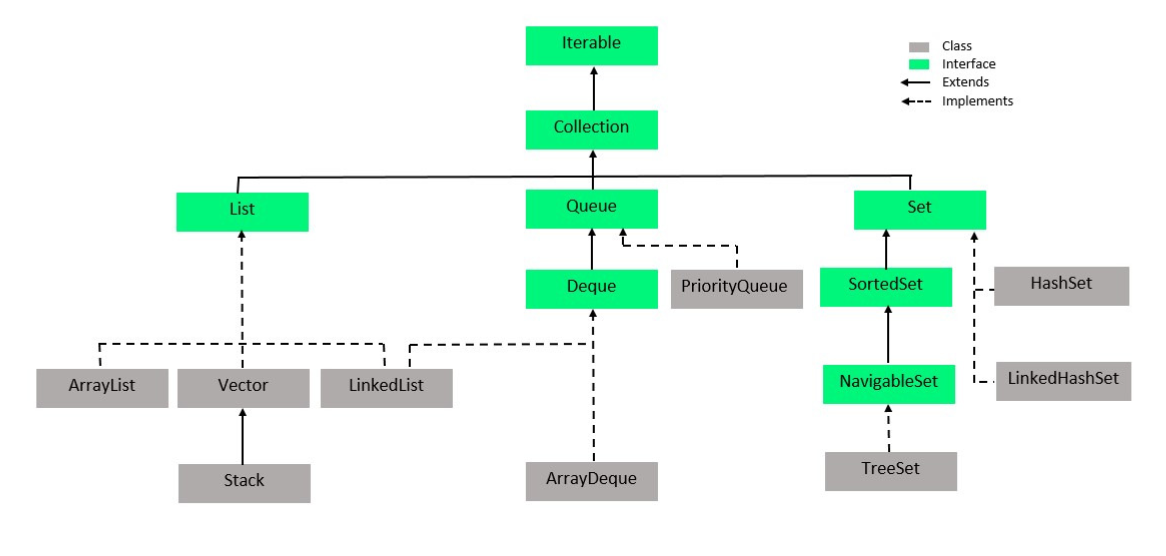
**Ventajas del Collection Framework en Java**

Como ya se mencionó anteriormente, la falta de un *framework* dio lugar a las desventajas descritas en la sección anterior. Sin embargo, luego de que se declaró el *framework* se comenzaron a presentar ventajas.

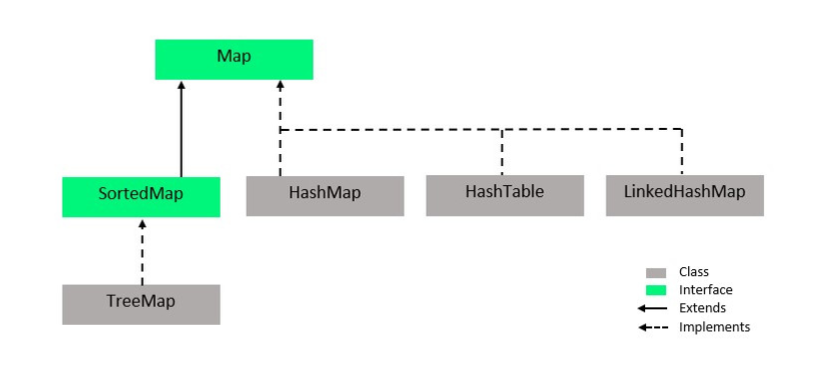
* **API consistente:** la API tiene un conjunto básico de interfaces como *Collection*, *Set*, *List* o *Map*, donde todas las clases, *ArrayList*, *LinkedList*, *Vector*, que implementan estas interfaces tienen métodos en común.
* **Reduce la complejidad al programar:** un programador ya no tiene que preocuparse por el diseño de la Colección, lo cual le permite priorizar el resto de su programa. Por lo anterior, uno de los principales aspectos del paradigma orientado a objetos, el cual es abstracción se logró implementar satisfactoriamente.
* **Aumenta la eficiencia y la calidad del programa:** la eficiencia se ve incrementada gracias a la proporción de implementaciones de alto rendimiento para las estructuras de datos junto con algunos algoritmos útiles, ya que, en este caso, el programador no necesita preocuparse por elegir la mejor implementación de una estructura de datos particular. Simplemente puede utilizar la implementación predefinida y así aumentar el rendimiento de su algoritmo/programa.

**Jerarquía del Collection Framework en Java**

El paquete **java.util** contiene todas las clases e interfaces que requiere el *Collection Framework*. Asimismo, el *Collection Framework* contiene una interfaz conocida como *Iterable*, la cual proporciona un iterador para recorrer todas las colecciones. Todas las colecciones que amplían la interfaz, aumentan al mismo tiempo el rango del iterador y los métodos de esta. La siguiente imagen ilustra la jerarquía del *Collection Framework*.



GeeksforGeeks. (2023). *Jerarquía del Collection Framework en Java* [PNG]. GeeksforGeeks. <https://media.geeksforgeeks.org/wp-content/uploads/20230124151239/Collections-in-Java-768.png>



GeeksforGeeks. (2023). *Jerarquía del Collection Framework en Java* [PNG]. GeeksforGeeks. <https://media.geeksforgeeks.org/wp-content/cdn-uploads/20200811210611/Collection-Framework-2.png>

Antes de profundizar en los diferentes componentes del modelo, es importante recordar los conceptos de clase e interfaz.

* **Clase:** una clase es un modelo o prototipo definido por el usuario a partir del cual se van a crear objetos. En esta, se representa un conjunto de atributos y métodos que serán comunes para todos los objetos de la clase.
* **Interfaz:** al igual que una clase, una interfaz puede tener métodos y atributos, pero los métodos declarados en una interfaz son abstractos por defecto. Entonces, las interfaces especifican qué debe hacer una clase más no el cómo, es decir, son la plantilla de la clase.

**Clase contra Interfaz**

| **Clase** | **Interfaz** |
| --- | --- |
| Una clase es un prototipo definido por el usuario para construir objetos en Java. | Una interfaz es un modelo definido por el usuario que describe la estructura de cada clase que la implementa. |
| Se utiliza para instanciar objetos. | No se puede utilizar para instanciar objetos. |
| Una clase puede tener modificadores de acceso públicos y predeterminados. | Una Interfaz puede tener modificadores de acceso públicos y predeterminados. |
| Las clases pueden ser concretas o abstractas. | Todas las interfaces son abstractas. |
| Una clase consta de constructores, métodos y atributos. Los métodos están definidos en una clase. | Una interfaz consta de atributos y métodos. Los métodos no están definidos en una interfaz, sólo contiene sus prototipos. |

**Métodos de la interfaz Collection**

Esta interfaz contiene varios métodos que pueden ser utilizados por todas las colecciones que implementan esta interfaz. Los cuales son:

* **add(Object)** se utiliza para agregar un objeto a la colección.
* **addAll(Collection c)** agrega todos los elementos de la colección que se recibe como parámetro a la colección.
* **clear()** elimina todos los elementos de la colección.
* **contains(Object o)** devuelve verdadero si la colección contiene el objeto especificado.
* **containsAll(Collection c)** devuelve verdadero si la colección contiene todos los elementos de la colección que recibe como parámetro.
* **equals(Object o)** compara el objeto especificado con la colección para determinar la igualdad.
* **hashCode()** se utiliza para devolver el valor del código hash para esta colección, es decir, el identificador de 32 bits que se almacena en un Hash en la instancia de la clase.
* **isEmpty()** devuelve verdadero si la colección no contiene elementos.
* **iterator()** devuelve un iterador sobre los elementos de esta colección.
* **max()** se utiliza para devolver el valor máximo presente en la colección.
* **parallelStream()** devuelve un *Stream* paralelo con esta colección como fuente, dicho de otra manera, este genera un *Stream* donde cada elemento no depende de otro para ser procesado.
* **remove(Object o)** se utiliza para eliminar el objeto dado de la colección. Sin embargo, si hay valores duplicados, este método elimina la primera aparición del objeto.
* **removeAll(Collection c)** se utiliza para eliminar de la colección todos los objetos contenidos en la colección que recibe como parámetro.
* **removeIf(Predicate filter)** se utiliza para eliminar todos los elementos de esta colección que satisfacen el predicado dado.
* **retainAll(Collection c)** se utiliza para conservar sólo los elementos de la colección que están contenidos en la colección especificada.
* **size()** se utiliza para conocer la cantidad de elementos de la colección.
* **spliterator()** se utiliza para crear un *Spliterator* sobre los elementos de esta colección, el cual nos permite recorrer y dividir una secuencia de elementos.
* **stream()** devuelve un *Stream* secuencial con la colección como fuente, el cual permite operar con la colección y hacer que el procesamiento masivo de datos sea rápido y fácil de leer.
* **toArray()** devuelve un *array* que contiene todos los elementos de la colección.

**Interfaces y clases del Collection Framework**

**Interfaz Iterable**

Esta interfaz es la raíz de todo el *Collection Framework*. Como se observa en la imagen, la interfaz de *Collection* amplía la interfaz *Iterable*. Por lo tanto, todas las interfaces y clases implementan esta interfaz. La función principal de esta interfaz es proporcionar un iterador para las colecciones. De ahí que, esta interfaz contiene sólo un método abstracto que es el iterador.

**Interfaz Collection**

Como ya se mencionó, esta interfaz amplía la interfaz *Iterable* y a su vez amplia la implementación de todas las clases en el *Collection Framework*. Esta interfaz contiene todos los métodos básicos que tiene cada colección, como agregar datos a la colección, eliminar datos, borrar datos. Todos estos métodos se implementan en esta interfaz porque estos son usados por todas las clases independientemente de su lógica. Además, tener estos métodos en esta interfaz garantiza que los nombres de los métodos sean universales para todas las colecciones. Por consiguiente, esta interfaz construye una base sobre la cual se implementan las clases del resto de interfaces.

**Interfaz List**

Esta es una interfaz secundaria de la interfaz de *Collection*. Se enfoca en las clases que son una colección secuencial en la que el usuario de la interfaz tiene control sobre cualquier elemento que es insertado a la lista. Además, el usuario puede acceder a sus elementos por un índice entero o buscar algún elemento en la lista. Por otra parte, a diferencia de la interfaz *Set*, la interfaz *List* si permite que haya elementos repetidos junto con varios elementos nulos. De ahí que esta interfaz está implementada por clases como *ArrayList*, *Vector*, *Stack* y *LinkedList*. En consecuencia, podemos crear una instancia de un objeto de *List* con cualquiera de las clases mencionadas.

**Clase ArrayList**

Esta clase implementa una lista como un arreglo al que se le puede cambiar el tamaño, junto con todas las operaciones opcionales de la lista e igual permite todos los elementos, incluido *null*. Asimismo, provee métodos que manipulan el tamaño del arreglo interno usado para almacenar la lista. Por lo que, esta clase es similar a la clase *Vector*, solo que no está sincronizada, es decir, que múltiples subprocesos pueden operar en un *ArrayList* al mismo tiempo para hacer un *ArrayList* seguro para subprocesos. Sin embargo, se puede sincronizar externamente usando el método *Collections.synchronizedList()*.

**Clase Vector**

Esta clase implementa una variedad creciente de objetos. Al igual que una matriz, contiene elementos a los que se puede acceder mediante un índice entero. Sin embargo, el tamaño de un *Vector* puede aumentar o disminuir según se requiera para permitir las operaciones de adición y eliminación de elementos una vez que ya se creó un objeto de la clase. Al igual que la clase *ArrayList*, esta clase también permite todos los elementos, incluido *null*. Además, como ya se mencionó anteriormente, la clase *Vector* si es sincronizada, lo cual implica que sólo un único subproceso puede operar en un método de *Vector* a la vez.

**Clase Stack**

La clase *Stack* representa una pila de objetos donde el último en entrar, es el primero en salir (LIFO). Por ende, cuando se crea una pila por primera vez, no contiene elementos. Asimismo, amplía la clase *Vector* con cinco operaciones que permiten tratar un vector como una pila.

* **empty()** prueba si la pila está vacía.
* **peek()** mira el objeto en la parte superior de la pila sin sacarlo de la pila.
* **pop()** elimina el objeto en la parte superior de la pila y devuelve ese objeto como el valor de esta función.
* **push(E item)** agrega un elemento a la parte superior de la pila.
* **search(Object o)** devuelve la posición basada en 1 de donde se encuentra un objeto en la pila.

**Clase LinkedList**

Esta clase es la implementación de una lista doblemente ligada de las interfaces *List* y *Deque*. Incluye todas las operaciones de lista opcionales y permite todos los elementos (incluido *null*). Además, todas las operaciones se realizan como se podría esperar en una lista doblemente ligada. Por otro lado, las operaciones que indexan la lista recorrerán la lista desde el principio o el final, de acuerdo con lo que esté más cerca del índice especificado. Hay que tener en cuenta que esta implementación no está sincronizada, es decir, si varios subprocesos acceden a una *LinkedList* al mismo tiempo y al menos uno de los subprocesos modifica la lista estructuralmente, debe sincronizarse externamente.

**Aspectos a tomar en cuenta para elegir alguna de las clases de List**

Como se vio anteriormente, las clases *ArrayList*, *LinkedList*, *Vector* y *Stack* son todas implementaciones de la interfaz *List*, pero se utilizan en diferentes situaciones debido a sus características y rendimiento. Por ende, conviene tener en cuenta algunos aspectos para saber cuándo conviene usar cada una.

* **ArrayList:**
  + Se suele utilizar cuando se necesita una lista dinámica que se puede redimensionar y acceder rápidamente a sus elementos con un índice. En consecuencia, la clase *ArrayList* es la elección más común.
  + En cuanto al rendimiento, esta clase ofrece un acceso rápido a elementos por índice, sin embargo, puede ser menos eficiente al insertar o eliminar elementos que se encuentran al medio de la lista debido a que se tienen que desplazar elementos.
* **LinkedList:**
  + Se recomienda utilizar cuando vas a insertar o eliminar de manera frecuente elementos al medio de la lista, ya que en este aspecto la clase *LinkedList* puede ser más eficiente que la clase *ArrayList*. También es útil cuando se requiere iterar sobre la lista en ambas direcciones, es decir del inicio al final o del final al inicio.
  + Ahora bien, el rendimiento en cuanto a insertar y eliminar elementos en la clase *LinkedList* es más rápido que en la clase *ArrayList*, pero el acceso aleatorio es más lento.
* **Vector:**
  + Se utiliza con menos frecuencia en las aplicaciones modernas. Es similar a la clase *ArrayList*, pero esta clase si es sincronizada, lo que significa que es segura para el uso de hilos múltiples. Por ello, si necesitas una lista sincronizada, puedes considerar usar la clase *Vector*. De lo contrario, es preferible usar *ArrayList*.
  + En lo que al rendimiento se refiere, a causa de la sincronización, la clase *Vector* puede ser más lenta que la clase *ArrayList* en operaciones sin hilos múltiples.
* **Stack:**
  + Se recomienda usar si necesitas una estructura de datos tipo LIFO (Last In, First Out). Sin embargo, es preferible usar la interfaz *Deque* con la clase *LinkedList* o *ArrayDeque* en lugar de la clase *Stack*, ya que el uso de esta clase no se aconseja desde Java 1.5.
  + El rendimiento suele ser aceptable para operaciones de apilar y desapilar, no obstante, se debe tener en cuenta que la interfaz *Deque* ofrece una mayor flexibilidad y opciones.

**Interfaz Queue**

La interfaz *Queue* mantiene la lógica FIFO donde el primero en entrar, es el primero en salir, lo cual es similar a una cola de espera en el mundo real. Por ello, esta interfaz está dedicada a almacenar elementos donde el orden de los elementos sí importa. Por ejemplo, cuando queremos reservar un boleto, los boletos se venden por orden de llegada. Por lo tanto, la persona cuya solicitud llega primero a la cola obtiene el billete antes que los demás. En este sentido, hay clases como *PriorityQueue*, *ArrayDeque* y *LinkedList* para implementar la interfaz *Queue*, por lo que, se puede crear una instancia con cualquiera de estas clases.

**Clase PriorityQueue**

Esta clase permite usar la lógica de una cola de prioridad ilimitada, la cual se basa en un *Heap* de prioridad. Además, los elementos de la cola de prioridad se ordenan según su orden natural, o de acuerdo con un parámetro *Comparator* proporcionado en el momento de la construcción de la cola, conforme al constructor que se utilice. Lamentablemente, la *PriorityQueue* no permite elementos *null*. Asimismo, como se basa en el orden natural tampoco permite la inserción de objetos no comparables, ya que hacerlo puede resultar en *ClassCastException*.

**Interfaz Deque**

Esta es una variación muy ligera de la interfaz *Queue*. Por ende, la interfaz *Deque*, también conocida como cola doble, es una interfaz con la que podemos agregar y eliminar elementos de ambos extremos de la cola. Como ya se mencionó, la interfaz *Deque* amplía la interfaz *Queue*. Por otro lado, las clases que implementan esta interfaz son *ArrayDeque* y *LinkedList*, por lo cual, podemos crear una instancia de alguna de estas clases.

**Clase ArrayDeque**

Esta clase implementa un *array* redimensionable con la interfaz *Deque* que a su vez amplía la interfaz *Queue*. Un *ArrayDeque* no tiene restricciones de capacidad, porque crece según sea necesario para soportar el uso. Sin embargo, no son seguros para subprocesos, ya que en ausencia de sincronización externa, no admiten el acceso simultáneo de varios subprocesos. Además, los elementos nulos están prohibidos en esta clase. A pesar de lo anterior, es probable que esta clase sea más rápida que la clase *Stack* cuando se usa como pila y más rápida que la clase *LinkedList* cuando se usa como cola.

**Clase LinkedList**

Esta clase es la implementación de una lista doblemente ligada de las interfaces *List* y *Deque*, la cual a su vez amplía la interfaz *Queue*. Incluye todas las operaciones de lista opcionales y permite todos los elementos (incluido *null*). Además, si se usa como una cola todas las operaciones se realizan como se podría esperar. Por otra parte, las operaciones que indexan la cola permiten recorrerla de inicio a fin o viceversa, según se requiera. Nuevamente, se debe tener en cuenta que esta implementación no está sincronizada, por lo cual, si varios subprocesos acceden a una *LinkedList* al mismo tiempo y al menos uno de los subprocesos modifica la lista estructuralmente, debe sincronizarse externamente.

**Aspectos a tomar en cuenta para elegir alguna de las clases de Queue**

De acuerdo con lo ya mencionado, elegir alguna de las clases *PriorityQueue*, *ArrayDeque* o *LinkedList* para implementar la interfaz *Queue* depende de nuestras necesidades y de las operaciones que planeamos realizar con la estructura de datos cola. Por consiguiente, conviene tener en cuenta algunos aspectos para determinar la clase que más nos convendrá usar para determinada aplicación.

* **PriorityQueue:**
  + Se recomienda usar cuando necesitas una cola de prioridad, es decir, cuando quieres que los elementos se almacenen y recuperen según un orden específico definido por su prioridad. Por lo anterior, los elementos se recuperan de la cola en orden ascendente o descendente de acuerdo con la prioridad que se haya definido.
  + En lo que al rendimiento se refiere, la *PriorityQueue* es eficiente para insertar y eliminar elementos según su prioridad, pero no es eficiente si necesitas acceder a elementos en posiciones arbitrarias o si la prioridad de los elementos cambia con frecuencia.
* **ArrayDeque:**
  + Cuándo quieres una cola doble para insertar y eliminar elementos de manera eficiente tanto al principio como al final, entonces la clase *ArrayDeque* es una excelente opción. Por otra parte, un objeto de esta clase se puede usar como una cola FIFO o una pila LIFO según tus necesidades.
  + El rendimiento de la clase *ArrayDeque* es muy eficiente para operaciones de inserción y eliminación en ambos extremos de la cola, pero no brinda un orden basado en una prioridad como la clase *PriorityQueue*.
* **LinkedList:**
  + Un objeto de la clase *LinkedList* puede usarse para implementar una cola FIFO o una pila LIFO. Sin embargo, generalmente es menos eficiente que la clase *ArrayDeque* para estas operaciones. A pesar de lo anterior, puedes considerar usar la clase *LinkedList* si necesitas una cola y también si deseas acceder a elementos en posiciones arbitrarias con frecuencia.
  + Respecto al rendimiento, la clase *LinkedList* es menos eficiente que la clase *ArrayDeque* para operaciones de inserción y eliminación en los extremos de la cola, pero es mejor si necesitas acceder a alimentos aleatorios.

**Interfaz Set**

Un *set* es una colección desordenada de objetos en la que no se pueden almacenar elementos duplicados, es decir, un *set* no posee dos elementos **e1** y **e2** tal que se pueda usar el método **e1.equals(e2)**. Esta interfaz se utiliza cuando deseamos evitar la duplicación de los objetos y deseamos almacenar sólo objetos únicos, con máximo un elemento nulo, puesto que, esta colección modela la abstracción matemática de un conjunto. Por ello, esta interfaz se puede implementar mediante la creación de una instancia de alguna clase como *HashSet*, *TreeSet* y *LinkedHashSet*.

**Clase HashSet**

Esta clase implementa la interfaz *Set*, y a su vez está respaldada por una tabla hash, la cual en realidad es una instancia de la clase *HashMap*. Sin embargo, esta clase no ofrece garantía en cuanto al orden de iteración del conjunto; en particular, no garantiza que el orden se mantendrá constante en el tiempo. Asimismo, esta clase si permite el elemento *null*. Por otra parte, tiene un rendimiento de tiempo constante para las operaciones de inserción y eliminación promedio. Finalmente, hay que tener en cuenta que esta implementación no está sincronizada, dicho de otra forma, si varios subprocesos acceden a un *HashSet* al mismo tiempo y al menos uno de los subprocesos lo modifica, se debe sincronizar externamente.

Por lo general, esto se logra mediante la sincronización de algún objeto que encapsule naturalmente el *HashSet*. Si no existe tal objeto, el *HashSet* debe **ajustarse** utilizando el método *Collections.synchronizedSet*. Es mejor hacerlo en el momento de la creación, para evitar el acceso accidental y no sincronizado al *HashSet*.

**Clase LinkedHashSet**

Esta clase es la implementación de una tabla hash y una lista ligada de la interfaz *Set*, por lo cual, tiene un orden de iteración predecible. A diferencia de la clase *HashSet*, la clase *LinkedHashSet* mantiene una lista doblemente ligada que recorre todas sus entradas. Por lo cual, esta lista doblemente ligada define el orden de iteración, el cual es el orden en el que se insertaron los elementos en el *Set*. No hay que perder de vista que el orden de inserción no se ve afectado si un elemento se vuelve a insertar en el *LinkedHashSet*. En consecuencia, esta implementación evita los pedidos no especificados y generalmente caóticos que se generan con la clase *HashSet*, sin incurrir en el mayor costo asociado con la clase *TreeSet*.

Por lo anterior, se puede utilizar la clase *LinkedHashSet* para generar una copia de un *Set* que tenga el mismo orden que el original, independientemente de la implementación del conjunto original.

**Interfaz SortedSet**

Esta interfaz es muy similar a la interfaz *Set*. La única diferencia es que esta interfaz tiene métodos adicionales que mantienen el orden de los elementos. Por consiguiente, la interfaz *SortedSet* amplía la interfaz *Set* y se utiliza para manejar los datos que deben ordenarse. La clase que implementa esta interfaz es *TreeSet*, por lo que se puede crear una instancia con esta clase para usar la interfaz *SortedSet*.

**Interfaz NavigableSet**

La interfaz *NavigableSet* se extiende desde la interfaz *SortedSet*. Además de los métodos de la interfaz *SortedSet*, la interfaz *NavigableSet* tiene métodos de navegación que brindan coincidencias más cercanas, como *floor*, *ceiling*, *lower* y *higher*. Un *NavigableSet* se puede recorrer en orden ascendente y descendente. Aunque permite el uso elementos nulos, no se recomienda, ya que estos elementos pueden generar resultados ambiguos.

**Clase TreeSet**

Esta clase implementa la interfaz *Set*, *SortedSet* y *NavigableSet* utilizando una estructura de árbol, normalmente un *Red-Black Tree*, pues se usa para administrar el orden. Por defecto, se tiene un orden ascendente, pero se puede cambiar usando un *Comparator*. Además, su complejidad temporal asintótica es de O(log n) para sus operaciones de inserción, eliminación y búsqueda. Por otra parte, al igual que la clase *HashSet* esta clase tampoco es sincronizada, por lo que, si se quiere sincronizar se debe hacer de la misma forma que se hace con un objeto de la clase *HashSet*. Además, dada la naturaleza de este *Set*, se tienen algunos métodos exclusivos a la hora de implementarlo.

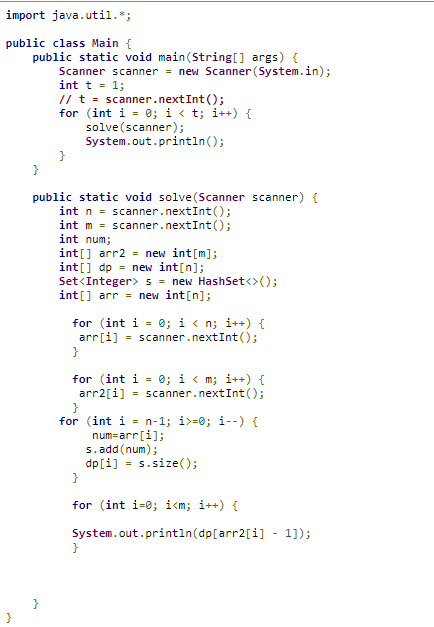
* **ceiling(E element)** devuelve el elemento más pequeño en el conjunto que es mayor o igual al elemento especificado.
* **floor(E element)** devuelve el elemento más grande en el conjunto que es menor o igual al elemento especificado.
* **higher(E element)** devuelve el elemento más pequeño en el conjunto que es estrictamente mayor que el elemento especificado.
* **lower(E element)** devuelve el elemento más grande en el conjunto que es estrictamente menor que el elemento especificado.
* **pollFirst()** recupera y elimina el primer elemento más pequeño en el conjunto.
* **pollLast()** recupera y elimina el último elemento más grande en el conjunto.
* **descendingSet()** devuelve una vista inversa del conjunto, donde los elementos se almacenan en orden inverso.

**Aspectos a tomar en cuenta para elegir alguna de las clases de Set**

Para las clases mencionadas anteriormente, se debe definir cuál queremos usar según el problema que se nos presente y que encaje con las características especiales de cada *Set*. Por ejemplo, en el siguiente problema (B. Sereja and Suffixes Codeforces Problem 368B):

Se nos presenta una situación en la que se nos da un conjunto de números que tiene cualquier orden y contiene números repetidos, y se no da otro conjunto de números que representan índices del anterior conjunto de números brindados, queremos saber la cantidad de números diferentes que se encuentren después de cada índice que se nos ha brindado, además el problema debe ser solucionado en máximo un segundo para entradas de 105.

La situación es clara, necesitamos un *Set*, pues la respuesta será su tamaño después de recorrer la colección de números desde el índice brindado, pero además de esto necesitamos que el *Set* sea rápido al momento de realizar sus operaciones para cumplir con el límite de un segundo, y la clase de la interfaz *Set* que cumple con estas características es el *HashSet*, dado que este tiene las operaciones más rápidas y a su vez cumple con la función, como el resto de clases en la interfaz, de no admitir números repetidos. Por lo anterior, el código en el lenguaje de programación **Java** queda de la siguiente manera.

:

Hernández, G. D. A. (2023). *Solución a B. Sereja and Suffixes Codeforces Problem 368B* [PNG]. Herramienta de Recortes.

Cabe destacar que también se uso programación dinámica para calcular todas las respuestas en una sola pasada de la colección, así el índice que se nos proporciona por la segunda colección será el índice que contenga la respuesta en nuestro arreglo de soluciones, en este caso se le resta uno para adecuarse a los índices de los arreglos que son de 0-n, porque los índices de la segunda colección son desde 1-n.

Como ya se dijo, la elección entre la clase *HashSet*, *LinkedHashSet* y *TreeSet* para implementar la interfaz *Set* en **Java** depende de nuestros requerimientos junto con el rendimiento y comportamiento que busquemos. Por ende, nos conviene tener en cuenta ciertos aspectos a la hora de elegir alguna de estas clases para nuestra aplicación.

* **HashSet:**
  + Se suele utilizar cuando se necesita una colección de elementos únicos sin tomar en cuenta el orden de inserción ni tampoco un orden específico para los elementos. Asimismo, es la clase más rápida para la mayoría de las operaciones, como inserción, eliminación y búsqueda.
  + En lo que al rendimiento se refiere, un objeto de la clase *HashSet* es altamente eficiente para las operaciones mencionadas anteriormente, ya que utiliza una tabla hash para el almacenamiento. Sin embargo, no garantiza ningún orden específico de los elementos.
* **LinkedHashSet:**
  + Se recomienda utilizar si quieres mantener el orden de inserción de los elementos, y en consecuencia los elementos también se pueden recuperar en el mismo orden en que se insertaron.
  + En cuanto al rendimiento la clase *LinkedHashSet* es un poco más lenta que la clase *HashSet* debido a que mantiene el orden de inserción, pero es más rápida que la clase *TreeSet*.
* **TreeSet:**
  + Se utiliza usualmente cuando necesitamos que los elementos se almacenen y recuperen de acuerdo con el orden natural o con algún orden personalizado. Por consiguiente, la clase *TreeSet* garantiza que los elementos estén ordenados.
  + Con relación al rendimiento de la clase *TreeSet*, dado que esta utiliza un *Red-Black Tree* para el almacenamiento, provoca que las operaciones de inserción, eliminación y búsqueda tengan un rendimiento ligeramente más lento que la clase *HashSet* y *LinkedHashSet*. No obstante, tener un orden nos puede ser muy útil para ciertas aplicaciones.

**Interfaz Map**

Un *Map* es una estructura de datos que admite el par clave-valor para mapear los datos. Esta interfaz no admite claves duplicadas porque la misma clave no puede tener múltiples asignaciones; sin embargo, permite valores duplicados en claves diferentes. Un *Map* es útil cuando existen datos con los que queremos realizar operaciones en base a una clave. Además, la interfaz *Map* se puede implementar mediante diversas clases como *HashMap*, *Hashtable* y *LinkedHashMap* con la creación de una instancia de alguna de estas.

**Clase HashMap**

Esta clase es una implementación de la interfaz *Map* basada en una tabla hash. Por ende, nos proporciona todas las operaciones opcionales de *Map* y permite valores *null* junto con claves *null*. Por otro lado, la clase *HashMap* es similar a la clase *Hashtable*, con la diferencia de que la clase *HashMap* no está sincronizada y permite valores nulos. Lamentablemente, esta clase no ofrece garantía en cuanto al orden del mapa; en particular, no garantiza que el orden se mantendrá constante en el tiempo.

Asimismo, esta implementación nos proporciona un rendimiento en tiempo constante para las operaciones básicas *get* y *put*, suponiendo que la función hash disperse los elementos adecuadamente entre los depósitos.

**Clase Hashtable**

Esta clase implementa una tabla hash, que asigna claves a valores. Cualquier objeto no nulo se puede utilizar como clave o como valor. Para almacenar y recuperar objetos de una tabla hash con éxito, los objetos utilizados como claves deben implementar el método *hashCode* y el método *equals*.

Consecuentemente, una instancia de la clase *Hashtable* tiene dos parámetros que afectan su rendimiento: *initial capacity* y *load factor*. El *load factor* es la cantidad de depósitos en la tabla hash y la *initial capacity* es simplemente la capacidad en el momento en que se crea la tabla hash. Hay que tener en cuenta que la tabla hash está abierta: en el caso de una **colisión**, un único depósito almacena varias entradas, que deben buscarse secuencialmente. El *load factor* es una medida de qué tan llena se permite que esté la tabla hash antes de que su capacidad aumente automáticamente. Por lo anterior, los parámetros de *initial capacity* y *load factor* son meros consejos para la implementación.

**Clase LinkedHashMap**

La clase implementa una tabla hash junto con una lista ligada de la interfaz *Map*, por lo cual, se cuenta con un orden de iteración predecible. Además, esta clase se diferencia de la clase *HashMap* porque mantiene una lista doblemente ligada que recorre todas sus entradas. Esta lista doblemente ligada define el orden de iteración, que normalmente es el orden en que se insertaron las claves en el mapa. Cabe destacar que el orden de inserción no se ve afectado si se vuelve a insertar una clave en el mapa.

Asimismo, la clase evita los pedidos no especificados y generalmente caóticos proporcionados por las clases *HashMap* y *Hashtable*, sin incurrir en el mayor costo asociado con *TreeMap*. Por consiguiente, esta clase se puede utilizar para producir una copia de un mapa que tenga el mismo orden que el original, sin importar la implementación del mapa original.

**Interfaz SortedMap**

La interfaz *SortedMap* amplía la interfaz *Map* con una estipulación adicional de un orden total de claves. Por ello, las claves se ordenan por orden natural o en base a un *Comparator* especificado en el momento de la construcción, de acuerdo con el constructor utilizado. En consecuencia, todas las claves deben ser comparables.

**Clase Treemap**

Esta clase usa un *Red-Black Tree* basado en la interfaz *NavigableMap* y *SortedMap*, el cual se ordena en el orden natural de las *Keys* o por un comparador definido por el programador. Esta implementación garantiza un costo de tiempo de log(n) para las operaciones de búsqueda, obtención, inserción y eliminación de los elementos. Sin embargo, esta clase no está sincronizada. Por otra parte, dada su naturaleza al igual que el *TreeSet* esta clase posee métodos exclusivos que nos permiten acceder a ciertos datos de manera más rápida.

* **firstKey()** devuelve la clave del primer elemento en el mapa, que es la clave mínima en orden natural.
* **lastKey()** devuelve la clave del último elemento en el mapa, que es la clave máxima en orden natural.
* **headMap(K toKey)** devuelve una vista del mapa que contiene todas las entradas cuyas claves son menores que *toKey*.
* **tailMap(K fromKey)** devuelve una vista del mapa que contiene todas las entradas cuyas claves son mayores o iguales a *fromKey*.
* **subMap(K fromKey, K toKey)** devuelve una vista del mapa que contiene todas las entradas cuyas claves están en el rango [*fromKey*, *toKey*), es decir, desde *fromKey* (inclusivo) hasta *toKey* (exclusivo).
* **comparator()** devuelve el *Comparator* utilizado para ordenar las claves en el mapa. Si el mapa utiliza el orden natural, este método devolverá *null*.

**Aspectos a tomar en cuenta para elegir alguna de las clases de Map**

La elección entre las clases *HashMap*, *Hashtable*, *LinkedHashMap* o *TreeMap* para implementar la interfaz *Map* en Java depende de los requerimientos que se nos soliciten junto con el rendimiento y comportamiento que se busque. Por lo cual, nos conviene tener presentes ciertos aspectos al momento de elegir alguna de estas clases para nuestra aplicación.

* **HashMap:**
  + Se recomienda utilizar cuando necesites una estructura de datos de mapeo clave-valor con acceso rápido a los elementos. Es la elección predeterminada en la mayoría de los casos, ya que ofrece un buen equilibrio entre rendimiento y funcionalidad.
  + El rendimiento de esta clase es altamente eficiente para la mayoría de las operaciones, como inserción, eliminación y búsqueda, pero no garantiza ningún orden específico de las claves.
* **Hashtable:**
  + Aunque la clase *Hashtable* es similar a la clase *HashMap* en términos de funcionalidad, se utiliza con menos frecuencia en aplicaciones modernas debido a su sincronización, que puede degradar el rendimiento en entornos multihilo. Por ello, si necesitas un mapa sincronizado, *thread-safe*, puedes optar por la clase *Hashtable*. De lo contrario, la clase *HashMap* es preferible.
  + El rendimiento de la clase *Hashtable* es menor que el de la clase *HashMap* debido a la sincronización. Por ende, si no requieres de la sincronización, la clase *HashMap* será más rápida.
* **LinkedHashMap:**
  + Si deseas mantener el orden de inserción de las claves o el orden en el que se accede a los elementos, se recomienda utilizar la clase *LinkedHashMap*.
  + El rendimiento de la clase *LinkedHashMap* es ligeramente más lento que el de la clase *HashMap*, ya que debe realizar un seguimiento del orden de inserción o acceso. Sin embargo, es más rápido que el de la clase *TreeMap*.
* **TreeMap:**
  + Se recomienda utilizar la clase *TreeMap* cuando requieras que las claves sean almacenadas de acuerdo al orden natural o de acuerdo con un orden personalizado. Por lo anterior, esta clase garantiza un orden ascendente o descendente de las claves.
  + El rendimiento de la clase *TreeMap* es menos eficiente que el de las clases *HashMap* y *LinkedHashMap* en la mayoría de las operaciones debido a la estructura de árbol utilizada para mantener el orden. No obstante, ofrece la ventaja de un acceso ordenado a las claves.