Apunte 6 - TDAs de Propósito Específico

Índice general

Índice de figuras

Algoritmos

Índice de ejercicios

Apunte 6

TDAs de propósito específico

Actualizado: 16 de abril de 2021

En esta unidad trabajaremos sobre la siguiente pregunta:

> *¿Qué tipos de datos abstractos pueden implementarse utilizando las estructuras vistas anteriormente y para resolver qué tipo de problemas resultan útiles?*

A lo largo de este capítulo se presentará una serie de tipos de dato abstracto (TDA) que, a diferencia de los vistos en los capítulos anteriores, que trataban sobre estructuras propiamente dichas, estos TDA aprovechan a las estructuras conocidas para lograr comportamientos particulares. De dichos TDA se presentan a continuación los conceptos básicos, usos y operaciones más importantes. También se propondrán implementaciones en base a las estructuras conocidas y se analizará el uso de memoria y el tiempo de ejecución en cada caso.

6.1. TDA Conjunto

El TDA Conjunto modela el comportamiento de un conjunto discreto, en el cual las operaciones se parecen a las vistas en matemática: agregar un elemento, saber si un elemento ya existe o pertenece al conjunto, eliminar un elemento.

Figura 6.1: Representación gráfica del modelo de un TDA Conjunto

6.1.1. Operaciones del TDA Conjunto

Como se explicó antes, el propósito principal del tipo de dato abstracto *Conjunto* es muy sencillo y sólo desea mantener elementos y preguntar si pertenecen o no al conjunto. Las operaciones básicas de dicha estructura son:

- *constructor vacío*:
  // crea un conjunto sin elementos.
- *agregar* (valor): boolean
  // recibe un valor de tipo elemento. Si no existe previamente en el conjunto y lo puede agregar con éxito devuelve *verdadero* y *falso* en caso contrario.
- *quitar*(valor): boolean
  // elimina al elemento del conjunto. Si lo encuentra y la operación de eliminación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
- *pertenece*(valor): boolean
  // si en el conjunto se encuentra almacenado el elemento ingresado por parámetro, esta operación devuelve *verdadero* y *falso* en caso contrario.
- *esVacio*(): boolean
  // devuelve *falso* si hay al menos un elemento y *verdadero* en caso contrario.

Se pueden incluir otras operaciones útiles, como *clonar* y *vaciar*, como se ha hecho en los TDA anteriores. También es conveniente agregar la operación *toString* para permitir ver la estructura durante la etapa de debugging (esta operación deberá ser comentada o redefinida como privada antes de publicar la clase).

Aunque en su modo básico se restringe a las operaciones ya mencionadas, en un TDA Conjunto pueden ser deseables las operaciones de *unión*, *intersección* y *diferencia* entre conjuntos.

6.1.2. Implementaciones de TDA Conjunto

Para implementar el TDA Conjunto, la estructura más básica es un TDA Lista, usando las operaciones insertar, eliminar y localizar (pertenece) que son de orden O(n). Sin embargo, cuando se desea mejorar la eficiencia de esas operaciones básicas las estructuras más adecuadas, entre las vistas, serán Tabla Hash y árbol AVL.

Como ya se ha visto, la Tabla Hash tiene una eficiencia en las operaciones de búsqueda, inserción y eliminación de orden constante, O(1), cuando se cuenta con una función hash adecuada para el tipo de datos del *elemento* que se quiere mantener y una tabla de tamaño acorde para evitar múltiples colisiones. En el caso del árbol AVL, si bien la eficiencia es de O(log n) para operaciones de inserción, eliminación y existe, permite que el conjunto crezca y la eficiencia no se vea reducida por ello.

6.1.3. Bibliografía recomendada sobre TDA Conjunto

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988.

6.2. TDA Cola de Prioridad

6.2.1. Definición

Una Cola de Prioridad (en inglés "*priority queue*") es una colección de elementos donde cada uno de ellos tiene asociado un valor susceptible de ordenación denominado *prioridad*. Una cola de prioridad se caracteriza por admitir inserciones de elementos nuevos, de acuerdo a una prioridad asignada a él, pero sólo permite consultar y eliminar el elemento de mayor prioridad. Los elementos de igual prioridad son atendidos en orden de llegada.

Un ejemplo de aplicación en la vida real es el de la sala de emergencias de una guardia de hospital, donde las personas llegan y son atendidas de acuerdo a la gravedad de sus síntomas o heridas, y no sólo en el orden que han llegado a la guardia. Otro ejemplo es el de las cajas de los bancos y supermercados, que asignan mayor prioridad a mujeres embarazadas, personas acompañadas de niños pequeños, ancianos, etc. En informática, un ejemplo es el de los sistemas multi-usuario, que otorgan prioridades a los procesos según una jerarquía predeterminada de usuarios (donde el usuario *administrador* tiene la prioridad más alta).

Respecto al algoritmo de inserción, en una cola de prioridad, cuando llega un elemento nuevo, primero se mira su prioridad y luego su orden de llegada. En la Figura 6.2 se muestra una cola de prioridad que recibe elementos de 4 tipos de prioridad (p1=urgente >p2=semi-urgente >p3=normal >p4=no hay apuro). Suponiendo que en la cola ya se encuentran cargados dos elementos de prioridad p2 y dos elementos de prioridad p4, en caso de llegar un elemento con prioridad p1, este se deberá insertar por delante de los elementos de prioridad p2, para que el servidor lo atienda primero. En caso que llegue un elemento de prioridad p3, este deberá colocarse adelante de los elementos de prioridad p4 y detrás de los elementos de prioridad p2. Si llega un elemento de prioridad p2, este deberá colocarse por delante de los de prioridad menor (p3 y p4) y por detrás de los de su propia prioridad (p2).

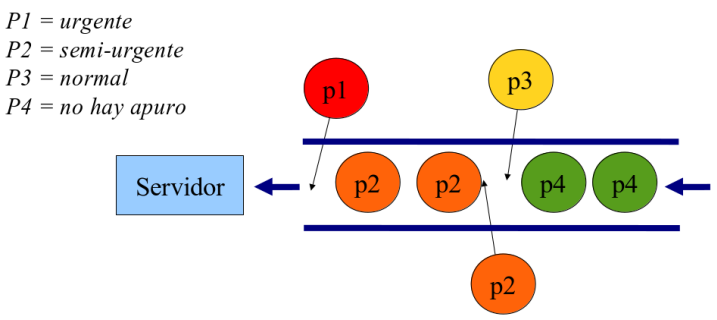

Figura 6.2: Orden de inserción en Cola de Prioridad

Respecto a consulta y eliminación, la cola de prioridad tiene el mismo funcionamiento de una Cola lineal, contando con operaciones equivalentes a *obtenerFrente* y *sacar* vistas anteriormente.

6.2.2. Operaciones del TDA Cola de Prioridad

Las operaciones básicas de TDA Cola de Prioridad son:

- *constructor vacío*:
  // crea una cola de prioridad sin elementos.
- *insertar* (elemento, prioridad): boolean
  // recibe por parámetro el elemento y la prioridad del mismo. Se agrega el elemento en la cola detrás de los de prioridad mayor o igual y por delante de los de prioridad menor. Si la operación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
- *eliminarFrente*(): boolean
  // elimina el elemento de mayor prioridad. Si hay más de uno con igual prioridad, elimina el que llegó primero. Si la operación de eliminación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
- *obtenerFrente*(): elemento
  // devuelve el elemento de mayor prioridad. Si hay más de uno con igual prioridad, devuelve el que llegó primero. *Precondición*: la cola no está vacía (si está vacía no se puede asegurar el funcionamiento de la operación).
- *esVacia*(): boolean
  // devuelve *falso* si hay al menos un elemento cargado en la estructura y *verdadero* en caso contrario.

Se pueden incluir otras operaciones útiles como *clonar* y *vaciar,* como se ha hecho en los TDA anteriores. También es conveniente agregar la operación *toString* para permitir ver la estructura durante la etapa de debugging (esta operación deberá ser comentada o redefinida como privada antes de publicar la clase).

6.2.3. Implementaciones

Para implementar el TDA Cola de Prioridad se debe elegir alguna estructura que permita tener acceso lo más eficientemente posible al elemento del frente (el de mayor prioridad y que haya llegado primero) y un acceso relativamente eficiente para la inserción de elementos nuevos con cualquier prioridad.

Entre las estructuras vistas anteriormente, el árbol Heap es el que permite el acceso más eficiente (O(1)) al elemento listo para ser atendido (el de la cima), mientras que para insertar y eliminar tiene una eficiencia de O(log n). Algo importante a tener en cuenta es que si los elementos del heap se ordenan solamente considerando su prioridad, el árbol Heap no asegura su atención por orden de llegada. Para ello será necesario ordenar los elementos por una clave compuesta por dos tributos (prioridad y orden de llegada).

El orden de llegada lo mantendrá la clase Cola de Prioridad de manera interna y transparente al usuario del TDA. El uso de un heap mínimo o máximo dependerá del tipoPrioridad usado. En general, se suele decir que una prioridad 1 es más urgente que una prioridad $>= 2$, por lo tanto en este caso, aunque se hable de "prioridad mayor", el heap debería ser mínimo. En la Figura 6.3 se presenta el diagrama de clases UML para esta implementación. Sin embargo, en algunos casos puede ser utilizada otra escala de prioridades que necesite que el heap sea máximo.

En la definición de orden entre dos elementos (CeldaCP) dentro del heap, primero deberá ser considerada la prioridad. Si esta es distinta, se define directamente cuál es mayor. Sin embargo, en caso que ambos elementos tengan igual prioridad, se deberá tomar el orden de llegada para definirlo.

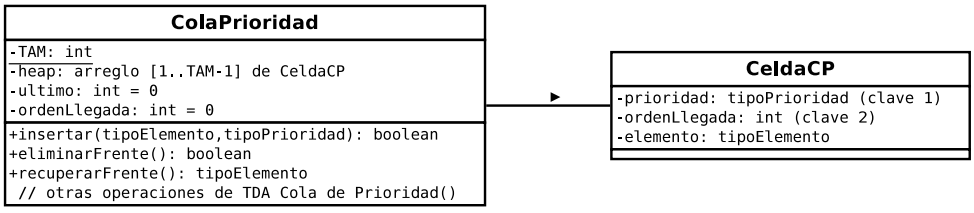

Figura 6.3: Diagrama UML de clases para Cola de Prioridad con árbol Heap

Otra implementación muy eficiente cuando hay pocas prioridades es utilizar una lista ordenada de colas. En esta implementación se utiliza una lista ordenada por prioridad donde para cada elemento de la lista se almacena, además de la prioridad (que es la que se toma como clave de ordenamiento), una cola simple (vista en el capítulo de Estructuras Lineales), para almacenar los elementos que tienen igual prioridad ordenados por orden de llegada.

Es importante que en esta implementación se mantengan los nodos de la lista ordenados por prioridad, para tener acceso en tiempo constante a la cola de elementos de mayor prioridad. También es importante que, al eliminar todos los elementos de una prioridad, se elimine también el nodo de dicha prioridad. De esta manera se asegura que el primer elemento a ser atendido esté siempre al frente de la cola del primer nodo de la lista.

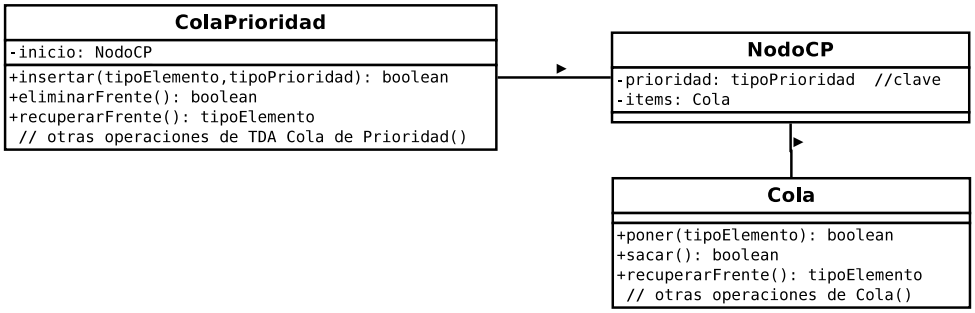

Figura 6.4: Diagrama UML de clases para Cola de Prioridad con árbol Heap

En la Figura 6.5 se presenta un ejemplo de una cola de prioridad que almacena los pacientes de una guardia, donde el paciente X llega con prioridad P, y se conoce su orden de llegada: A la izquierda, se presenta la implementación con árbol heap. Los elementos se han insertado considerando como clave primero la prioridad y luego el orden de llegada. A la derecha se muestra el mismo conjunto de pacientes almacenados en una cola de prioridad implementada con lista de colas. Como puede observarse en la figura, en este caso no es necesario almacenar el orden de llegada, dado que la estructura Cola de cada nodo lo mantiene por defecto.

Orden llegada	Nombre paciente	Prioridad
1	Tomás	2
2	Luca	4
3	Josefina	2
4	Victoria	1
5	Camila	4
6	Valentina	1
7	Mateo	2

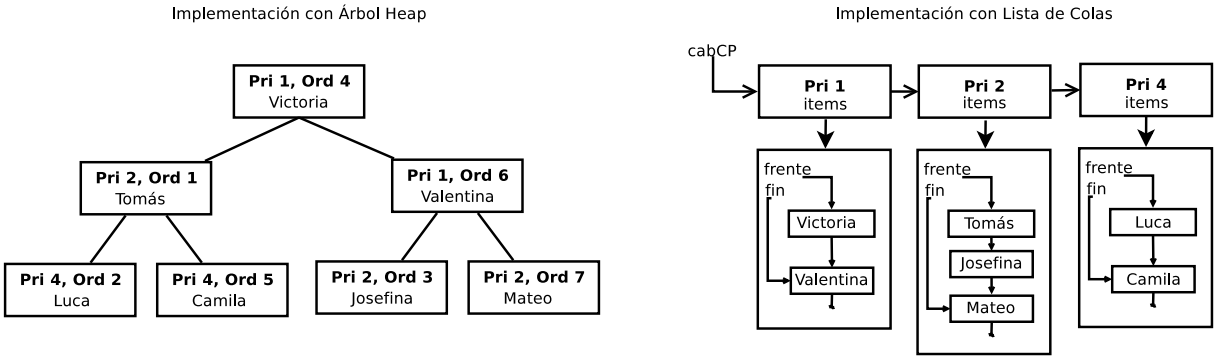

Figura 6.5: Representación de la misma cola de prioridad en las distintas implementaciones

Ejercicio 6.1: Implementaciones de TDA Cola de Prioridad

- Implementar el TDA Cola de Prioridad utilizando una de las implementaciones sugeridas (Árbol Heap o Lista de Colas).
- Realizar la clase de test correspondiente.

6.2.4. Bibliografía recomendada sobre Cola de Prioridad

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988. Sección 4.10.

6.3. TDA Mapeo

6.3.1. Definición

El tipo de dato abstracto Mapeo (llamado tambien "Correspondencia" según la traducción), representa a un tipo de estructura que permite relacionar valores de un conjunto de tipo dominio con uno o más valores de un conjunto tipo rango, como lo haría una función matemática. A diferencia de los grafos y árboles, que también representaban relaciones entre elementos, en un mapeo el tipo de los valores que relaciona no es necesariamente el mismo. En la Figura 6.6 se ve una representación de un mapeo donde un valor del conjunto de números enteros (dominio) se relaciona con un valor del conjunto de números naturales (rango). Los elementos que debe mantener el mapeo son duplas (valor dominio, valor rango). En el caso del ejemplo de la Figura 6.6, el mapeo M contiene las duplas {(-1,1), (1,1), (0,0), (2,4), (-3,9)}.

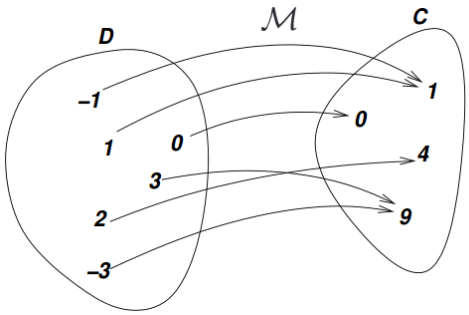

Figura 6.6: Representación de un mapeo con dominio $\mathbb{Z}$ y rango $\mathbb{N}$

Si la relación deseada se puede definir por medio de una función matemática (por ejemplo: valor absoluto, cuadrado de un número, raíz cuadrada, etc), conviene almacenar la función y calcularla para cualquier valor del dominio para conocer su correspondiente valor en el rango. Por el contrario, la idea del TDA Mapeo es permitir almacenar la relación entre valores aún cuando la función que los relaciona no sea matemática.

El mapeo más sencillo, que se llamará "Mapeo a Uno", considera que cada elemento del dominio se relaciona con un único elemento del rango (aunque un elemento del rango puede estar relacionado con varios elementos del dominio).

- *Ejemplo 1: Dominio*: persona ->*Rango*: país de nacimiento (muchas personas pueden haber nacido en el mismo país, pero cada persona ha nacido en un único país).
- *Ejemplo 2: Dominio*: empleado ->*Rango*: departamento donde trabaja.
- *Ejemplo 3: Dominio*: alumno ->*Rango*: ciudad de residencia.
- *Ejemplo 4: Dominio*: palabra en castellano ->*Rango*: su traducción en inglés.

La relación que representa el "Mapeo a Uno" puede ser mostrada como una tabla donde los valores independientes (dominio) se muestran en la columna izquierda, mientras que los valores dependientes (rango) se muestran en la columna derecha. En la siguiente tabla se muestra la tabla para un conjunto de palabras del ejemplo 4:

Dominio (valor independiente)	Rango (valor dependiente)
perro	dog
gato	cat
casa	house
vivienda	house

En un mapeo la operación más importante es poder obtener el valor asociado a un valor dado del dominio (por ejemplo, el valor asociado a *perro* en el ejemplo anterior es *dog*). Otras operaciones interesantes son aquellas que permiten consultar cuál es el *Conjunto Dominio* y cuál es el *Conjunto Rango* de la función. En el caso del *Conjunto Dominio*, se trata de los valores del dominio relacionados con algún conjunto del rango. En la tabla del ejemplo anterior los elementos del *Conjunto Dominio* son los que se encuentran en la primera columna: {perro, gato, casa, vivienda}. Por el contrario, el *Conjunto Rango* son los valores de la segunda columna, considerando una sola vez los valores repetidos {dog, cat, house}.

Otro tipo de mapeo es aquel que permite que los valores del dominio estén relacionados con más de un valor del conjunto rango. Este tipo de mapeo suele llamarse "Mapeo a Muchos". Ejemplos de aplicación de dicho tipo de mapeo son:

- *Dominio*: palabra ->*Rango*: una o más palabras que son sinónimos de la palabra dominio.
- *Dominio*: materia ->*Rango*: alumnos anotados para cursarla.
- *Dominio*: departamento ->*Rango*: empleados que trabajan allí.

La relación que representa el "Mapeo a Muchos" puede también ser mostrada como una tabla, donde la columna izquierda mantiene un único valor del dominio. Por el contrario, la columna derecha mantiene varios valores de tipo rango que se pueden pensar como subconjuntos o listas de dichos valores. En la siguiente tabla se muestra un ejemplo de un diccionario de sinónimos implementado con un Mapeo a Muchos.

Dominio (valor independiente)	Rango (valores dependientes)
casa	domicilio, hogar, vivienda, morada, residencia, piso, habitación, lar
mesa	escritorio, consola, tablero, mostrador
seleccionar	elegir, preferir, distinguir, separar, apartar, entresacar, aislar, nombrar, optar
preferir	inclinarse, querer, anteponer, seleccionar, elegir, distinguir, amar, mimar, favorecer, desear, optar

En este caso, la operación que devuelve el rango asociado a un valor de dominio debe devolver una lista o conjunto de datos. Respecto al *Conjunto Dominio*, como en el Mapeo a Uno, se trata del conjunto de los valores que se encuentran en la primera columna {casa, mesa, seleccionar, preferir}. En el caso del *Conjunto Rango* este será la unión de los pequeños conjuntos de la segunda columna {domicilio, hogar, vivienda, ..., favorecer, desear, optar}. Como se dijo anteriormente, en el caso del *Conjunto Rango* se consideran una sola vez los valores repetidos, mientras que en el caso del *Conjunto Dominio* no habrá valores repetidos por la propia definición de mapeo.

6.3.2. Operaciones del TDA Mapeo

Como se explicó antes, el propósito principal del tipo de dato abstracto *Mapeo* es permitir almacenar relaciones arbitrarias entre valores de dos conjuntos (tipo dominio y tipo rango) que pueden (o no) ser distintos. Como se ha mencionado anteriormente, podemos considerar dos variantes, respecto a si permiten un sólo valor de rango para cada valor de dominio (Mapeo a Uno) o más de uno (Mapeo a Muchos). A continuación se presentan las operaciones de ambos.

6.3.2.1. Operaciones del TDA Mapeo a Uno

Para el *"Mapeo a Uno"*, el elemento que guarda la estructura es un par o tupla de dos valores (dominio, rango). Las operaciones básicas de dicha estructura son:

- *constructor vacío*:
  // crea un mapeo sin elementos.

- *asociar* (valorDominio, valorRango): boolean
  // recibe un valor que representa a un elemento del dominio y un segundo valor que representa a un elemento del rango. Si no existe otro par que contenga a *valorDominio*, agrega en el mapeo el par (*valorDominio*, *valorRango*). Si la operación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
  Nota: Cada valor del dominio puede aparecer en un único par, es decir, no se aceptan valores del dominio repetidos.

- *desasociar*(valorDominio): boolean
  // elimina el par cuyo dominio coincida con el valor recibido por parámetro. Si lo encuentra y la operación de eliminación termina con éxito devuelve *verdadero* y *falso* en caso contrario.

- *obtenerValor*(valorDominio): elemento de tipo rango
  // si en el mapeo se encuentra almacenado algún par cuyo dominio es *valorDominio*, esta operación devuelve el valor de rango asociado a él. Precondición: *valorDominio* está en el mapeo (si no existe, no se puede asegurar el funcionamiento de la operación).

- *obtenerConjuntoDominio*(): Conjunto de elementos de tipo dominio
  // devuelve un conjunto con todos los valores de tipo dominio almacenados en el mapeo.

- *obtenerConjuntoRango*(): Conjunto de elementos de tipo rango
  // devuelve un conjunto con la unión de todos los valores de tipo rango almacenados en el mapeo.

- *esVacio*(): boolean
  // devuelve *falso* si hay al menos un par cargado en el mapeo y *verdadero* en caso contrario.

Se pueden incluir otras operaciones útiles como *clonar* y *vaciar,* como se ha hecho en los TDA anteriores. También es conveniente agregar la operación *toString* para permitir ver la estructura durante la etapa de debugging (esta operación deberá ser comentada o redefinida como privada antes de publicar la clase).

6.3.2.2. Operaciones del TDA Mapeo a Muchos

Para el *"Mapeo a Muchos"*, el elemento que guarda la estructura es un par o tupla de dos valores (*dominio*, *{rango1, rango2, ...}*). Las operaciones básicas de dicha estructura son similares a las de Mapeo a Uno, exceptuando las siguientes operaciones que modifican su funcionalidad de manera acorde:

- *asociar* (valorDominio, valorRango): boolean
  // recibe un valor que representa a un elemento del dominio y un segundo valor que representa a un elemento del rango. Si no existe otro par que contenga a *valorDominio*, se agrega en el mapeo el par (*valorDominio*, *{valorRango}*), donde el segundo término es un conjunto de rangos con un solo valor. Si ya existe un par con *valorDominio*, se agrega *valorRango* al conjunto de rangos existente. Si la operación termina con éxito devuelve *verdadero* y *falso* en caso contrario.

- *desasociar*(valorDominio, valorRango): boolean
  // elimina *valorRango* del conjunto de rangos asociado a *valorDominio*. Caso especial: si al eliminar *valorRango* del conjunto este quedara vacío, debe eliminar el par *(valorDominio,{})* del mapeo. Si encuentra el par y la operación de eliminación termina con éxito devuelve *verdadero* y *falso* en caso contrario.

- *obtenerValores*(valorDominio): Conjunto de elemento de tipo rango
  // si en el mapeo se encuentra almacenado algún par cuyo dominio es *valorDominio*, devuelve el conjunto de valores de rango asociado a él. Precondición: valorDominio está en el mapeo (si no existe, no se puede asegurar el funcionamiento de la operación).

6.3.3. Implementaciones

Para implementar el TDA Mapeo se utilizarán algunas de las estructuras vistas en las unidades anteriores. Dado que las operaciones más importantes (*asociar, desasociar y obtenerValor*) requieren un acceso eficiente para un valor dado *dominio*, las estructuras más adecuadas serán la Tabla Hash o el árbol AVL (que se han visto en la Unidad de Estructuras Conjuntistas).

Como se ha visto antes, la Tabla Hash permite una eficiencia en las operaciones de búsqueda, inserción y eliminación de orden constante u O(1), cuando se cuenta con una función hash adecuada para el tipo de datos que se almacena como clave (en este caso el tipo dominio). Luego, para poder almacenar un valor rango (en Mapeo a Uno) o un conjunto de valores rangos (en el caso del Mapeo a Muchos), asociados

a cada valor dominio, se deberá modificar de manera adecuada la estructura vista para Tabla Hash. Por ejemplo, para la implementación de hash abierto, se debe modificar la clase Nodo para almacenar dos valores (dominio y rango) en lugar del tipo elemento. A la clase Nodo modificada se la ha llamado NodoHashMapeo. En la clase MapeoAUno la función hash deberá estar definida sólo para el tipo del dominio. Las operaciones que devuelven un conjunto de elementos se han implementado utilizando el tipo Lista visto en la Unidad "Estructuras Lineales" aunque podría ser reemplazado por otra estructura acorde.

El diagrama de clases correspondiente se presenta en la Figura 6.7.

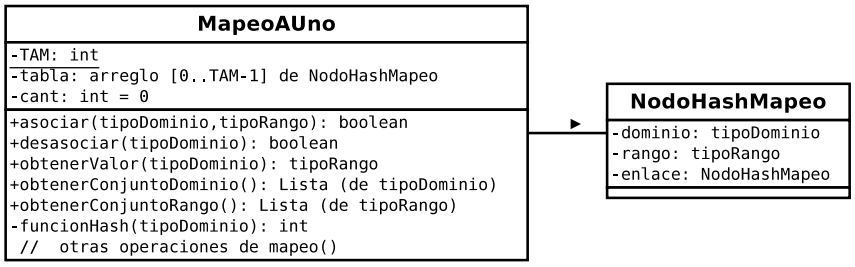

Figura 6.7: Diagrama UML de clases para MapeoAUno con hash abierto

Otra posibilidad para almacenar los pares (dominio, rango) del Mapeo y recuperarlos eficientemente es utilizar un árbol AVL, que como se ha visto permite una eficiencia en las operaciones de búsqueda, inserción y eliminación de O(log n). Para ello, es necesario modificar de manera adecuada el nodo del árbol AVL para almacenar, además del dominio, un valor rango (en Mapeo a Uno) o un conjunto de valores rango (en el caso del Mapeo a Muchos). Por ejemplo, en la Figura 6.8 se muestra el diagrama de clases para la implementación de MapeoAUno utilizando una clase NodoAVL modificada, a la cual se ha llamado NodoAVLMapeo.

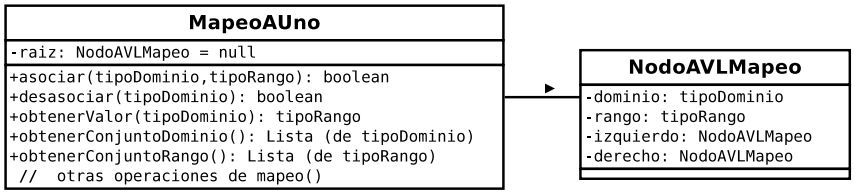

Figura 6.8: Diagrama UML de clases para MapeoAUno con árbol AVL

En la Figura 6.9 se presenta un ejemplo de un mapeo que almacena una palabra en castellano (dominio) e inglés (rango) implementado de las dos maneras presentadas anteriormente: A la izquierda, se presenta la implementación con hash abierto. Los elementos se han insertado de acuerdo a una función hash arbitraria sobre el valor *dominio* del par. A la derecha se muestra el mismo conjunto de pares almacenados en un árbol AVL. En este caso, es el orden de los elementos en los nodos el que se ha definido sobre la palabra *dominio*.

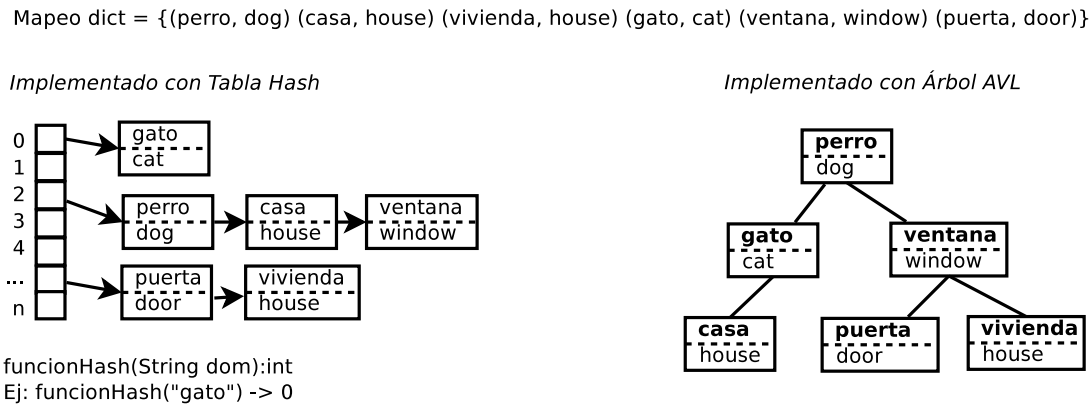

Figura 6.9: Representación de un Mapeo en las distintas implementaciones

Respecto al Mapeo a Muchos, la diferencia es que al dominio se le asocia una lista de rangos a la que se crea con un elemento (cuando el dominio no estaba asociado a otros rangos) o se agrega a la lista de

rangos existente. El diagrama de clases correspondiente a la implementación con Tabla Hash se presenta en la Figura 6.10 y con Árbol AVL en la Figura 6.11.

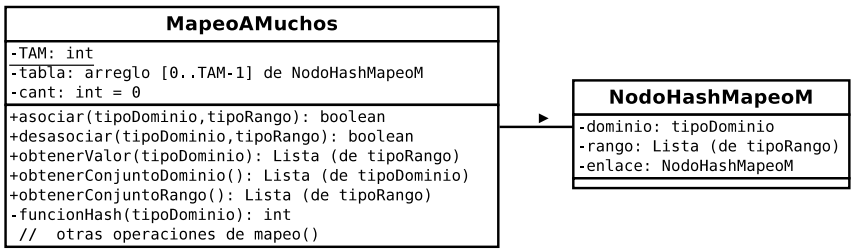

Figura 6.10: Diagrama UML de clases para MapeoAMuchos con hash abierto

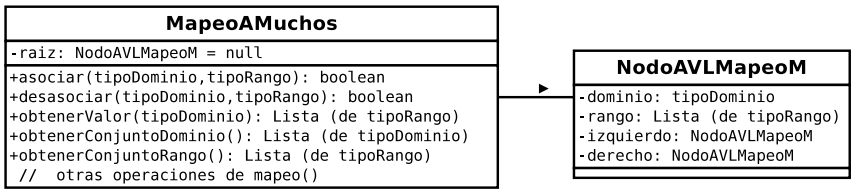

Figura 6.11: Diagrama UML de clases para MapeoAMuchos con árbol AVL

En la Figura 6.12 se presenta un ejemplo de un mapeo a muchos que almacena el legajo de un alumno (dominio) y las materias que está cursando (rango), implementado de las dos maneras presentadas anteriormente: A la izquierda, se presenta la implementación con hash abierto. Los elementos se han insertado de acuerdo a una función hash arbitraria sobre el valor *dominio*. A la derecha se muestra el mismo conjunto de pares almacenados en un árbol AVL. En este caso, es el orden de los elementos en los nodos el que se ha definido sobre el legajo *(dominio)*.

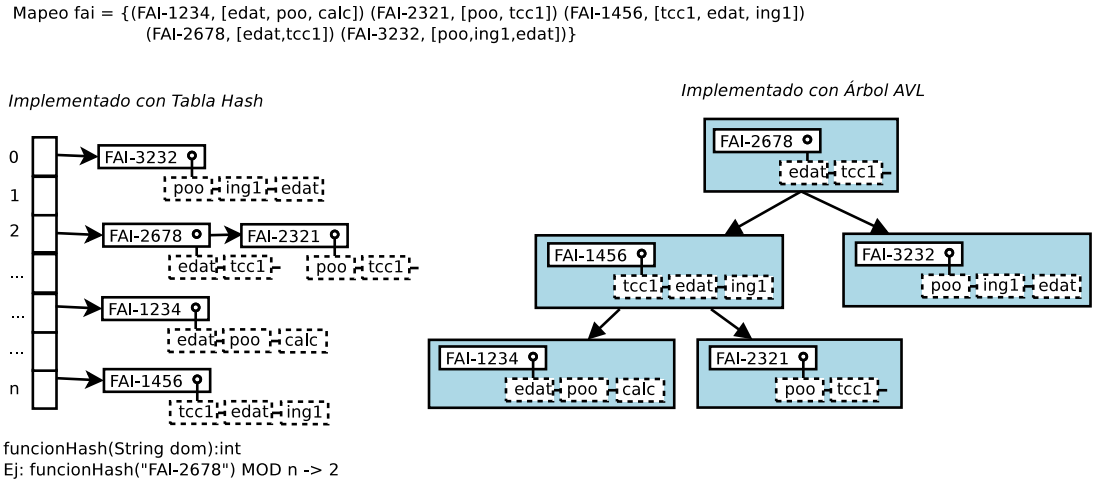

Figura 6.12: Representación de un Mapeo en las distintas implementaciones

---

Ejercicio 6.2: Implementación de TDA Mapeo

- Implementar el TDA Mapeo utilizando una de las implementaciones sugeridas (Tabla Hash o Árbol AVL).
- Realizar la clase de test correspondiente.

6.3.4. Análisis de eficiencia de las implementaciones

Como se ha mencionado, la eficiencia de las operaciones del TDA Mapeo dependerán de la eficiencia de la estructura elegida para almacenar los pares (dominio, rango). Las estructuras ideales son aquellas que permitan realizar inserciones, búsquedas y eliminaciones por medio de un valor clave, que en este caso será siempre un valor del tipo del dominio. Las estructuras vistas que sirven para este objetivo son Tabla Hash y árbol AVL. En el caso de Tabla Hash, la eficiencia dependerá de la función hash elegida y se espera que esta sea de O(1), mientras que en el árbol AVL la eficiencia dependerá de la cantidad de niveles del árbol, que al mantenerse balanceado, es siempre de O(log n).

6.3.5. Bibliografía recomendada sobre Mapeo

- A.V. Aho, J.E. Hopcroft y J.D. Ullman, Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988. (Presentado como TDA Correspondencia)
- M. Storti, J. D'Elía, R. Paz, L. Dalcín y M. Pucheta: Algoritmos y Estructuras de Datos, Apuntes de la cátedra. Universidad Nacional del Litoral. Disponible en http://www.cimec.org.ar/~mstorti/aed/aednotes.pdf (Presentado como TDA Correspondencia)

6.4. TDA Diccionario o Tabla de Búsqueda

6.4.1. Definición

El TDA *diccionario* o *tabla de búsqueda* es un conjunto de elementos que están formados por una clave e información asociada a ella. La restricción principal es que las claves deben ser únicas.

Las operaciones principales de un TDA diccionario son las de creación de un diccionario vacío, buscar una clave para determinar si está o para obtener su información relacionada, insertar un nuevo elemento y borrar un elemento dado del diccionario. Como en los diccionarios de la vida real, la operación más importante de un TDA diccionario será la de consulta o búsqueda, por lo que se procurará la máxima eficiencia para esta operación. También se tratará de que sean eficientes las operaciones de inserción y eliminación, aunque es probable que no sean utilizadas tan a menudo como las consultas.

Este tipo de estructura es el más utilizado en la implementación de sistemas de bases de datos, para almacenar grandes cantidades de elementos que se identifican por una clave única, por ejemplo:

- *Los estudiantes de una facultad*: La *clave* de cada estudiante es su *legajo* y la información relacionada son los datos propios del estudiante, por ejemplo su nombre, apellido, DNI, fecha de nacimiento, etc.
- *El padrón de electores de un país*: En este caso la *clave* es el *documento* (generalmente formado por una clave compuesta tipo + número de documento, por ejemplo LE 6908565, DNI 24315224). La información asociada sería, además del documento, su género, nombre, apellido, fecha de nacimiento, domicilio, etc.
- *Los usuarios de una red social:* Dado un sistema tipo Facebook o Twitter, la *clave* única es la *dirección de email* con la que se registró el usuario y la información adicional son su contraseña, nombre, y toda otra información que forme parte del perfil del usuario.

6.4.2. Operaciones del TDA

Como se explicó antes, el propósito principal del *TDA Diccionario* es permitir almacenar elementos formados por una clave única e información asociada a ella, es decir información del tipo (clave, información). Las operaciones básicas de dicha estructura son:

- *constructor vacío*:
  // crea una estructura sin elementos.
- *insertar* (clave, dato): boolean
  // recibe la *clave* que es única y el *dato* (información asociada a ella). Si no existe en la estructura un elemento con igual clave, agrega el par (clave, dato) a la estructura. Si la operación termina con éxito devuelve *verdadero* y *falso* en caso contrario.
- *eliminar*(clave): boolean
  // elimina el elemento cuya clave sea la recibida por parámetro. Si lo encuentra y la operación de eliminación termina con éxito devuelve *verdadero* y *falso* en caso contrario.

- *existeClave*(clave): boolean
  // devuelve *verdadero* si en la estructura se encuentra almacenado un elemento con la *clave* recibida por parámetro, caso contrario devuelve *falso*.
- *obtenerInformacion*(clave): tipo dato
  // si en la estructura se encuentra almacenado un elemento con la *clave* recibida por parámetro, devuelve la información asociada a ella. Precondición: si no existe un elemento con esa clave no se puede asegurar el funcionamiento de la operación.
- *listarClaves* (): Lista (de tipo de la clave)
  // recorre la estructura completa y devuelve una lista ordenada con las claves de los elementos que se encuentran almacenados en ella.
- *listarDatos* (): Lista (de tipo dato)
  // recorre la estructura completa y devuelve una lista ordenada con la información asociada de los elementos que se encuentran almacenados en ella.
- *esVacio*(): boolean
  // devuelve *falso* si hay al menos un elemento cargado en la estructura y *verdadero* en caso contrario.

Se pueden incluir otras operaciones útiles como *clonar* y *vaciar,* como se ha hecho en los TDA anteriores. También es conveniente agregar la operación *toString* para permitir ver la estructura durante la etapa de debugging (esta operación deberá ser comentada o redefinida como privada antes de publicar la clase).

6.4.3. Implementaciones

Para implementar el TDA Diccionario, dado que las operaciones más importantes (*insertar*, *eliminar,* *existeClave* y *obtenerInformación*) requieren un acceso eficiente para un valor *clave* dado, las estructuras más adecuadas serán Tabla Hash y árbol AVL (vistas en la Unidad de Estructuras Conjuntistas).

Como ya se ha visto, la Tabla Hash tiene una eficiencia en las operaciones de búsqueda, inserción y eliminación de orden constante, O(1), cuando se cuenta con una función hash adecuada para el tipo de datos que se almacena como *clave*. Luego, se deberá modificar la estructura de Tabla Hash de manera adecuada para permitir almacenar la información adicional de cada clave. Por ejemplo, para la implementación de hash abierto, se debe modificar la clase Nodo para almacenar, en lugar del elemento, el par (clave, información adicional). A la clase Nodo modificada se la ha llamado NodoHashDicc.

El diagrama de clases correspondiente se presenta en la Figura 6.13.

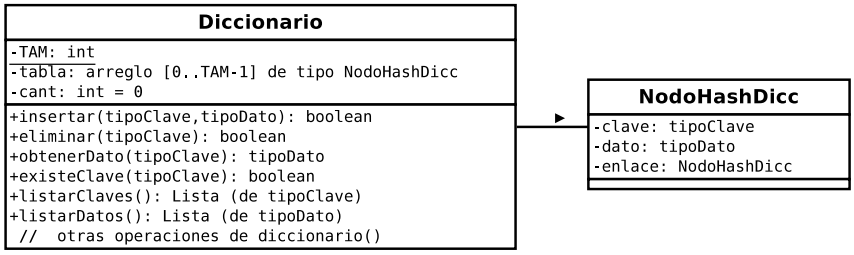

Figura 6.13: Diagrama UML de clases para TDA Diccionario implementado con hash abierto

Otra posibilidad para almacenar los pares (clave, información adicional) y recuperarlos eficientemente es utilizar (como se hizo antes con Mapeo), un árbol AVL, que como se ha visto permite una eficiencia en las operaciones de búsqueda, inserción y eliminación de O(log n). Para ello, es necesario modificar de manera adecuada el nodo del árbol AVL para almacenar, además de la clave, la información adicional. Por ejemplo, en la Figura 6.14 se muestra el diagrama de clases utilizando una clase NodoAVL modificada, a la cual se ha llamado NodoAVLDicc.

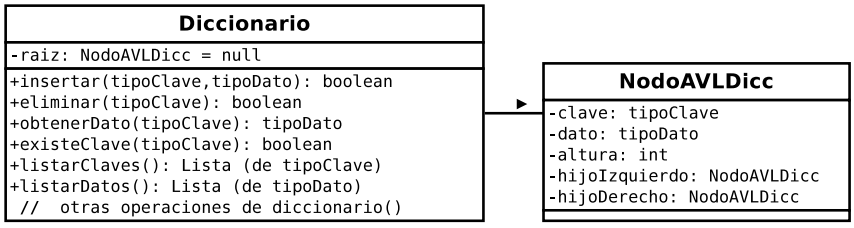

Figura 6.14: Diagrama UML de clases para TDA Diccionario implementado con árbol AVL

En la Figura 6.15 se presenta un ejemplo de un diccionario que almacena los datos de un conjunto de alumnos implementado de las dos maneras presentadas anteriormente: A la izquierda, se presenta la implementación con hash abierto. Los elementos se han insertado de acuerdo a una función hash arbitraria sobre el valor *legajo* que es la clave de cada alumno. A la derecha se muestra el mismo conjunto de alumnos almacenado en un árbol AVL. En este caso el orden de los elementos en los nodos el que se ha definido sólo sobre la clave *legajo*. Es importante notar que el orden de las claves de tipo String está definido lexicográficamente, es decir de acuerdo al valor ASCII de cada caracter. Luego, el legajo "FAI-23" es mayor que "FAI-123", dado que la subcadena "FAI-" es igual en ambas cadenas, al comparar el valor ASCII de los caracteres en la posición 4 de cada String, es mayor el ASCII del caracter '2' (ASCII 50) que el del caracter '1' (ASCII 49). Por ese motivo el elemento con clave "FAI-123" queda a la izquierda del elemento con clave "FAI-23".

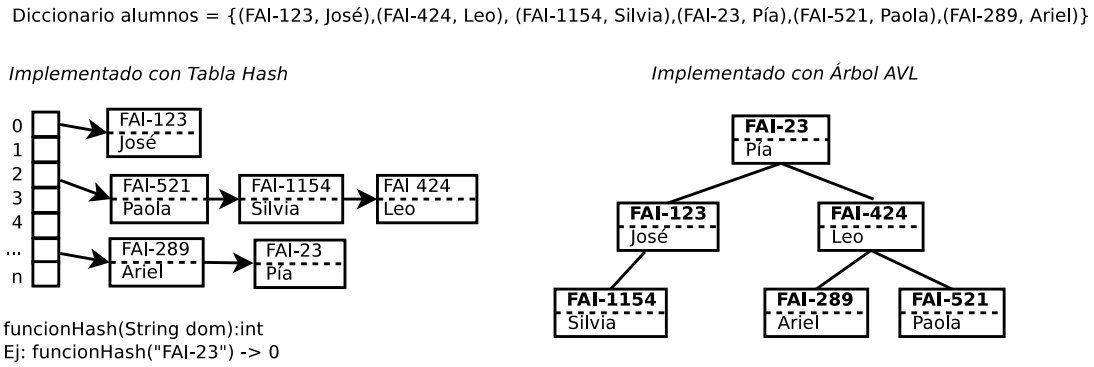

Figura 6.15: Representación de un diccionario en las distintas implementaciones

---

Ejercicio 6.3: Implementaciones de TDA Tabla de Búsqueda o Diccionario

- Implementar el TDA Tabla de Búsqueda o Diccionario utilizando una de las implementaciones sugeridas (Tabla Hash o Árbol AVL).
- Realizar la clase de test correspondiente.

6.4.4. Análisis de eficiencia de las implementaciones

La eficiencia de las operaciones del TDA Diccionario dependerá de la eficiencia de la estructura elegida para almacenar los pares (clave, dato). Las estructuras ideales son aquellas que permiten realizar inserciones, búsquedas y eliminaciones por medio de un valor que define cada elemento de manera unívoca, llamado *clave*. Las estructuras vistas en la unidad "Estructuras Conjuntistas" que cumplen este objetivo son Tabla Hash y árbol AVL. En el caso de Tabla Hash, la eficiencia dependerá de la función hash elegida y se espera que esta sea de O(1). Como desventaja de esta implementación podemos mencionar dos aspectos principales:

- *La elección de la función hash*: Cuando la función hash devuelve el mismo valor para varios elementos, ocasiona que estos colisionen a menudo, haciendose necesario aplicar una estrategia de rehashing en la implementación de hash cerrado, o que se creen listas más largas de lo deseado en la implementación de hash abierto. En ambos casos, la eficiencia de las operaciones deja de ser constante y empieza a parecerse a un orden lineal.
- *El tamaño máximo de la tabla*: El segundo problema es cuando en hashing cerrado se utiliza un tamaño de tabla insuficiente para los elementos que se deben almacenar. En este caso será necesario crear una tabla más grande y volver a cargar cada elemento en la estructura nueva.

En conclusión, la Tabla Hash será una buena implementación del TDA Diccionario siempre que la función *hash* sea adecuada para el conjunto de claves a almacenar (función uniforme) y que la cantidad de elementos no crezca demasiado a lo largo del tiempo para que no se llene la estructura y sea necesario redefinir la tabla.

Respecto a la implementación del TDA Diccionario con un árbol AVL, la eficiencia dependerá de la cantidad de niveles del árbol, que al mantenerse balanceado, es siempre de O(log n). Como desventaja se puede mencionar la complejidad de la implementación de las rotaciones, aunque es un costo menor comparado con la ganancia en eficiencia al almacenar conjuntos grandes de elementos, y que al ser una estructura dinámica puede crecer tanto como sea necesario. En conjuntos de miles de elementos, la cantidad de niveles de un árbol AVL puede hacer un poco más lento el tiempo de acceso en esta estructura en comparación con otras más avanzadas (árboles B, 2-3, Trie, etc.), que se verán en cursos más avanzados.

6.4.5. Bibliografía recomendada sobre Tabla de Búsqueda

- A.V. Aho, J.E. Hopcroft y J.D. Ullman: Estructuras de Datos y Algoritmos, Addison Wesley Iberoamericana, 1988.

6.5. ¿Qué TDA conviene usar?

En esta asignatura vimos muchas estructuras y diferentes formas de implementarlas. Pero cada estructuras puede tener múltiples usos, y por otro lado, no se deberían usar siempre las mismas estructuras, sino evaluar cada caso y elegir la más adecuada según los requerimientos del dominio en particular.

Ejercicio 6.4: Responder brevemente

1. ¿Qué tipo de relación entre elementos define un TDA Mapeo?
2. ¿Cuáles son las operaciones básicas del TDA Mapeo?
3. ¿En qué se parecen y en qué se diferencian el TDA Mapeo a Uno y el TDA Diccionario o Tabla de Búsqueda?
4. ¿Qué tipo de relación hay entre los pares que almacena un TDA Mapeo a Uno y los valores que almacena el TDA Diccionario?
5. ¿Cuándo es una buena elección una Tabla Hash para implementar el TDA Diccionario? ¿Cuándo no lo es?
6. Si se desea implementar un TDA Diccionario para almacenar los alumnos de una universidad ¿qué implementación será más adecuada: Tabla Hash o árbol AVL?
7. ¿Dado que ambos son árboles balanceados, es lo mismo usar un árbol AVL que un árbol Heap para implementar el TDA Diccionario?
8. ¿Sería posible implementar el TDA Cola de Prioridad usando un árbol AVL? Compare la eficiencia de las operaciones en dicha implementación con las implementaciones sugeridas en este capítulo (Heap y Lista de Colas).
9. ¿Por qué al implementar Cola de Prioridad con árbol Heap es necesario agregar el orden de llegada y en la implementación con Lista de Colas no?
