Estrategias de Programación y Estructuras de Datos

Dpto. de Lenguajes y Sistemas Informáticos

Material permitido: NINGUNO. Duración: 2 horas

Alumno:	**D.N.I.:**
C. Asociado en que realizó la Práctica Obligatoria:	

Este documento detalla una posible solución propuesta por el Equipo Docente con intención didáctica antes que normativa. Nótese que el nivel de detalle de este documento no es el pretendido para los ejercicios realizados por los alumnos en la solución de sus exámenes.
Por brevedad, no se incluyen las interfaces de los tipos, que el alumno puede consultar en la documentación de la asignatura.

P1 Práctica. Las tesis doctorales se evalúan y califican por medio de un tribunal de tesis, compuesto por varios doctores expertos en el área. Se quiere extender la práctica de la asignatura de manera que se almacenen los tribunales de tesis de todos los doctores de la academia, bajo el supuesto de que todos los miembros de los tribunales son doctores de la academia. Se pide:

a) (0'5 puntos) ¿Qué modificaciones habría que realizar para almacenar los tribunales de tesis de todos los doctores de la academia?

En cuanto a la representación de un tribunal de tesis: Puesto que no se especifican relaciones de jerarquía, no tiene sentido usar árboles para almacenar un tribunal de tesis. Por otro lado, el almacenamiento de los tribunales de tesis no se ajusta a las políticas FIFO o LIFO, de manera que no tiene sentido utilizar pilas o colas. Por tanto, un tribunal de tesis puede almacenarse mediante una lista o un conjunto. En particular, si se usa un conjunto, se evita que la estructura que almacena el tribunal de tesis contenga repeticiones. La estructura escogida (lista o conjunto) puede contener o las referencias a los doctores que forman parte del tribunal (`ListIF<DoctorIF>` o `SetIF<DoctorIF>`) o los identificadores de estos (`ListIF<Integer>` o `SetIF<Integer>`), puesto que identifican de manera unívoca a los doctores de la academia.

Puede haber diferentes maneras de almacenar todos los tribunales de tesis de la academia. A continuación se describen dos opciones:

A) Cada doctor guarda su tribunal de tesis en un nuevo atributo.
B) La academia guarda un registro de todos los tribunales de tesis.

Para la opción A, los cambios a realizar son los siguientes:

- Nuevo atributo en DoctorIF denominado "committee". El tipo de este atributo se correspondería con alguna de las cuatro opciones descritas anteriormente.
- Nuevo constructor: Se requiere un nuevo constructor de la clase `DoctorIF` que reciba un tribunal, y lo asigne al nuevo atributo.
- Métodos `addDoctor` en `AcademiaS` y `AcademiaC`: recibe como nuevo parámetro el tribunal de tesis del doctor que va a incluirse en la academia.
- Nuevo/s método/s de consulta del tribunal de tesis del doctor.

En el caso de la opción B, sería conveniente crear una nueva clase `Committee` que almacene tanto al doctor como a su tribunal de tesis. Esta nueva estructura tendría dos atributos: Uno de ellos identifica al doctor (por ejemplo, su `ID`, de tipo entero), y el otro almacena su tribunal de tesis

(alguna de las cuatro opciones descritas atrás). Se debería implementar constructores y métodos consultores de esta nueva clase. La academia tendría un nuevo atributo register que representa el registro de tribunales de tesis de la academia y que puede representarse mediante una lista, i.e. `ListIF<Committee>`.

Los cambios a realizar serían los siguientes:

- Nuevo atributo en `AcademiaIF` que represente el registro de tribunales de tesis representado mediante una lista de objetos de tipo `Committe`.
- El constructor de `AcademiaIF` debe inicializar el registro de tribunales de tesis como una lista vacía.
- Métodos `addDoctor` en `AcademiaS` y `AcademiaC`: recibe como nuevo parámetro el tribunal de tesis del doctor que va a incluirse en la academia. Añadir esta información al registro de tribunales de tesis.
- Nuevo método de consulta del tribunal de tesis de un doctor, dado su `ID`.

b) (1'5 puntos) Se quiere extender la interfaz `DoctorIF` con una nueva operación, que llamaremos `numThesisCommittees()` que devuelve el número de tribunales de tesis en los que ha participado el doctor. Describir cómo implementaría esta función usando la representación de los TAD que ha utilizado en su práctica (no es necesario que implemente la función). Nótese que dicha función debe comportarse de la misma manera en los dos escenarios propuestos en la práctica.

Siguiendo la metodología de las operaciones de la práctica, el doctor debe consultar en su academia en cuantos tribunales de tesis ha participado. Por tanto, la función pedida debe llamar a una función de perfil `public int numThesisCommittees(int ID)` de `AcademiaIF`, tomando como parámetro su propio `ID`. En el caso de la opción A, la función `numThesisCommittees (int ID)` mantendría un contador inicialmente con valor 0. Recorrería todos los doctores de la academia. Para cada uno de ellos, se extrae su tribunal y en caso de que el `ID` dado por parámetro perteneciese a alguno de los doctores del tribunal, se incrementaría el contador en una unidad. Finalmente, se devolvería dicho contador. En el caso de la opción B, la función `numThesisCommittees(int ID)` mantendría un contador inicialmente con valor 0. La academia recorrería el registro de tesis doctorales. Se tomaría el tribunal de cada entrada del registro, y en caso de que el `ID` dado por parámetro perteneciese a alguno de los doctores del tribunal, se incrementaría el contador en una unidad. Finalmente, se devolvería dicho contador.

1. Responda a las siguientes cuestiones sobre árboles AVL:

   a) (1 punto) Defina qué es un árbol AVL y qué propiedades cumple.

   Un árbol AVL es un tipo de árbol de búsqueda binaria autobalanceable que cumple la siguiente condición de equilibrio: la diferencia entre las alturas de su hijo izquierdo y su hijo derecho no es superior a 1 (en valor absoluto). Además, los hijos izquierdo y derecho de un árbol AVL deben ser también árboles AVL. Al tratarse de un tipo especial de árboles de búsqueda binaria (ABB o BST en inglés), cumplen las condiciones de este tipo de árboles: son árboles binarios, el elemento raíz es mayor que todos los elementos del hijo izquierdo, y menor que todos los elementos del hijo derecho. Este tipo de árboles aplican operaciones de reequilibrado, denominadas rotaciones AVL o simplemente rotaciones, cuando tras añadir o eliminar un elemento del árbol se incumple la condición de equilibrio descrita anteriormente.

   b) (2 puntos) Dibujar el árbol AVL de enteros resultante tras insertar consecutivamente los números de la siguiente secuencia: 7, 3, 9, 2, 1, 6, 5, 10, 4, 11, 12, 13. Detalle las rotaciones aplicadas.

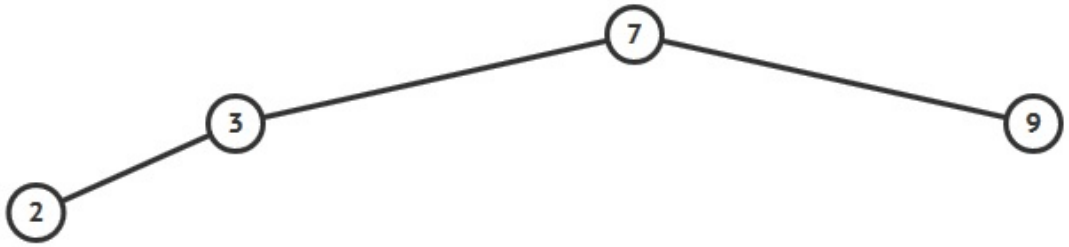
   Mientras se construye el árbol AVL, se ha de tener en cuenta lo siguiente:

   A) La inserción debe respetar las condiciones de los BSTs: si el elemento es menor que la raíz, estará en el hijo izquierdo, y en caso de ser mayor, irá al hijo derecho.
   B) En caso de incumplirse la condición de equilibrio de los árboles AVL, se debe aplicar la operación de rotación correspondiente (consultar Texto recomendado).

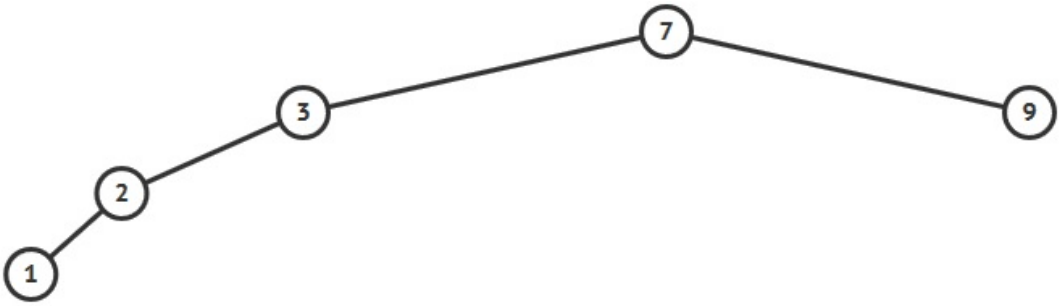
A continuación, se dan todos los detalles de la construcción del árbol AVL pedido por razones didácticas. Adviértase que esta solución no pretende ser normativa sino que intenta explicar el proceso paso a paso. Tampoco es la solución que se pretende constituya la respuesta de los estudiantes[1]:

Recordemos la secuencia: $\{7, 3, 9, 2, 1, 6, 5, 10, 4, 11, 12, 13\}$.

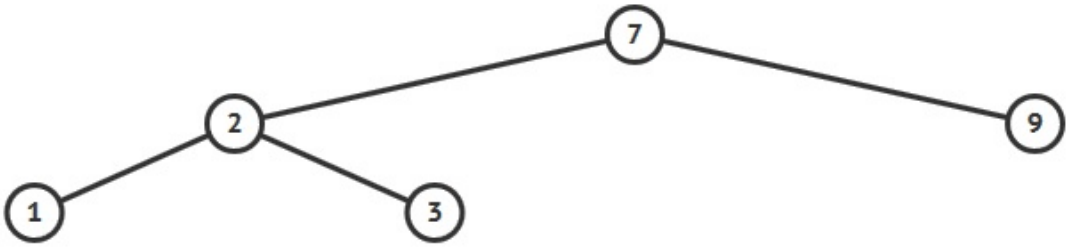
Tras insertar los números 7, 3, 9, 2 teniendo en cuenta (A), obtenemos el siguiente árbol AVL:



Tras insertar el valor 1 se incumple la condición de equilibrio[2], ya que, si nos fijamos en el subárbol izquierdo (cuya raíz es 3), la altura (número de aristas que forman el camno entre un nodo y su hoja más lejana) de *su* subárbol izquierdo es 2 mientras que la de su subárbol derecho es 0.



Por lo tanto, hay que aplicar una rotación simple (en este caso, tomando 2 como nueva raíz del subárbol y ubicando 3 como su hijo derecho). El resultado tras aplicar la rotación es el siguiente (nótese que, ahora, todos los subárboles están equilibrados conforme a la condición AVL, es decir, si calculamos los factores de equilibrio de cada nodo, ninguno supera 1 en valor absoluto):

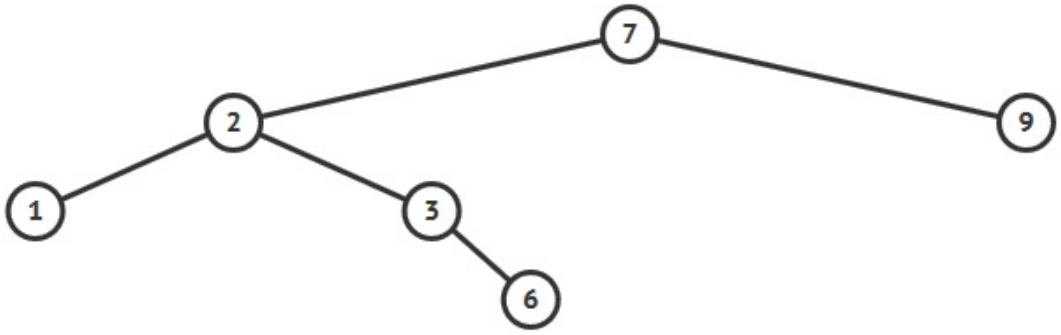


A continuación se inserta el valor 6, que deberá colocarse como hijo derecho de 3 (dado que es mayor que éste pero menor que la raíz, 7). Esto provocará, de nuevo, el incumplimiento de la

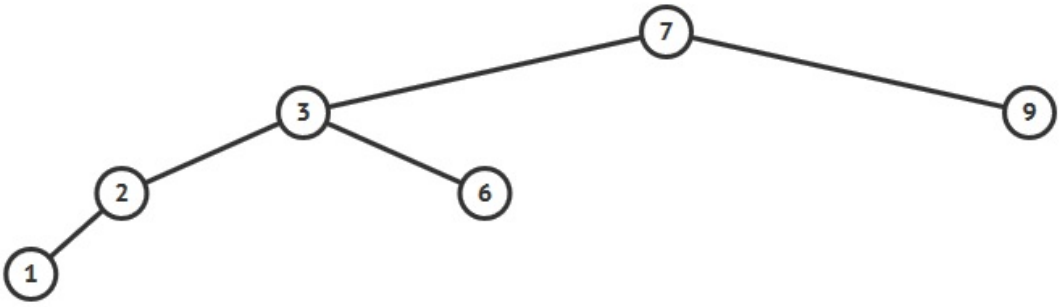
[1] En el enlace adjunto, que se abrirá en su navegador *favorito*, puede encontrarse una herramienta de simulación de árboles AVL.Debe seleccionarse `AVL` en la parte superior de la interfaz, `Create` (y luego `Empty` para un árbol vacío) en la parte inferior izquierda, para luego seleccionar la opción `Insert`, que abre un campo de texto donde puede introducirse una serie de valores separados por comas. Para lanzar la simulación, debe pulsarse en `Go`. La velocidad de la animación y otros parámetros son configurables y la simulación va indicando los pasos que realiza

[2] El primer vértice o nodo donde sucede esto es en el 3. Si se calculan las alturas de todos los nodos tras insertar el 1, serían 1(0), 2(1), 3(2), 7(3), 9 (0). Si, ahora, calculamos la diferencia de alturas entre hijos izquierdos y derechos de cada nodo serían (en **negrita**, implican **desequilibrio**) 1(0), 2(+1), 3(**+2**), 7(**+3**), 9(0). Nótese que estas diferencias se calculan desde las hojas a la raíz y que, una vez que se encuentra un desequilibrio de la regla AVL, hay que resolverlo sin que importe cómo se propagaría, ya que su solución es siempre posible a distancia 1 o 2 del nodo que presenta dicho primer desequilibrio

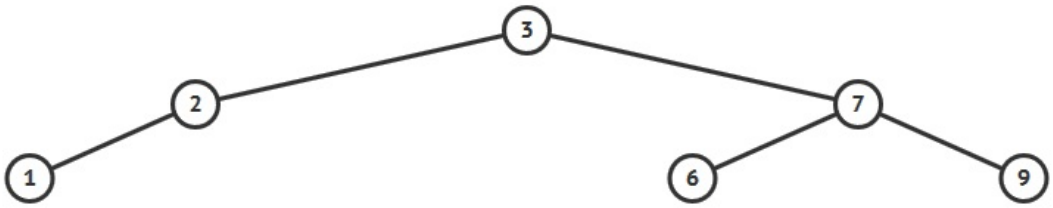
condición de equilibrio de los árboles AVL. El desequilibrio se producirá, esta vez, en la raíz, cuyo hijo izquierdo tras la inserción del nuevo nodo (el subárbol izquierdo sin contar la raíz, es decir, aquél que tiene 2 como raíz) tiene altura 2 mientras que su hijo derecho (que comprende tan solo la hoja 9) de altura 0. Si calculamos de nuevo las alturas y factores de equilibrio para cada nodo, obtendremos los siguientes valores: 1 (0,0), 2(1,1), 3(2,1), 6(0,0), 7(3,**2**), 9(0,0).



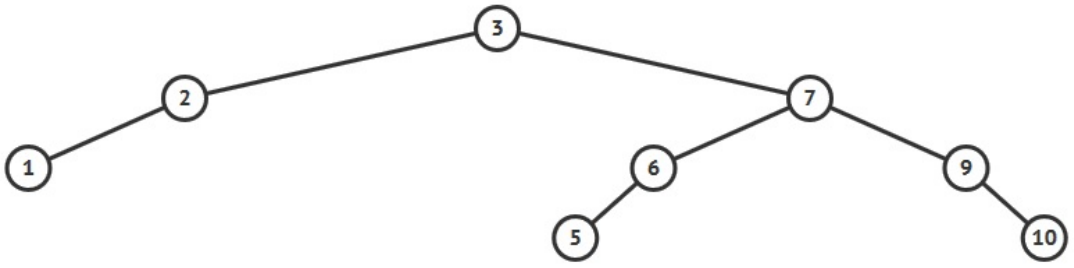
En este caso, se debe aplicar una rotación doble izquierda-derecha porque el nodo desequilibrado es el hijo derecho del hijo izquierdo. Esta doble rotación consistirá en dos rotaciones simples: una primera rotación simple a la izquierda, y a continuación, una segunda rotación simple a la derecha. El resultado de la primera rotación (simple a la izquierda) es el siguiente:



Nótese que el nodo 3 pasa a ser raíz del subárbol pero este movimiento no es suficiente para devolver el equilibrio al árbol completo (dicho desequilibrio estaba en la raíz, 7). Al aplicar la segunda rotación simple a la derecha se reequilibra el árbol completo:

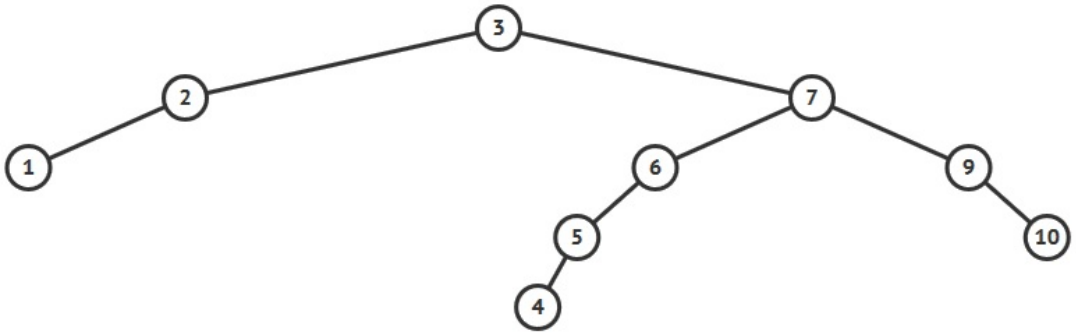


A continuación se insertan los valores 5 y 10, lo cual no supone un desequilibrio del árbol:

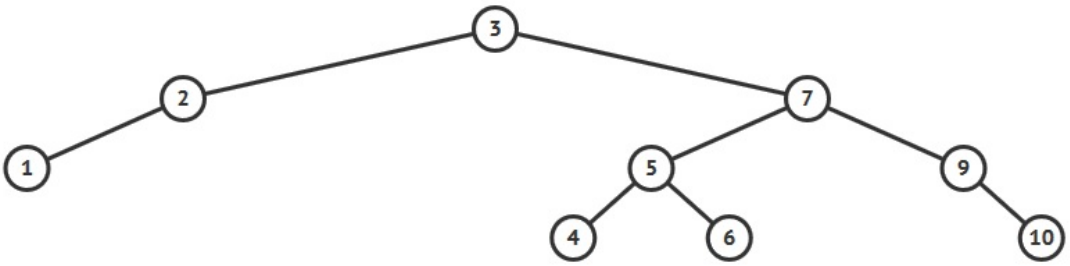


El siguiente valor es 4, el cual provocará un desequilibrio en el árbol que se resolverá mediante una rotación simple (a la derecha, haciendo que 5 ocupe la raíz del subárbol que era hijo

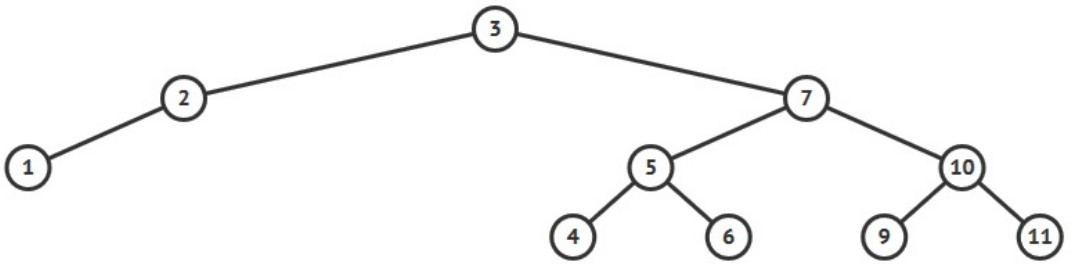
izquierdo de 7 y que 6 pase a ser hijo derecho de 5):



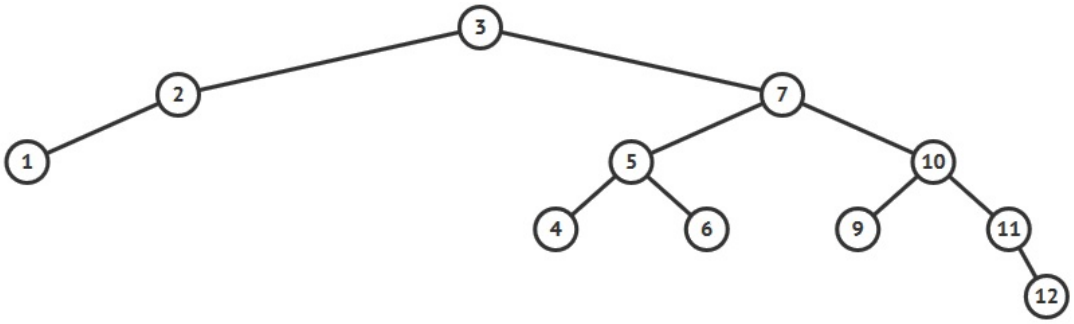
Tras aplicar la rotación se obtiene lo siguiente:



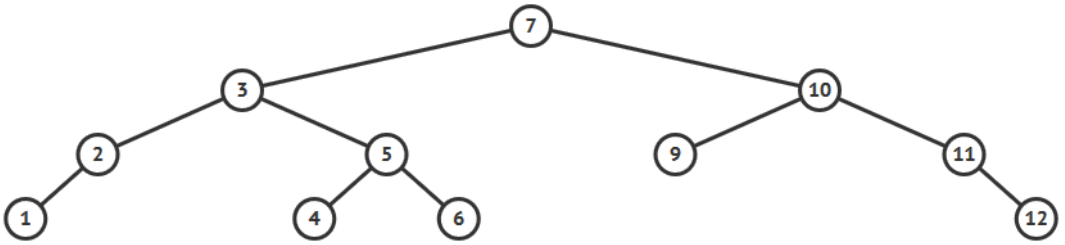
El siguiente valor de la secuencia es 11. El efecto de insertarlo es análogo al del valor anterior, pero afectando al subárbol formado por los nodos 9 y 10. Tras aplicar la rotación correspondiente (simple a la izquierda), se obtiene lo siguiente:



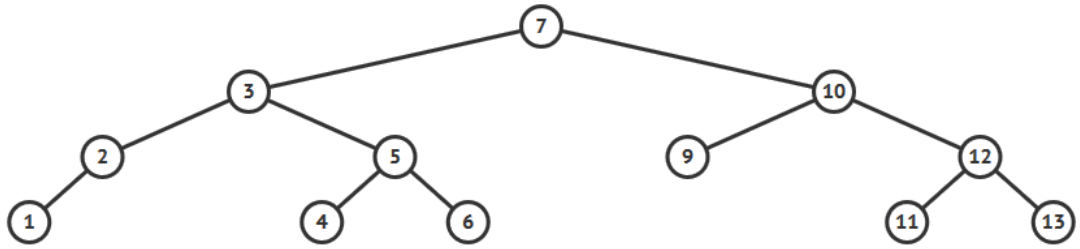
A continuación, se inserta el valor 12:



Este valor provoca un nuevo desequilibrio en el árbol que se resuelve mediante una rotación simple:



Por último, se inserta el elemento 13. Esta inserción provoca un nuevo desequilibrio, que se corregirá mediante una rotación doble: el árbol AVL pedido es el siguiente:



2. Se dice que una partición de un conjunto s es una división de dicho conjunto en grupos (subconjuntos de s), de manera que todos los elementos de s pertenecen a un único grupo. Por ejemplo, tomando $s = \{1, 2, 3, 4, 5\}$, dos particiones de s son $P1 = \{\{1, 2\}, \{3, 5\}, \{4\}\}$ y $P2 = \{\{1, 4, 5\}, \{2, 3\}\}$. La siguiente interfaz, denominada `SetPartitionIF` representa particiones de un conjunto de elementos:

`SetPartitionIF<T>`:

```
public interface SetPartitionIF<T> {
  /* Añade el grupo unitario {e} a la partición                    */
  public void addGroup(T e); // [0'5 puntos]
  /* Elimina de la partición el grupo al que pertenece el elemento e *
  public void removeGroup(T e); // [0'5 puntos]
  /* Devuelve el grupo al que pertenece el elemento e.             */
  public SetIF<T> getGroup(T e); // [0'5 puntos]
  /* Mezcla los grupos a los que pertenecen los elementos e1 y e2  */
  public void merge(T e1, T e2); // [0'5 puntos]
  /* Decide si la partición es válida con respecto al conjunto s    *
   * dado por parámetro. Esto es, comprueba que todos los elementos *
   * de s pertenecen a algún grupo de la partición y que todos los  *
   * grupos de la partición distintos entre sí son disjuntos, i.e., *
   * no comparten ningún elemento entre sí.                         */
  public boolean checkPartition(SetIF<T> s); // [1 punto]
}
```

Se pide:

a) (1 punto) Describa detalladamente cómo realizaría la representación interna de este tipo de datos usando los TAD vistos en la asignatura. Justifique su respuesta. Implemente un constructor de clase que reciba un conjunto de objetos declarado `SetIF<T> s`, y construya la partición unitaria, consistente en crear un grupo por cada elemento del conjunto. Por ejemplo, para el conjunto $s = \{1, 2, 3, 4, 5\}$, dicha partición es $P = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.

Una partición se define como un conjunto de conjuntos (denominados grupos en este ejercicio), de manera que un grupo puede representarse como `SetIF<T>`.

Puesto que la interfaz `SetIF` ofrece operaciones de unión, intersección y diferencia, interpretar de esta manera los grupos nos facilitará la implementación de algunas operaciones del TAD con respecto a usar otras estructuras lineales como las listas. Por ejemplo, para la operación `removeGroup`, es útil contar con la operación de diferencia de conjuntos, para la operación merge es útil contar con la operación de unión de conjuntos y para la operación `checkPartition`, es útil contar con la operación de intersección de conjuntos para comprobar si dos conjuntos son disjuntos. Atendiendo a la definición de partición comentada anteriormente, una posible representación de la misma puede ser `SetIF<SetIF<T>> partition`.

No obstante, otra posible opción es utilizar una estructura indexada como una lista, esto es, `ListIF<SetIF<T>> partition`. El empleo de las otras estructuras de datos vistas en la asig-

natura (pilas, colas y árboles) para representar la partición no ofrece ninguna ventaja adicional. En lo sucesivo tomaremos la presentación de una partición como una lista de conjuntos:

```
public class SetPartition<T>{
  private ListIF<SetIF<T>> partition;
  ...
}
```

El constructor pedido en el enunciado inicializará la lista de grupos y recorrerá los elementos del conjunto dado como parámetro, añadiendo cada uno de los grupos unitarios mediante el método `addGroup(T e)` que implementaremos más adelante:

```
public SetPartition(SetIF<T> s){
  partition = new List<SetIF<T>>();
  IteratorIF<T> iter = s.iterator();
  while(iter.hasNext()){
    T e = iter.getNext();
    addGroup(e);
  }
}
```

b) (3 puntos) Implemente en JAVA todos los métodos de una clase que implemente la interfaz SetPartitionIF. Nótese que en la descripción de la interfaz viene anotada la puntuación de cada uno de ellos.

- Método `addGroup` (0.5 puntos):

```
public void addGroup(T e){
  // Creamos un conjunto unitario con el elemento dado por parámetro
  SetIF<T> s = new Set<T>(e);
  // Se inserta en la lista en una nueva posición
  int size = partition.size();
  partition.insert(s,size+1);
}
```

Nótese que este método, que se utiliza en el constructor de la clase, plantea ciertas dificultades si se declara como público, ya que, su uso sobre particiones ya construidas podría generar ciertas inconsistencias (en términos de particiones) como elementos o grupos repetidos. Para evitarlas, existen dos soluciones rápidas: declararlo privado para que su uso sea exclusivo por parte del constructor (lo que garantizaría la corrección de las particiones *por construcción*) o comprobar en el método que el grupo que se vaya a añadir no forme ya parte de la partición.

- Método `getGroup` (0.5 puntos):

```
public SetIF<T> getGroup(T e){
  IteratorIF<SetIF<T>> iterator = partition.iterator();
  while(iterator.hasNext()){
    SetIF<T> group = iterator.getNext();
    If(group.contains(e)){
      //Encontrado el grupo al que pertenece e
      return group;
    }
  }
  //en caso de no encontrar el grupo, devolvemos el conjunto vacío
  return new Set<T>();
}
```

- Método `removeGroup` (0.5 puntos):

```
public void removeGroup (T e){
 // Buscamos la posición del conjunto de e para eliminarlo de la
    lista.
 int pos = -1;
  int cont = 1; // Mantiene la posición actual en el recorrido
 boolean b = true;
 IteratorIF<T> iterator = s.iterator();
 while(iterator.hasNext() && b){
   SetIF<T> group = iterator.getNext();
   If(group.contains(e)){
     // Hemos encontrado el grupo al que pertenece el elemento e
     pos = cont;
     b = false;
   }
   cont++;
 }
 // Eliminamos el grupo
 If(pos>-1){
   partition.remove(pos);
 }
}
```

- Método merge (0.5 puntos):

```
public void merge(T e1, T e2){
 // Obtenemos los conjuntos de los elementos e1 y e2
 SetIF<T> s1 = getGroup(e1);
 SetIF<T> s2 = getGroup(e2);
 s1 = s1.union(s2);   // Guardamos la unión en el primero de ellos
 // Eliminamos el grupo de e2: evitamos grupos con elementos comunes.
 removeGroup(e2);
}
```

- Método `checkPartition` (1 punto):

```
1  public boolean checkPartition(setIF<T> s){
2  /* [1] todo elemento de s está en algún grupo.
3   * [2] todos los grupos son disjuntos entre sí.
4   * Dos grupos son disjuntos si su intersección es vacía. Como
5   * hay que intersecar cada conjunto con todos los demás,
6   * construiremos un conjunto auxiliar como la unión de todos los
7   * elementos de los conjuntos ya comprobados. La intersección de
8   * cada nuevo conjunto con dicho auxiliar, debe ser vacía por [2]
9   * Por último, la unión de todos los grupos debe coincidir con
10  * el conjunto parámetro para satisfacer [1]                    */
11  SetIF<T> aux = new Set<T>();
12  Iterator<SetIF<T>> it = partition.iterator();
13  while (it.hasNext()){
14    SetIF<T> nGroup = it.getNext();
15    if (!aux.intersection(nGroup).isEmpty()) return false;
16    aux = aux.union(nGroup);
17  }
18  return aux.size() == s.size() && aux.isSubset(s);
19 }
```

c) (1 punto) Para el método `checkPartition(SetIF<T> s)`, calcule el coste asintótico temporal en el caso peor indicando claramente sobre qué valor o valores calcula dicho coste. Justifique adecuadamente su respuesta.

En lo sucesivo, denotaremos por G al número de grupos de la partición, M, al número de elementos diferentes que existen en la partición y E al número de elementos del conjunto s dado por parámetro. El coste lo obtendremos a partir de estos valores, ya que el tamaño será una función de dichos valores[3]:

- El coste será el máximo de las distintas aportaciones, aplicando la regla de la suma. Desde este punto de vista, el coste de la producción del iterador no será superior al de recorrerlo, por lo que la aportación de la línea 12 quedará subsumida en la de las líneas siguientes.
- El bucle que comprende las líneas 13 a 17 se ejecutará, a lo sumo, tantas veces como grupos contenga la partición, es decir, G.
- La línea 14 no depende del tamaño del problema.
- En las líneas 15 y 16, se ejecutan la intersección y la unión conjuntistas con los sucesivos grupos sobre los que se idera. Estas operaciones tendrán como tamaño del problema M, ya que, a lo sumo, operarán sobre todos los elementos del conjunto. No conocemos la implementación de los conjuntos utilizados, por lo que llamaremos $T_{\cap}(n)$ al coste de la intersección y $T_{\cup}(n)$ al de la unión. Podemos suponer, sin pérdida de generalidad, que ambos costes serán similares en magnitud, por lo que podríamos representar ambos por el mayor (o cualquiera) de ellos, que llamaremos $T_{\star}$. Como ambas operaciones se ejecutan en cada vuelta y son las que mayor coste aportan, el del bucle puede calcularse como $T_{while} \in \mathcal{O}(G \times T_{\star}(M))$
- La línea 18 se ejecutará una sola vez y su coste dependerá del número de elementos de cualquiera de los conjuntos (ya que, en este punto, deberían ser iguales) y de la implementación de los conjuntos. Llamaremos a este coste $T_{\subset}(M)$. Llegados a este punto y en el caso peor (una partición donde todos sus grupos fuesen unitarios), el número máximo de grupos también coincidiría con la cardinalidad de cualquiera de estos conjuntos ($M = E = G$).

El coste total se obtiene como $max(T_{while}, T_{\subset})$. Como hemos visto que los tres parámetros aludidos al principio de este apartado, en el peor de los casos, serían iguales, podemos expresar el coste en función de cualquiera de ellos, digamos, E. Si suponemos que la unión y la intersección conjuntistas son cuadráticas ($T_{\star}(E) \in \mathcal{O}(E^2)$) y que la determinación de si un conjunto es subconjunto de otro también puede serlo ($T_{\subset}(E) \in \mathcal{O}(E^2)$). Tendremos, entonces, que $T_{while} \in \mathcal{O}(E \times E^2)$, que es claramente superior a $T_{\subset}$, por lo que el coste total sería $T(E) \in \mathcal{O}(E^3)$.

[3]No es preciso conocer dicha función para determinar el tamaño. Nótese que cualquier función creciente (no necesariamente de forma estricta) en ambos parámetros valdría.