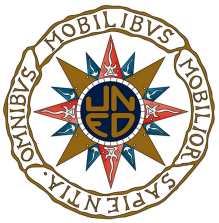

Estrategias de Programación y Estructuras de Datos

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

---

P1 (1'5 puntos) **Práctica**. El criterio que tenemos en la Torre de Control para elegir pista a igualdad de `getDelay`, y que no haya pista vacía, es la primera pista en el índice. En este ejercicio se desea que se elija la pista de menor ocupación (en caso de que el resultado de `getDelay` fuese el mismo), para optimizar así la utilización de las pistas. Describa cómo lo haría justificando su respuesta

Nota: Para el ejercicio de la práctica, damos una indicación y no una solución completa para contemplar la variedad de respuestas posibles dependiendo de la implementación que cada alumno haya realizado.

La solución consistiría en consultar el resultado del método `getDelay` de cada pista y, si el mínimo valor lo tuviese más de una pista, entonces se consultaría la ocupación de las pistas con dicho valor y se elegiría la que tuviese menor ocupación.

---

1. (1 punto) Diseñe una función recursiva que devuelva la concatenación de dos pilas en su orden de acceso, por ejemplo, si leyésemos las pilas desde su cima, la concatenación de [1,2,3,4] con [a,b,c,d,e,f,g] sería [1,2,3,4,a,b,c,d,e,f,g]

```
//concatena la pila con la recibida como parametro
StackIF<T> concat(StackIF<T> stk)
```

Dado que el acceso a la pila invierte su orden de almacenamiento podría parecer que se necesita una pila para volver a tener el original. Sin embargo, al ser el método recursivo, basta con acceder al último elemento sin realizar ninguna inserción. Llegados a ese punto, se pueden insertar elementos desde la base a la cima.

Por otra parte, el enunciado no dice que haya que utilizar las pilas como constantes (es decir, conservarlas), por lo que, la solución más sencilla sería devolver la segunda pila (la que se recibe como parámetro) modificada para contener, en sus posiciones desde la cima y en el orden de almacenamiento original, la primera pila (aquella sobre la que se aplica el método).

Un código para hacerlo podría ser:

```
//concatena la pila con la recibida como parametro
StackIF<T> concat(StackIF<T> stk){
  if(isEmpty()){// caso trivial o base
    return stk;
  }
  else{ // caso no trivial o recursivo
    T cima = getTop();
    pop();
    StackIF<T> stack = concat(stk);
    stack.push(cima);
    return stack;
  }
}
```

2. (1'5 puntos) Diseñe una función que determine si una cola es subcola de otra, recibida como parámetro. Por ejemplo, la cola [1,2,3,4,5] es subcola de [2,3,5,1,2,3,4,5,7,8,9] pero no de [1,2,2,3,4,5,5,6,7]

```
//determina si es subcola de la recibida como parametro
boolean isSubqueue (QueueIF<T> q)
```

Para este problema, consideraremos que la cola sobre la que se realiza la llamada al método es la cola candidata a ser subcola y la que se recibe como parámetro es la candidata a ser la cola contenedora. Por tanto, se utilizan dos iteradores: uno para la candidata a ser subcola y otro para la candidata a ser contenedora.

Debido a que hay que hacer la comprobación sobre la cola contenedora partiendo de cada posición de la misma (mientras no se encuentre una subcola), es necesario llevar un contador que marque en qué posición hay que empezar la próxima comprobación (en caso de ser necesario). Este contador se utiliza para avanzar el iterador de la cola contenedora hasta la posición deseada antes de comenzar la comprobación (Nótese que esto sería distinto en listas. El porqué se deja como ejercicio).

Para cada posición de comienzo de este iterador (el de la cola contenedora), se empieza a recorrer la cola candidata a ser subcola con un iterador desde el principio. Mientras los elementos siguientes que devuelven ambos iteradores sean iguales, se continúa buscando si hay éxito en la búsqueda. En caso de que se termine de recorrer la cola candidata a ser subcola, entonces esta cola es realmente una subcola de la pasada por parámetro.

Un código para hacerlo podría ser:

```
//determina si es subcola de la recibida como parametro
boolean isSubqueue (QueueIF<T> q){
  IteratorIF<T> itContenedora = q.getIterator();
  /** este contador indica empezando desde la cabeza, a partir de qué
     elemento de la cola candidata a ser contenedora se ha empezado la
     última comprobación de si la cola actual es subcola */
  int posicionEnCursoCola = 0;
  boolean esSubCola = false;
  while(!esSubCola && itContenedora.hasNext()){
    /* se reinicia el iterador y se avanza hasta el valor nuevo que se va
       a comprobar */
    itContenedora.reset();
    for (int i = 0; i < posicionEnCursoCola; i++) {
      itContenedora.getNext();
    }
    /* se crea el iterador de la candidata a subcola y se empieza a
       comprobar si la segunda cola está contenida en la primera */
    IteratorIF<T> itSubcola = getIterator();
    /* se guarda un  booleano que indica que los elementos comprobados
       hasta ahora son iguales (y quizás la cola)*/
    boolean comprobados = true;
    while(comprobados &&  itContenedora.hasNext() && itSubcola.hasNext()){
      if(!itContenedora.getNext().equals(itSubcola.getNext())){
        /* si los valores no son iguales, no puede ser una subcola y hay
           que continuar la búsqueda en otro punto */
        comprobados = false;
      }
    }
    /* si todos los elementos comprobados son iguales y se ha recorrido
       toda la cola candidata a subcola, tendremos una subcola. Si sólo
       fuese vacía la cola candidata a contenedora no habría certeza de
       que tuvieramos una subcola */
    if(comprobados && !itSubcola.hasNext()) {
      esSubCola = true;
```

```
        }
        posicionEnCursoCola ++;
      }
      return esSubCola;
    }
```

3. (1'5 puntos) Discuta los costes asintóticos temporales en el caso peor de las siguientes operaciones sobre árboles binarios de búsqueda equilibrados con estrategia AVL. Debe justificar su respuesta.

   En primer lugar y como en todo problema que trata del coste, deberemos establecer cuál es el tamaño de dicho problema. En el caso de los árboles, suele utilizarse el número de nodos del árbol, al que llamaremos n para simplificar la explicación.

   a) Inserción de un elemento

   En los árboles binarios de búsqueda equilibrados mediante la técnica de Adelson-Velskii y Landis (en lo sucesivo, AVL), se garantiza que la altura del árbol (y de cada uno de sus subárboles) es logarítmicamente proporcional al número de nodos de dicho árbol (o subárbol).

   Para insertar un elemento, hay que buscar su ubicación, lo que se realiza en $\mathcal{O}(\log n)$ y, si procede, realizar una o dos rotaciones para restablecer el equilibrio en caso de que éste se hubiese roto, lo que debería ser independiente del tamaño ($\mathcal{O}(1)$). Por lo tanto y aplicando la regla del máximo, el coste debería ser $\mathcal{O}(\log n)$

   b) Búsqueda de un conjunto de n elementos consecutivos

   Para buscar n' (éste "n" no tiene por qué ser el mismo del tamaño, por lo que lo renombramos como n') elementos consecutivos, hay que encontrar el primero ($\mathcal{O}(\log n)$, como hemos visto) y, posteriormente, seguir el recorrido en inorden desde éste para acceder a los restantes $n' - 1$. Dado que n' podría ser comparable con n, en el caso peor, este recorrido implica el completo del árbol, por lo que su coste sería $\mathcal{O}(n)$.

   c) Borrado de un elemento

   El borrado de un elemento requiere, en primer lugar, localizarlo, ya que el elemento que se quiera borrar se identificará por su valor. Esta operación tiene coste $\mathcal{O}(\log n)$. Una vez encontrado el elemento, borrarlo en sí tiene coste constante pero puede producir un desequilibrio, que habrá que remediar y que se puede propagar hasta la raíz del árbol. Como la distancia entre un nodo cualquiera, en particular, el que se iba a borrar y la raíz es logarítmicamente proporcional a la altura, el coste del borrado completo, incluyendo la propagación que se requiera para reequilibrarlo, será $\mathcal{O}(\log n)$

4. Queremos programar la clase de los vectores en $\mathbb{R}^n$. Un vector se representa en función de n componentes reales, que numeraremos de 0 a n-1 y que son sus coordenadas respecto a la base del espacio vectorial. El tipo responde a las siguientes operaciones independientemente de su representación:

VectorIF

```
// Representa un Vector en IR^n
  public interface VectorIF {

    // Devuelve la dimensión del vector (numero de componentes)
    public int getDimension();

    // Obtiene la coordenada index-esima del vector
    public float getCoordinate(int index);

    // reemplaza el valor de la coordenada index-esima por el nuevo
    // que recibe como parametro
```

```
    public void setCoordinate(int index, float newValue);

    // añade una coordenada nueva, que recibe como parámetro,
    //  como última del vector
    public void addCoordinate(float value);

    // calcula el módulo, magnitud o longitud de un vector, que es
    // la raiz cuadrada de la suma de sus componentes elevadas
    // al cuadrado
    public float getModule ();

    // devuelve la suma con el vector parámetro
    public VectorIF sum (VectorIF  vect);

    // devuelve la multiplicación del vector por un escalar, es decir,
    // multiplica cada componente por el parámetro escalar
    public VectorIF scale (float scl);

    // obtiene el producto escalar por el vector parámetro
    public float scalarProduct (VectorIF vect);

}
```

a) (0'5 puntos) Describa detalladamente cómo realizaría la representación interna de este tipo (usando los TAD estudiados en la asignatura). Justifique su elección.

La representación del tipo debe contemplar las operaciones de la interfaz propuesta para soportarlas de la manera más eficiente posible, en especial, aquellas que vayan a suponer el uso más común de entre las disponibles que, por lo general, son las operaciones consultoras.

En este caso, se necesita disponer de las componentes del vector, que serán `n`, si bien ese valor no se detalla ni se declara como constante, esto es, será arbitrario y debería admitirse cualquier tamaño, por lo que elegiremos una representación dinámica, ya que, además, el método `addCoordinate` permite el crecimiento del vector al añadir una nueva coordenada al final.

Dado que existen operaciones en la interfaz para consultar o modificar componentes específicas, debe poder accederse a cualquiera de ellas, por lo que descartaremos pilas y colas, cuyo uso implica (salvo para consulta si se usan iteradores) la consumición de los elementos almacenados en la estructura. Además, habrá que elegir si se utiliza una representación posicional (en la que el `k`-ésimo término de una secuencia contiene la componente `k`-ésima del vector) o bien una representación que asocie cada término a una componente específica de forma explícita. Cada una de las opciones tiene ventajas y desventajas. La representación implícita ocupa menos espacio pero requiere contar el número de términos recorridos hasta llegar a uno dado mientras que la representación explícita identifica cada componente pero a costa de invertir, para cada una de ellas, el espacio redundante de su identificación.

Para esta solución, utilizaremos la representación mediante listas dinámicas con identificación implícita de las componentes o sea, el vector será una secuencia ordenada (lista) de componentes. Dejamos como ejercicio resolver este problema utilizando la otra representación comentada.

La representación del tipo requeriría un atributo privado que contuviese las componentes:

```
/** representación de vectores en IR^n mediante
 * listas dinámicas de componentes    */
public class VectorList implements VectorIF{
// el tipo debe ser un objeto, por tanto, Float (no float)
 private ListIF<Float> components;
}
```

b) (0'5 puntos) Detalle el constructor de una clase que implemente esta interfaz.
Por motivos didácticos, ofreceremos dos constructores, uno de los cuales recibe una lista de reales que serían las componentes del vector.

```
//constructor por defecto
  public VectorList (){
   components = new ListDynamic<Float>();
 }
// constructor por copia; recibe una lista de componentes
   public VectorList (ListIF<Float> vlist){
   components = new ListDynamic<Float>(vlist);
  }
```

c) (2'5 puntos) Basándose en las respuestas anteriores, implemente todos los métodos de la interfaz `VectorIF`. Se valorará que detalle los contratos de las operaciones (pre y postcondiciones) o que comente, al menos, las restricciones que deben aplicarse a los parámetros de entrada.
Para la solución de este ejercicio, programaremos *por contrato*, es decir, incluiremos pre y postcondiciones[1], entendiendo que la precondición debe cumplirse antes de invocar el método en cuyo caso, debe satisfacerse la postcondición.
Para la solución de este examen modelo, incluímos estos comentarios que, si bien no son obligatorios en un examen real, serán valorados positivamente si aparecen y son correctos. En el código, se incluyen otros comentarios para hacer más fácil su comprensión. Una vez más, esto no es algo que se exija a los alumnos, aunque, por supuesto, los comentarios se valorarán si aparecen y son correctos.
Por motivos de maquetación, cambiaremos el orden de resolución en lugar de seguir el orden en el que aparecen los métodos en la interfaz)

```
// Representa un Vector en IR^n
   public interface VectorIF {

    /** Devuelve el la dimensión del vector (numero de
     *  componentes)                                    */
    public int getDimension(){
     return components.getLength();
    }

    /** Añade una coordenada nueva, que recibe como parámetro,
     *  como última del vector
     * @pre: true
     * @post: getCoordinate(n+1) == newValue; la lista de coordenadas
        crece una unidad en tamaño.
     * Es necesario insertar el nuevo elemento al final de la
     * lista de componentes. Para ello hay que llegar al final
     * de la lista. Cuando se llega al final, se inserta en
     * la lista la nueva coordenada.                      */

     public void addCoordinate(float value) {
       /** la nueva coordenada se tiene que insertar al final
        *  de la lista de coordenadas                      */
        ListIF<Float> lista = components;

       /** se va recorriendo la lista hasta llegar a la lista
        * vacía.                                           */
        while(!lista.isEmpty()){
```

[1]Mediante las anotataciones @pre y @post, en las que el resultado, si fuese preciso referirse a él separadamente, se referiría como @return

```
      lista = lista.getTail();
     }
     lista.insert(value);
   }

 /** reemplaza el valor de la coordenada index-esima por
  *  el nuevo que recibe como parametro
  * @pre: 0 <= index <= n-1
  * @post: getCoordinate(index) == newValue
  * Teniendo en cuenta el tad seleccionado (una lista) y que
  * en el mismo no se puede modificar un elemento sin más, hay
  * que realizar una serie de operaciones para llevarlo a cabo
  * concretamente:
  * 1: hay que llegar al punto de la lista donde hay que
  *    modificar el valor
  * 2: crear una lista con el resto de la lista donde se está
  *    (sin incluir el elemento en la posición a modificar)
  * 3: insertar el nuevo elemento
  * 4: insertar los elementos que estaban por delante. Como el
  *    acceso a los mismos no es en el orden en el que se
  *    tienen que insertar, se utiliza una pila para
  *    almacenarlos                                          */

  public void setCoordinate(int index, float newValue) {
    /** 1: hay que llegar al punto de la lista donde hay que
     *     modificar el valor */
    StackIF<Float> pilaGuarda = new StackDynamic<Float>();
    ListIF<Float> listaActual = components;
    for (int i = 0; i < index; i++) {
      pilaGuarda.push(listaActual.getFirst());
      listaActual = listaActual.getTail();
    }
    /** 2: crear una lista con el resto de la lista donde se
     *     está (sin incluir el elemento en la posición que hay
     *     que  modificar) */
    ListIF<Float> listaNueva = new ListDynamic<Float>(listaActual.
       getTail());
    /** 3: insertar el nuevo elemento */
    listaNueva.insert(newValue);
    /** 4: insertar los elementos que estaban por delante. Como
      *    el acceso a los mismos no es en el orden en el que
      *    se tienen que insertar, se utiliza una pila para
      *    almacenarlos   */
    while(!pilaGuarda.isEmpty()){
      listaNueva.insert(pilaGuarda.getTop());
      pilaGuarda.pop();
    }
    components = listaNueva;
  }
```

```
/** Obtiene la coordenada index-esima del vector
 * @pre: 0 <= index <= n-1
 * @post: @return el valor de la componente index-esima
 *        del vector
 * Hay que realizar un acceso mediante un iterador a la
 * componente que se desea                                */

 public float getCoordinate(int index) {
   IteratorIF<Float> it = components.getIterator();
   for (int i = 0; i < index; i++) {
     it.getNext();
   }
   return it.getNext();
 }
/** devuelve la suma con el vector parámetro
 * asumimos que ambos vectores son de igual dimension
 * @pre: this.getDimension()== vect.getDimension()
 * @post: @return vectSum donde
 * vectSum.getCoordinate(i)==
 *         this.getCoordinate(i)+vect.getCoordinate(i)
 * Se crea un nuevo vector similar al actual. A continuación
 * se obtiene el número de componentes del vector, para
 * utilizar un bucle para sustituir cada componente por el
 * valor de sumar la componente antigua a la componente
 * correspondiente (de la misma posición) en el vector pasado
 * por parámetro.                                         */

 public VectorIF sum(VectorIF vect) {
   VectorIF salida = new VectorList(components);
   int numComponents = components.getLength();
   for (int i = 0; i < numComponents; i++) {
     salida.setCoordinate(i, salida.getCoordinate(i) + vect.
        getCoordinate(i));
   }
   return salida;
 }

/** Calcula el módulo, magnitud o longitud de un vector,
 *  que es la raiz cuadrada de la suma de sus componentes
 *  elevadas al cuadrado
 * @pre: true
 * @post: \sqrt{\sum_{i=0}^{n-1} getCoordinate(i)^2}
 * Se hace un recorrido por cada componente de la lista de
 * componentes, acumulando la suma de los cuadrados. Al final,
 * se calcula la raiz cuadrada                            */

 public float getModule() {
   float resultado = 0;
   IteratorIF<Float> it = components.getIterator();
   while(it.hasNext()){
     resultado = resultado +
                 (float)Math.pow(it.getNext() , 2);
   }
   resultado = (float)Math.sqrt(resultado);
   return resultado;
 }
```

```
/** Devuelve la multiplicación del vector por un escalar,
 *  es decir, multiplica cada componente por el parámetro
 *  escalar
 * @pre: true
 * @post: @return vectEsc donde
 * vectEsc.getCoordinate(i)== scl
 *                    * this.getCoordinate(i)
 * Se crea un nuevo vector a partir del actual, y a
 * continuación se recorre el nuevo vector sustituyendo cada
 * componente por el resultado de multiplicar la componente
 * antigua por el valor escalar pasado por parámetro.        */

 public VectorIF scale(float scl) {
   VectorIF salida = new VectorList(components);
   int numComponents = components.getLength();
   for (int i = 0; i < numComponents; i++) {
     salida.setCoordinate(i, scl * salida.getCoordinate(i));
   }
   return salida;
 }

/** obtiene el producto escalar por el vector parámetro
 * @pre: this.getDimension()== vect.getDimension()
 * @post: @return \sum_{i=0}^{n-1} this.getCoordinate(i) * vect.getCoordinate(i)
 * Se utiliza un bucle para ir recorriendo ambos vectores
 * componente a componente. En cada paso del recorrido se
 * acumula el valor de multiplicar la componente actual del
 * vector con la componente de la misma posición del vector
 * parámetro.                                                */

 public float scalarProduct(VectorIF vect) {
   float salida = 0;
   int numComponents = components.getLength();
   for (int i = 0; i < numComponents; i++) {
     salida = salida +
              getCoordinate(i) * vect.getCoordinate(i);
   }
   return salida;
 }
}
```

d) (1 punto) Calcule el coste asintótico temporal en el caso peor para el método `scalarProduct`. Justifique su respuesta y discuta si se necesita un gasto adicional de espacio para implementar este método.

En primer lugar, seleccionaremos como tamaño del problema el número de componentes del vector (que el enunciado llamaba `n`). Como se puede observar, `scalarProduct` es un código iterativo que realiza un número de vueltas proporcional al tamaño definido (`n`, el número de componentes del vector).

Dentro del cuerpo del bucle, se llama (dos veces) a `getCoordinate` (cuyo código puede verse en la página 7), por lo que el coste de la iteración dependerá del de dicho método.

Coste de `getCoordinate`. El tamaño será igualmente el número de coordenadas, n. Se trata de un bucle que, en el caso peor, realiza tantas iteraciones como componentes tenga el vector. En cada una de ellas, las operaciones que se llevan a cabo son independientes del tamaño (manipulación del

iterador, cuya construcción se realiza fuera del bucle), por lo que el coste de este método será lineal respecto a n ($T_{\texttt{getCoordinate}}(n) \in \Theta(n)$)

Por lo tanto, el coste del cuerpo del bucle será lineal respecto a `n` (la suma, la asignación y el producto no dependen del tamaño y, como las medidas asintóticas son independientes de constantes, da igual tener dos llamadas a un método lineal que una en el cuerpo). Como se realiza un número de iteraciones también linealmente proporcional respecto a `n`, el coste del bucle completo será ($T_{\texttt{bucle}}(n) \in \Theta(n^2)$)

Sólo queda ya evaluar cuál es el coste de las operaciones que se realizan fuera del bucle, es decir, la inicialización de la variable de salida, que no depende del tamaño y la obtención de la longitud (número de componentes del vector) mediante la llamada a `getLength`. Este método *podría* depender del número de componentes, si bien lo óptimo es que dicha operación se implemente mediante la lectura de un atributo privado que se modifique adecuadamente en cada operación que proceda[2] de manera que el coste de hallar la longitud sea constante. Aún en el caso de ser lineal (dependiente del tamaño en forma lineal), aplicando la regla del máximo, el coste del método sería el del bucle, ya que es de un orden de magnitud mayor que el de la inicialización.

En resumen, el coste completo es cuadrático respecto al número de componentes del vector, o, lo que es lo mismo $T_{\texttt{scalarProduct}}(n) \in \Theta(n^2)$

Por último, en cuanto al espacio, como los datos que se utilizan son los parámetros y los que ya se tenían en el vector original y no hace falta usar un iterador, concluímos que no será necesaria ninguna cantidad adicional de espacio.

[2]Se deja como ejercicio averiguar cuáles son esas operaciones y añadir el código adecuado para el cálculo de la longitud en tiempo constante

ListIF (Lista)

```
/* Representa una lista de
   elementos */
public interface ListIF<T>{
    /* Devuelve la cabeza de una
       lista*/
     *
     public T getFirst ();
     /* Devuelve: la lista
        excluyendo la cabeza. No
        modifica la estructura */
     public ListIF<T> getTail ();
    /* Inserta una elemento
       (modifica la estructura)
     * Devuelve: la lista modificada
     * @param elem El elemento que
        hay que añadir*/
     public ListIF<T> insert (T
        elem);
     /* Devuelve: cierto si la
        lista esta vacia */
     public boolean isEmpty ();
     /* Devuelve: cierto si la
        lista esta llena*/
     public boolean isFull();
     /* Devuelve: el numero de
        elementos de la lista*/
     public int getLength ();
     /* Devuelve: cierto si la
        lista contiene el elemento.
      * @param elem El elemento
         buscado */
     public boolean contains (T
        elem);
     /* Ordena la lista (modifica
        la lista)
      * @Devuelve: la lista ordenada
      * @param comparator El
         comparador de elementos*/
     public ListIF<T> sort
        (ComparatorIF<T>
        comparator);
     /*Devuelve: un iterador para
        la lista*/
     public IteratorIF<T>
        getIterator ();
}
```

StackIF (Pila)

```
/* Representa una pila de
   elementos */
public interface StackIF <T>{
    /* Devuelve:  la cima de la
       pila */
     public T getTop ();
    /* Incluye un elemento en la
       cima de la pila (modifica
       la estructura)
     * Devuelve: la pila
        incluyendo el elemento
     * @param elem Elemento que se
        quiere añadir */
     public StackIF<T> push (T
        elem);
    /* Elimina la cima de la pila
       (modifica la estructura)
     * Devuelve: la pila
        excluyendo la cabeza */
     public StackIF<T> pop ();
    /* Devuelve: cierto si la pila
       esta vacia */
     public boolean isEmpty ();
    /* Devuelve: cierto si la pila
       esta llena */
     public boolean isFull();
    /* Devuelve: el numero de
       elementos de la pila */
     public int getLength ();
    /* Devuelve: cierto si la pila
       contiene el elemento
     * @param elem Elemento
        buscado */
     public boolean contains (T
        elem);
    /*Devuelve: un iterador para
       la pila*/
     public IteratorIF<T>
        getIterator ();
}
```

QueueIF (Cola)

```
/* Representa una cola de
   elementos */
public interface QueueIF <T>{
    /* Devuelve: la cabeza de la
       cola */
     public T getFirst ();
    /* Incluye un elemento al
       final de la cola (modifica
       la estructura)
     * Devuelve: la cola
        incluyendo el elemento
     * @param elem Elemento que se
```

```
        quiere añadir */
      public QueueIF<T> add (T
         elem);
     /* Elimina el principio de la
        cola (modifica la
        estructura)
      * Devuelve: la cola
         excluyendo la cabeza    */
      public QueueIF<T> remove ();
     /* Devuelve: cierto si la cola
        esta vacia */
      public boolean isEmpty ();
     /* Devuelve: cierto si la cola
        esta llena */
      public boolean isFull();
     /* Devuelve: el numero de
        elementos de la cola */
      public int getLength ();
     /* Devuelve: cierto si la cola
        contiene el elemento
      * @param elem elemento
         buscado */
      public boolean contains (T
         elem);
     /*Devuelve: un iterador para
        la cola*/
      public IteratorIF<T>
         getIterator ();
}
```

TreeIF (Árbol general)

```
/* Representa un arbol general de
   elementos */
public interface TreeIF <T>{
    public int PREORDER  = 0;
    public int INORDER   = 1;
    public int POSTORDER = 2;
    public int BREADTH   = 3;
    /* Devuelve:  elemento raiz
       del arbol */
     public T getRoot ();
    /* Devuelve: lista de  hijos
       de un arbol.*/
     public ListIF <TreeIF <T>>
        getChildren ();
    /* Establece el elemento raiz.
     * @param elem Elemento que se
        quiere poner como raiz*/
     public void setRoot (T
        element);
    /* Inserta un subarbol como
       ultimo hijo
     * @param child el hijo a
        insertar*/
     public void addChild
        (TreeIF<T> child);
    /* Elimina el subarbol hijo en
       la posicion index-esima
     * @param index indice del
        subarbol comenzando en 0*/
     public void removeChild (int
        index);
    /* Devuelve: cierto si el
       arbol es un nodo hoja*/
     public boolean isLeaf ();
    /* Devuelve: cierto si el
       arbol es vacio*/
     public boolean isEmpty ();
    /* Devuelve: cierto si la
       lista contiene el elemento
     * @param elem Elemento
        buscado*/
     public boolean contains (T
        element);
    /* Devuelve: un iterador para
       la lista
     * @param traversalType el
        tipo de recorrido, que
     * sera PREORDER, POSTORDER o
        BREADTH  */
     public IteratorIF<T>
        getIterator (int
        traversalType);
}
```

BTreeIF (Árbol Binario)

```
/* Representa un arbol binario de
   elementos */
public interface BTreeIF <T>{
  public int PREORDER  = 0;
  public int INORDER   = 1;
  public int POSTORDER = 2;
  public int LRBREADTH = 3;
  public int RLBREADTH = 4;
 /* Devuelve: el elemento raiz del
    arbol */
  public T getRoot ();
 /* Devuelve: el subarbol
    izquierdo o null si no existe
    */
  public BTreeIF <T> getLeftChild
     ();
```

```
  /* Devuelve: el subarbol derecho
     o null si no existe */
   public BTreeIF <T> getRightChild
      ();
  /* Establece el elemento raiz
   * @param elem Elemento para
      poner en la raiz */
   public void setRoot (T elem);
  /* Establece el subarbol izquierdo
   * @param tree el arbol para
      poner como hijo izquierdo */
   public void setLeftChild
      (BTreeIF <T> tree);
  /* Establece el subarbol derecho
   * @param tree el arbol para
      poner como hijo derecho */
   public void setRightChild
      (BTreeIF <T> tree);
  /* Borra el subarbol izquierdo */
   public void removeLeftChild ();
  /* Borra el subarbol derecho */
   public void removeRightChild ();
  /* Devuelve: cierto si el arbol
     es un nodo hoja*/
   public boolean isLeaf ();
  /* Devuelve: cierto si el arbol
     es vacio */
   public boolean isEmpty ();
  /* Devuelve: cierto si el arbol
     contiene el elemento
   * @param elem Elemento buscado */
   public boolean contains (T elem);
  /* Devuelve un iterador para la
     lista.
   * @param traversalType el tipo
      de recorrido que sera
     PREORDER, POSTORDER, INORDER,
        LRBREADTH o RLBREADTH */
 public IteratorIF<T> getIterator
     (int traversalType);
}
```

ComparatorIF

```
/* Representa un comparador entre
   elementos */
public interface ComparatorIF<T>{
  public static int LESS  = -1;
  public static int EQUAL = 0;
  public static int GREATER = 1;
  /* Devuelve: el orden de los
     elementos
   * Compara dos elementos para
      indicar si el primero es
   * menor, igual o mayor que el
      segundo elemento
   * @param e1 el primer elemento
   * @param e2 el segundo elemento
       */
   public int compare (T e1, T e2);
  /* Devuelve: cierto si un
     elemento es menor que otro
   * @param e1 el primer  elemento
   * @param e2 el segundo elemento
      */
   public boolean isLess (T e1, T
      e2);
  /* Devuelve: cierto si un
     elemento es igual que otro
   * @param e1 el primer  elemento
   * @param e2 el segundo elemento
      */
   public boolean isEqual (T e1, T
      e2);
  /* Devuelve: cierto si un
     elemento es mayor que otro
   * @param e1 el primer  elemento
   * @param e2 el segundo elemento*/
   public boolean isGreater (T e1,
      T e2);
}
```

IteratorIF

```
/* Representa un iterador sobre
   una abstraccion de datos */
public interface IteratorIF<T>{
    /* Devuelve: el siguiente
       elemento de la iteracion */
     public T getNext ();
    /* Devuelve: cierto si existen
       mas elementos en el
       iterador */
     public boolean hasNext ();
    /* Restablece el iterador para
       volver a recorrer la
       estructura */
     public void reset ();
}
```