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**. Explíquense y justifíquense cuáles serían las diferencias entre utilizar como representación interna del tipo una lista o un árbol binario para la torre de control.

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.

Diferencias entre utilizar una lista o un árbol:

- El tad que se utilice para la torre de control debe de permitir un acceso indexado a las pistas. Es decir, dado un valor entero que represente un índice, permitir acceder a la pista representada por dicho índice
- La utilización de una lista facilita este acceso debido a que se supone el orden mismo de la lista para tener los valores del índice. Es decir, la pista que se encuentra en cabeza de la lista es la pista con índice 1. Sin embargo, al ser secuencial el acceso a las pistas, este acceso tiene coste lineal. Por otro lado, para poder acceder a un determinado índice haciendo uso de un árbol binario (asumiendo que es de búsqueda), el contenido de cada nodo debe de ser un valor entero que contenga el valor del índice, además de la pista. Teniendo en cuenta este detalle de implementación, el uso de este tad permite el acceso a cada pista con coste $\mathcal{O}(\log n)$

---

1. (3 puntos) Implemente un algoritmo que reciba una lista de enteros, y haciendo uso de solamente un árbol binario de búsqueda con equilibrado AVL, devuelva una lista con el mismo contenido que la primera pero con los elementos en orden creciente. El algoritmo debe tener un coste $\mathcal{O}(n \log n)$ y no debe implementar ni la ordenación por inserción, ni por mezcla. La cabecera en Java del algoritmo que se pide es:

```
ListIF<int> ordenar (ListIF<int> datos)
```

Aunque en el enunciado ponga `int` por simplicidad debe ser `Integer` (debe ser una referencia para que compile).

```
ListIF<Integer> ordenar (ListIF<Integer> lista_entrada){
  BTreeIF<Integer> arbol = new BTreeIFAVL<Integer>();
  IteratorIF it = lista_entrada.getIterator();
  while(it.hasNext()){
    insert(arbol, it.getNext());
  }
  ListIF<Integer> listaSalida = new ListDynamic<Integer>();
  while(!arbol.isEmpty()){
    Integer max = getMax(arbol);
    listaSalida.insert(max);
    remove(arbol, max);
  }
  return listaSalida;
}
```

Nota: se deja como ejercicio implementar `insert`, `getMax` y `remove`. Nótese que estas operaciones se están invocando como procedimientos que reciben un árbol como parámetro en lugar de hacerlo desde la propia interfaz de la clase, ya que esto requeriría cambiar la interfaz prescrita por la asignatura, que perdería su carácter recursivo. Por este motivo, se ha optado por esta solución para los métodos privados antedichos.

2. Un conjunto es una colección de datos de capacidad ilimitada cuyos elementos no pueden repetirse y donde no existe un orden establecido. Las operaciones que pueden aplicarse sobre los conjuntos aparecen definidas en la siguiente interfaz:

SetIF

```
// Representa un conjunto de elementos
public interface SetIF<T>{

 // añade un elemento al conjunto
 public void add (T e);

 // elimina un elemento del conjunto
 public void remove (T e);

 // devuelve cierto si el elemento pertenece al conjunto
 public boolean contains (T e);

 // devuelve la unión con el conjunto set (@param)
 public SetIF<T> union (SetIF<T> set);

 // devuelve la intersección con el conjunto set (@param)
 public SetIF<T> intersection (SetIF<T> set);

 // devuelve la diferencia simétrica con el conjunto set (@param)
 public SetIF<T> difference (SetIF<T> set)
}
```

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

Si bien la representación e implementaciones más eficientes para este ejercicio se deberían basar en árboles AVL, en consideración al gran porcentaje de alumnos que han utilizado listas, lo haremos de esta última forma. Dejamos como ejercicio realizar este problema mediante árboles AVL.

- Representación interna: Uso de una lista para así poder tener acceso a todos los elementos almacenados sin perder los intermedios, como ocurriría en pilas y colas. Por otra parte, los árboles binarios facilitarían la búsqueda de elementos, si bien, para ello sería precisa la utilización de comparadores (interfaz `ComparatorIF`), que estableciesen un orden entre los elementos (ésta no es una característica base del TAD Conjunto, sino que la añadimos a la implementación concreta por razones de eficiencia). Para conseguir mayor eficiencia (coste proporcional a la altura del árbol, siendo ésta logarítmica respecto al número de nodos del mismo), habría que utilizar una de las técnicas de equilibrado que se han estudiado y que se pueden consultar en el texto base.
- Implementación:

```
public class SetUNED<T> implements SetIF<T>{

  private ListIF<T> contenido;

  public SetUNED() {
```

```
            contenido = new ListDynamic<T>();
          }
        }
```

b) (4'5 puntos) Basándose en la respuesta anterior, implemente todos los métodos de la interfaz **SetIF<T>**

```
public interface SetIF<T>{

 // añade un elemento al conjunto
 public void add(T e) {
     if(!contenido.contains(e)){
         contenido.insert(e);
     }
 }

 // elimina un elemento del conjunto
 public void remove(T e) {
     //es un simple remove de lista, se pueden hacer varias
     //versiones
     IteratorIF<T> it = contenido.getIterator();
     ListIF<T> contenidoAux = new ListDynamic<T>();
     while(it.hasNext()){
         T element = it.getNext();
         if(!element.equals(e)){
             contenidoAux.insert(element);
         }
     }
     contenido = contenidoAux;
 }

 // devuelve cierto si el elemento pertenece al conjunto
 public boolean contains(T e) {
     return contenido.contains(e);
 }

 // devuelve la unión con el conjunto set (@param)
 public SetIF<T> union(SetIF<T> set) {
   IteratorIF<T> it = contenido.getIterator();
   while(it.hasNext()){
     set.add(it.getNext());
   }
   return set;
 }

 // devuelve la intersección con el conjunto set (@param)
 public SetIF<T> intersection(SetIF<T> set) {
   SetIF<T> salida = new SetUNED<T>();
   IteratorIF<T> it = contenido.getIterator();
   while(it.hasNext()){
     T elem = it.getNext();
     if(set.contains(elem)){
       salida.add(elem);
     }
   }
   return salida;
```

```
}

  /** devuelve la diferencia simétrica con el conjunto set (@param)
   *  hemos aceptado la diferencia (no simétrica), cuyo código
   *  damos aquí. La que se solicitaba era
   *  A∆B = (A ∪ B) \ (A ∩ B) y su código requiere un
   *  bucle más, que añade a salida los elementos que están en set
   *  pero no en el conjunto cliente
   */
  public SetIF<T> difference(SetIF<T> set) {
    SetIF<T> salida = new SetUNED<T>();
    IteratorIF<T> it = contenido.getIterator();
    while(it.hasNext()){
      T elem = it.getNext();
      if(!set.contains(elem)){
        salida.add(elem);
      }
    }
    return salida;
  }
}
```

c) (0'5) ¿Qué coste asintótico temporal en el caso peor tiene el método de adición (`add`) en su implementación?

En primer lugar y como en todo ejercicio de cálculo del coste, hay que determinar el tamaño del problema, que, en este caso, será la cardinalidad del conjunto (el número de elementos que contiene), al que llamaremos `card`. Cuando se trata de una representación con una lista, `card=set.getLength()`

El método `add` requiere comprobar que el elemento que se va a añadir no esté ya en el conjunto (llamada a `contains`), por lo que el coste depende del de este último método.

`contains` supone una llamada al método homónimo de la lista que representa el conjunto, lo que implica el recorrido de la lista completa (en el peor caso), cuyo tamaño es `card`, es decir, se realizará un bucle de `card` vueltas, cada una de las cuales se compara el elemento parámetro que se quiere añadir al conjunto con el elemento en curso del recorrido de la lista que representa a dicho conjunto. Esta comparación tiene un coste independiente del tamaño del conjunto, por lo que lo consideraremos constante.

Por último, la inserción de un elemento en la lista que representa el conjunto tiene coste constante, ya que no implica recorrido.

En definitiva $T_{add}(card) \in \mathcal{O}(card)$

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 ();
}
```