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**. Suponga que se permite que una de las pistas (y como máximo una) de nuestro aeropuerto pueda estar inoperativa. Para ello, se incorpora a la torre de control el método `public void closeRunway (int index)`, que marca a la pista con índice `index` (los índices comienzan en 1) como cerrada, de modo que ya no se puede realizar ninguna inserción en ella. Además, este método reubica a todos los vuelos de la pista que se ha marcado como cerrada en las demás pistas, cumpliéndose los requisitos que se tenían antes para la asignación de pista cuando se pedía pista para una operación. Implemente el método `public void closeRunway (int index)` e indique si hay que realizar modificaciones adicionales en 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.

a) Modificaciones a la Torre de Control.
Se añade un atributo de tipo entero que recoje información sobre si hay alguna pista inoperativa y cuál es. La implementación podría ser del estilo de la siguiente:

```
/**     -1: no hay pistas inoperativas
 * n != -1: indice (desde 1) de la pista inoperativa
 */
int  inoperativa =   -1;
```

b) En el método `dispatch` de la torre de control, se debe controlar que no se inserten vuelos en la pista inoperativa. Para ello, no debe consultarse el estado de dicha pista para que así no se compruebe que está vacía y sea candidata a contener nuevos vuelos. El código sería algo así:

```
if(inoperativa != pistaComprobadaActual){
 /** TODO: obtener el valor de getDelay de la pista
   *  pistaComprobadaActual y ver si es menor al acumulado
   */
}
```

c) Implementación del método que se pide:

```
public void closeRunway(int index){
/** se va vaciando la pista y se reubican los vuelos
  *  utilizando el método dispatch
  */
  while(!getRunway(index).isEmpty()){
    OperationIF op = getRunway(index).releaseOperation();
       dispatch(op);
  }
}
```

1. (1 punto) Diseñe un método `ListIF<int> digits (int input)` que, dado un número, obtenga la lista de dígitos que lo representa

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

```
ListIF<Integer> digits(int input){
  ListIF<Integer> salida = new ListDynamic<Integer>();
  while (input != 0){
    int digit = input % 10;
    input = input / 10;
    salida.insert(digit);
  }
  return salida;
}
```

2. (1'5 puntos) Diseñe un método `ListIF<int> repeated(ListIF<int> input)` que genere una lista (de salida) con el número de veces que aparece cada elemento en la lista de entrada, donde el elemento i-ésimo de la lista de salida se corresponde con el número de veces que aparece el elemento i-ésimo (distinto) en la de entrada (por ejemplo, para la entrada $[0, 3, 0, 2, 8, 8, 4, 5]$ el resultado sería $[2, 1, 1, 2, 1, 1]$ (0 aparece 2 veces, 3 aparece 1, 2 está una vez, hay 2 ochos, un cuatro y un cinco). Se valorará la eficiencia.

   Los problemas con los que nos enfrentamos son los siguientes:

   - la inserción en la lista se realiza por delante y si vamos mirando la lista parámetro desde el principio, tendríamos en última posición el número de elementos que hay del primer elemento
   - hay que vigilar que, según se avance al recorrer la lista parámetro, no se vuelvan a contar elementos ya contados

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

```
ListIF<Integer> repeated(ListIF<Integer> input){
  IteratorIF<Integer> itControlador = input.getIterator();
  //lista donde se guardan los elementos ya contados
  ListIF<Integer> listaControl = new ListDynamic<Integer>();
  int numRecorridos = 0;
  // los contadores de los elementos se guardan en una pila
  // luego se pasarán  a una lista en el orden deseado
  StackIF<Integer> salidaAux = new StackDynamic<Integer>();
  while (itControlador.hasNext()){
    Integer actual = itControlador.getNext();
    //solo se cuentan los elementos no contados ya
    if(!listaControl.contains(actual)){
      IteratorIF<Integer> itContador = input.getIterator();
      //avanzar hasta la posición del elemento que se va a contar
      for (int i = 0; i < numRecorridos; i++) {
        itContador.getNext();
      }
      int contador = 0;
      while(itContador.hasNext()){
        if(itContador.getNext().equals(actual)){
          contador ++;
        }
      }
      listaControl.insert(actual);
```

```
        salidaAux.push(contador);
      }
      numRecorridos ++;
    }
    //se crea una lista de salida con el orden deseado
    ListIF<Integer> salida = new ListDynamic<Integer>();
    while(!salidaAux.isEmpty()){
      salida.insert(salidaAux.getTop());
      salidaAux.pop();
    }
    return salida;
  }
```

3. Se pretende implementar, a partir de los TAD estudiados en la asignatura, un nuevo TAD Dictionary, cuyo propósito consiste en albergar una colección de palabras con sus respectivas acepciones. Las palabras se encuentran almacenadas en el diccionario en orden lexicográfico (el de un diccionario al uso). Una palabra puede tener asociada una colección de posibles acepciones.

   Un ejemplo de entrada de diccionario sería la siguiente:

   trueque

   a) Acción y efecto de trocar o trocarse.

   b) Intercambio directo de bienes y servicios, sin mediar la intervención de dinero.

   La definición del interfaz del tipo es la siguiente:

   DictionaryIF

```
/** Representa un diccionario de terminos (palabras)
* cada una de las cuales incluye una colección de
* acepciones  */
public interface DictionaryIF{

 // Añade una definición a una palabra del diccionario
 public void addDefinition (String term, String definition);

 // Devuelve la primera definición del término
 public String getDefinition (String term);

 // Devuelve la index-ésima definición del término
 public String getDefinition (String term, int index);

 // Devuelve la lista completa de definiciones del término
 public ListIF<String> getDefinitions (String term);
}
```

   Nota: se permite utilizar los métodos de la clase `String`, en particular, se recomiendan **`int`** `length()`, que devuelve la longitud de una cadena (String) y **`char`** `charAt(`**`int`** `index)`, que, dada una posición de la cadena indexada desde 0, devuelve el carácter que ocupa dicha posición.

   a) (0'5 puntos) Defina la estructura o representación interna del tipo (usando los TAD estudiados en la asignatura). Justifique su elección

   Lo esencial en este ejercicio es conseguir que las búsquedas de elementos sean lo más eficientes posible, cualquiera que sea su propósito.

Un diccionario es un conjunto de palabras, que en este ejercicio se llaman *términos*, cada una de las cuales tiene un conjunto de *definiciones*. Tanto los términos como las definiciones se componen de cadenas de caracteres (`String`) según la morfología del idioma representado en el diccionario.

Nótese que un término tiene como prefijos todas las cadenas de caracteres ordenadas de la misma forma que dicho término pero con menor longitud que éste. Por ejemplo, son prefijos de `abadía` los siguientes `prefijos={a,ab,aba,abad,abadí}`, algunos de los cuales pueden ser también términos del diccionario. Además, dos términos podrían tener prefijos comunes. Por ejemplo, el más largo prefijo común de `abadía` y `abadesa` es el término `abad`, que se encontraría, previsiblemente, en el diccionario.

Esta explicación permite ver que la representación puede hacerse en función de árboles de prefijos, de manera que, cada nodo interno representase un carácter y un recorrido por el árbol representase el prefijo común mayor de todas las hojas que sean hijas del nodo donde dicho recorrido finalice. Véase que sería conveniente que el árbol tuviese como raíz común la cadena vacía. En esta representación, sólo las hojas del árbol contendrían palabras.

Una representación alternativa permitiría que cada nodo contuviese una lista (ordenada lexicográficamente) de términos cuyo prefijo común mayor fuese el marcado por el recorrido desde la raíz del árbol hasta dicho nodo.

Además, cada término (bien cada hoja bien cada nodo interno según se elija una de las representaciones anteriormente discutidas) dispondría de una lista de definiciones (también cadenas de caracteres) que podrían estar ordenadas por frecuencia de uso en el lenguaje del diccionario.

Nótese que esta representación, cuyo detalle dejamos para el lector de esta solución (**ese detalle es imprescindible para la solución en un examen**), permite que el acceso a un término sea proporcional en número de operaciones (pasos del recorrido por el árbol) a la longitud (medida en número de caracteres) del término. Como cada palabra requiere un camino de su misma longitud, no es preciso equilibrar el árbol, puesto que dicho espacio, con la excepción del heurístico explicado en el párrafo siguiente, no se podría reducir al ser propio del problema.

Por último, véase que sería admisible que un nodo contuviese no un carácter sino una cadena de estos para reducir la altura del árbol en el caso de que un cierto prefijo no tuviese términos sino que fuese, simplemente, una parte de un camino (un prefijo propio) que llevase hasta términos del diccionario.

b) (5 puntos) Implemente todos los metodos de la interfaz `DictionaryIF`

Para este ejercicio y como en el caso del anterior, sólo ofreceremos indicaciones para la construcción de la implementación.

DictionaryIF

```
/** Representa un diccionario de terminos (palabras)
* cada una de las cuales incluye una colección de
* acepciones  */
public interface DictionaryIF{

 /** Añade una definición a una palabra del diccionario
  *
  * algoritmo: se trataría de
  *    1.- encontrar el término (buscar el camino comparando los
  *        caracteres en los hijos del nodo con el siguiente de la
  *        cadena buscada
  *    2.- si existe el término: añadir la definición a su lista
  *        si no, añadir el término y una lista con la definición
  *        dada como parámetro como único elemento de la misma.
  */
 public void addDefinition (String term, String definition);
```

```
    /** Devuelve la primera definición del término
     *
     * algoritmo:
     *    1.- encontrar el término
     *    2.- devolver la primera definición de su lista
     */
    public String getDefinition (String term);

    /** Devuelve la index-ésima definición del término
     *
     * algoritmo:
     *    1.- encontrar el término
     *    2.- recorrer su lista de definiciones hasta
     *        encontrar la index-ésima y devolverla
     *        -> comprobar la longitud de la lista antes de
     *        realizar el recorrido y devolver un error si
     *        fuese menor que index
     */
    public String getDefinition (String term, int index);

    /** Devuelve la lista completa de definiciones del término
     *
     * algoritmo:
     *    1.- encontrar el término
     *    2.- devolver la lista de definiciones (una referencia
     *        o una copia, según se decida implementar
     */
    public ListIF<String> getDefinitions (String term);
}
```

Véase que la operación más importante, que es común a todas las demás, es la de encontrar un término, por lo cual, podría hacerse como un método privado. Su trabajo sería recorrer el diccionario desde su raíz eligiendo a cada paso el camino congruente con el siguiente carácter del término buscado.

c) (0'5 puntos) Obtenga el coste asintótico temporal en el caso peor del metodo `getDefinitions`

En este apartado discutiremos de forma abstracta este coste, ya que no hemos dado una implementación para el método.

En primer lugar el espacio de búsqueda, puede establecerse como el conjunto de términos del diccionario. Sin embargo, el tamaño del problema tiene que ver con el del término que se busque, es decir, con la longitud de la cadena de caracteres que lo representa, longitud que marcará la del camino hasta encontrar el término. Además, para decidir el siguiente paso, en cada nodo, habrá que comparar el carácter siguiente con los posibles para ese nodo, que corresponderán a los distintos comienzos de los sufijos de los términos cuyo prefijo común sea el camino ya recorrido.

En este punto, hay que hacer varias suposiciones o estimaciones:

- máximo número de caracteres de la palabra más larga del diccionario: sea `m = term.length` y $M = max(m \mid term \in dictionary)$
- máximo número de opciones (sufijos posibles) para un camino a partir de un nodo dado: sea `o=aTree.getChildren().length` y $O = max(o \mid nodo \in diccionario)$

El coste de obtener la lista de definiciones como una referencia sería proporcional al producto de ambos términos en el caso peor, es decir, $T(M) \in M \times O$. Si la lista se obtuviese copiando los elementos uno a uno, habría que incluir un factor más correspondiente a la máxima longitud (en número de definiciones) de cualquiera de estas listas.

Nótese que, aunque esto no es materia de esta asignatura, podría estimarse un coste promedio definiendo una distribución de probabilidad sobre m (cuyo máximo sería M) y sobre o (con máximo O).

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 ();
}
```