Departamento de Lenguajes y Sistemas Informáticos

**Material permitido:** NINGUNO. **Duración:** 2 horas

**Entregue este folio con sus datos consignados**

| | |
|---|---|
| **Alumno:** | **Identificación:** |
| **C. Asociado en que realizó la Práctica Obligatoria:** | |

**P1** (1'5 puntos) **Práctica**. Explique cómo debería variar el algoritmo de salida de vehículos y las estructuras de datos que le den soporte si se debiese garantizar que un vehículo determinado *nunca* espera más de un cierto tiempo (de cortesía) para salir del aparcamiento desde que abandona su plaza

---

1. (1'5 puntos) Diga si cada una de estas afirmaciones es cierta o falsa y por qué:

   *a*) Un algoritmo de coste $\mathcal{O}(n)$ siempre tardará menos en ejecutarse que un algoritmo de coste $\mathcal{O}(n\log(n))$

   *b*) Un algoritmo de coste $\Theta(n)$ siempre tardará menos en ejecutarse que un algoritmo de coste $\Theta(n\log(n))$

   *c*) Si el coste asintótico temporal de un algoritmo es $\mathcal{O}(n)$, entonces también es $\mathcal{O}(n\log(n))$

   *d*) Un algoritmo de coste asintótico temporal $\mathcal{O}(n)$ puede tardar un tiempo arbitrariamente más grande en ejecutarse que otro del mismo coste asintótico temporal

   *e*) Siempre hay un tamaño de problema a partir del cual es más eficiente usar el algoritmo de menor coste asintótico temporal

2. Llamemos diccionario a un inventario de palabras que permita consultar si una palabra pertenece o no a él. La interfaz del tipo sería:

   **DictionaryIF**

```
/* Representa un inventario de palabras                          */
public interface DictionaryIF {

 /* Consulta si el diccionario contiene una palabra [1 punto]    */
 /* @returns cierto si y sólo si la palabra está en el diccionario */
 /* @param la palabra                                            */
 public boolean exists (String i);

 /* Añade una palabra al diccionario [1 punto]                   */
 /* @param la palabra                                            */
 public void addword (String i);

 /* Elimina una palabra del diccionario. Si no pertenece al      */
 /* diccionario, no hace nada. [1'5 puntos]                      */
 public void removeword (String i)
}
```

   *a*) (1 punto) Supondremos que el número de palabras del diccionario puede ser muy grande. ¿Cómo realizaría la representación interna de este tipo para que el tiempo de búsqueda dependa de la longitud de la palabra pero no del número de palabras del diccionario? Justifique su elección. [Sugerencia: piense en un árbol cuyos nodos sean caracteres, y utilice un carácter reservado como señal de fin de palabra]

   *b*) (0'5 puntos) Detalle el constructor de una clase que implemente esta interfaz. Debe haber un parámetro opcional que permita pasar una listado inicial de palabras.

---

*c*) (3'5 puntos) Implemente, basándose en su elección, los métodos de la interfaz `DiccionarioIF` (la puntuación de cada método se detalla en la exposición de la interfaz).

*d*) (1 punto) Calcule el coste asintótico temporal en el caso peor de cada uno de los métodos implementados en función del número de palabras del diccionario ($p$), el tamaño de la palabra de entrada ($t$), y el tamaño del alfabeto ($a$) (entendemos por alfabeto la lista de caracteres permitidos en una palabra).

*e*) (1 punto) ¿Cómo depende el tamaño necesario en memoria para representar un diccionario del tamaño del alfabeto ($a$), del número de palabras ($p$) y de la longitud máxima de las palabras ($l$)?

## **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 ();
}
```