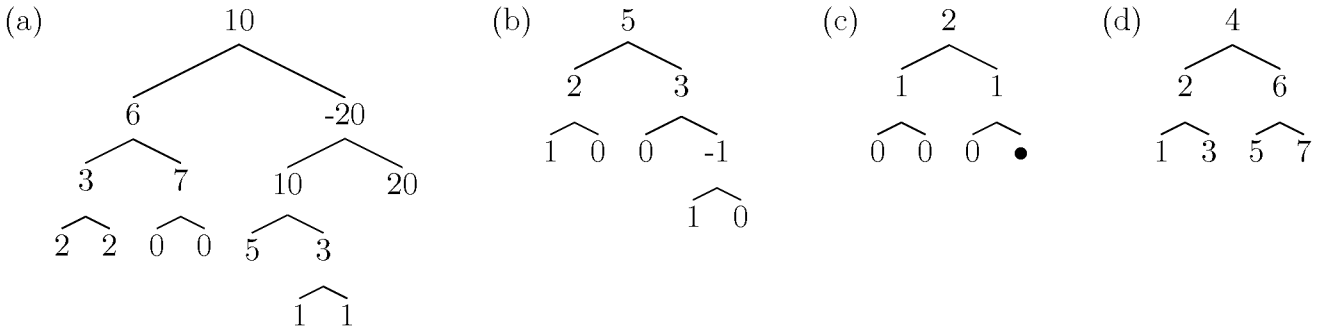
P1 (2 puntos) **Práctica**. Suponga la implementación en árbol de caracteres de la clase `QueryDepotTree`. Codifique un algoritmo que calcule el prefijo más frecuente (o uno cualquiera de ellos si hubiera varios) de longitud mayor o igual que un parámetro (*length*) dado. Por ejemplo, para las entradas `caso, casa, casos, caso, casa, cosa`, y para *length* = 3, la respuesta sería el par <"*cas*", 5>, ya que el prefijo "cas" lo es de cinco entradas y no hay ningún otro con longitud mayor que tenga más entradas. La signatura del metodo debera ser la siguiente:

```
Query getLongestCommonPrefix (int length).
```

1. (1 punto) Se dice que un árbol binario de enteros es *equitativo* (respecto a la adición) si la suma de su mitad izquierda es idéntica a la de su mitad derecha (nótese que no se incluye la suma del nodo raíz en ninguna de sus mitades). Esta definición puede aplicarse a cada nodo, por lo que podremos hablar de un árbol binario *completamente equitativo* si para cada nodo, se cumple la propiedad de ser equitativo. Por ejemplo, dados los siguientes árboles binarios (los nodos marcados como • representan hojas vacías) los árboles etiquetados (a) y (c) son completamente equitativos, el (b) es equitativo en su raíz (pero no completamente equitativo) y el (d) no es equitativo.



Codifíquese en JAVA un método `boolean equitable ()` como un enriquecimiento sobre la interfaz `BTreeIF` (es decir, un nuevo método que se incorpora a la interfaz) que devuelva cierto si un árbol binario es *completamente* equitativo.

2. Una lista autoorganizada (*self-organising list*, en inglés) es una lista que reordena sus elementos basándose en alguna heurística de autoorganización para mejorar el tiempo de acceso medio. El propósito de estas listas es aumentar la eficiencia de la búsqueda lineal moviendo los elementos más frecuentemente accedidos hacia la cabeza de la lista. Una lista autoorganizada con una heurística apropiada logra un tiempo *casi* constante de acceso a los elementos en el caso mejor. Así, se trata de utilizar un algoritmo que reorganice los elementos de la lista en función de cuáles sean las consultas más frecuentes sobre los elementos de la misma (qué elementos se busquen más a menudo) y esto es algo que se hace en tiempo de ejecución. En cuanto a las estrategias para conseguirlo, consideramos la de *Conteo (CM)*, en la que cada elemento tiene asociado un contador de accesos. Cuando se accede a un elemento, se incrementa dicho contador y, si procede, se reorganiza la lista según este nuevo valor (por ejemplo, si dos nodos ubicados en posiciones sucesivas de la lista tenían igual conteo de accesos y se accede al segundo de ellos, ambos deberán intercambiar sus posiciones en la lista. La interfaz del tipo se define a continuación:

SelfOrgListIF

```
/*  Representa una lista autoorganizada de elementos                    */
/*  cualesquiera.  La lista se mantiene siempre en orden descendente    */
/*  de frecuencia de consulta de sus elementos.                         */
public interface SelfOrgListIF<T>{

/* [0'5 puntos] devuelve el número de elementos de la lista             */
public int size ();

/* [1 punto] Añade el elemento a la lista conservando las               */
/* propiedades (elementos ordenados según frecuencia de acceso)        */
/* @param el elemento que se quiere añadir                              */
public void add (T elem);

/* [1 punto] Elimina un elemento de la lista conservando las            */
*/ propiedades (elementos ordenados según frecuencia de acceso)        */
/* @param el elemento que se desea eliminar de la lista                 */
 public void remove (T elem);

/* [2 puntos] Comprueba si la lista contiene el elemento buscado        */
/* @param el elemento que se quiere buscar                              */
public boolean contains(T elem);
}
```

a) (1 punto) Detalle el constructor de una clase que implemente esta interfaz. Describa detalladamente cómo realizaría la representación interna de este tipo. Justifique su elección.

b) (4'5 puntos) Basándose en la respuesta anterior, implemente todos los métodos de la interfaz `SelfOrgListIF`. Deberá también implementar, como métodos privados, todos aquellos que necesite como auxiliares para dar soporte a los métodos de la interfaz. Se valorará la eficiencia. (Nota: las puntuaciones asignadas a cada método se indican en la especificación de dicho método en la interfaz del tipo)

c) (1'5 puntos) Calcule el coste asintótico temporal en el caso *medio* para la implementación realizada del método **boolean** `contains(T elem)` . Detállense **todas** las suposiciones necesarias para realizar dicho cálculo (por ejemplo, cuán común es que haya elementos distintos con la misma frecuencia de consulta, lo que puede expresarse como una probabilidad, es decir $p_f \in [0, 1]$). Puede apoyarse en un ejemplo o plantear y resolver la recurrencia de forma aproximada (sin utilizar la fórmula de cálculo).

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 ();
}
```