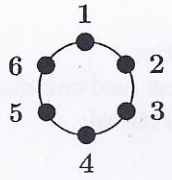
Algoritmos e Estruturas de Dados
Exercícios das Aulas Práticas Com Frequência

Departamento de Informática
Faculdade de Ciências e Tecnologia, UNL
Ano Lectivo 2010/11

1. (Normal 2010) No problema de Josephus, há n pessoas, numeradas de 1 a n, dispostas em círculo. Começando na pessoa 1, percorre-se o círculo (as vezes necessárias) eliminando pessoas, até ficar apenas uma que se salva. A pessoa a ser eliminada é sempre a segunda que se encontra no percurso circular (que não tem descontinuidades).



Por exemplo, se $n = 6$, na primeira volta pelo círculo, eliminam-se as pessoas **2**, **4** e **6**. Depois, (encontra-se a pessoa **1** e) é eliminada a **3**, (encontra-se a pessoa **5** e) é eliminada a **1**. Neste caso, salva-se a pessoa **5**.

A função *josephus* calcula quem se salva, quando o número de pessoas é n.

Só para ver se parâmetro é válido. Não custa nada

```
public static int josephus( int n ) throws NonPositiveNumberException
{
    if ( n <= 0 )
        throw new NonPositiveNumberException();

    return josephRec(n);
}

private static int josephRec( int n )
{
    if ( n == 1 )
        return 1;
    else if ( n % 2 == 0 )
        return 2 * josephRec(n / 2) - 1;
    else (impar e parte de 3)
        return 2 * josephRec(n / 2) + 1;
}
```

O custo do josephus = custo josephRec como mesmo argumento

Custo
1 (constante).
1 independentemente do n, o nº de passos é sempre o mesmo, faz-se sem 1 mul, uma div e uma sub. Não o que varia é o resultado.
1

Determine a complexidade temporal do método *josephus*, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

é constante dentro de cada passo de achinação, depois temos de contar o nº de chamadas recursivas para calcular a complexidade.

D10

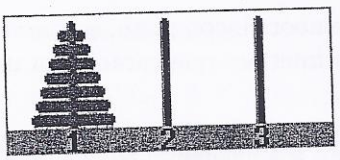
2. (Normal 2008) Considere a função pública *max*, que calcula o máximo de um vector de elementos do tipo E, de acordo com o comparador fornecido, pela técnica da divisão e conquista.

```
public static <E> E max( E[] vector, Comparator<E> comp )
    throws EmptyArrayException
{
    if ( vector.length == 0 )
        throw new EmptyArrayException();
    return max(vector, comp, 0, vector.length - 1);
}

// Precondition: firstPos <= lastPos.
private static <E>
    E max( E[] vector, Comparator<E> comp, int firstPos, int lastPos )
{
    if ( firstPos == lastPos )
        // The vector has just one element.
        return vector[firstPos];
    else
    {
        // The vector has at least two elements.
        int middlePos = ( firstPos + lastPos ) / 2;
        E maxLeft = max(vector, comp, firstPos, middlePos);
        E maxRight = max(vector, comp, middlePos + 1, lastPos);
        return maxOf2(maxLeft, maxRight, comp);
    }
}

public static <E>
    E maxOf2( E element1, E element2, Comparator<E> comp )
{
    if ( comp.compare(element1, element2) >= 0 )
        return element1;
    else
        return element2;
}
```

Determine a complexidade temporal do método público *max* quando este é chamado com um vector de n elementos (com $n \geq 1$), no melhor caso, no pior caso e no caso esperado, assumindo que a complexidade da comparação de dois elementos do tipo E é sempre constante. Justifique todos os cálculos com muita clareza.

3. (Normal 2004) No Problema das Torres de Hanoi, existem $n \geq 1$ discos, todos de diâmetros diferentes, e 3 estacas (denominadas 1, 2 e 3). Pretende-se deslocar os n discos, que se encontram *em pirâmide* na estaca 1, para a estaca 3,

   - movendo um disco de cada vez e
   - não podendo nunca colocar um disco maior sobre um disco menor.



Uma sequência de movimentos que resolve o problema pode ser obtida pelo seguinte algoritmo.

```
public static void hanoi( int numberOfDisks )
{
    if ( numberOfDisks >= 1 )
        hanoi(numberOfDisks, 1, 3, 2);
}

private static
    void hanoi( int nDisks, int source, int destination, int theOther )
{
    if ( nDisks == 1 )
        // Mover o disco da origem para o destino.
        moveDisk(source, destination);
    else
    {
        // Mover os nDisks−1 discos menores da origem para a auxiliar.
        hanoi(nDisks − 1, source, theOther, destination);
        // Mover o disco maior da origem para o destino.
        moveDisk(source, destination);
        // Mover os nDisks−1 discos menores da auxiliar para o destino.
        hanoi(nDisks − 1, theOther, destination, source);
    }
}

private static void moveDisk( int source, int destination )
{
    System.out.print("Mova o disco (do topo) da estaca " + source);
    System.out.println(" para a estaca " + destination);
}
```

Determine a complexidade temporal do método público *hanoi*, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

4. (Recurso 2007) A Transformada Rápida de Fourier, geralmente abreviada por *FFT* (de *Fast Fourier Transform*), tem inúmeras aplicações em áreas tão variadas como o processamento de sinal, o processamento de imagem ou a aritmética sobre polinómios. A FFT transforma um vector de números complexos noutro vector de números complexos (com a mesma dimensão). É necessário que a dimensão do vector a transformar seja uma potência de dois.

   O código que se segue, embora incompleto, esquematiza a implementação da FFT, onde *ComplexNumber* é uma interface que caracteriza um número complexo.

```
// Computes the FFT of the specified array, which has vec.length
// complex numbers (where vec.length is a power of 2).
public static void fft( ComplexNumber[] vec )
{
    permute(vec);
    fft(vec, 0, vec.length − 1);
}


private static void fft( ComplexNumber[] vec, int firstPos, int lastPos )
{
    if ( firstPos < lastPos )
    {
        int middle = ( firstPos + lastPos ) / 2;
        fft(vec, firstPos, middle);
        fft(vec, middle + 1, lastPos);
        combine(vec, firstPos, lastPos);
    }
}


// Performs the adequate permutation of the specified array.
// This method runs in O(vec.length) time,
// in the best-case and in the worst-case.
private static void permute( ComplexNumber[] vec );


// Performs the combination of two FFTs in the specified array,
// from the first specified position to the last specified position.
// This method runs in O(lastPos − firstPos + 1) time,
// in the best-case and in the worst-case.
private static void combine( ComplexNumber[] vec, int firstPos, int lastPos );
```

   Determine a complexidade temporal do método público *fft*, quando este é chamado com um vector de n números complexos (onde n é uma potência de dois), no melhor caso, no pior caso e no caso esperado. Justifique todos os cálculos com muita clareza.

Aula 2

5. (Normal 2006) Pretende-se aplicar a técnica da função-memória ao seguinte problema de sequências de caracteres.

   Seja $X = (x_1, x_2, \ldots, x_m)$, com $m \geq 1$, uma sequência não vazia de caracteres. Uma *sub-sequência de X de dimensão k* (onde $k \geq 0$) é uma sequência de elementos de X que pode ser obtida seleccionando k elementos de X, da esquerda para a direita.

   Por exemplo, (b, a, b) é uma sub-sequência de (b, d, c, a, b, a) de dimensão 3.

   Dadas duas sequências não vazias de caracteres,

   $$X = (x_1, x_2, \ldots, x_m) \text{ e } Y = (y_1, y_2, \ldots, y_n),$$

   uma *sub-sequência comum a X e Y de dimensão k* é uma sub-sequência de X e de Y, de dimensão k.

   Por exemplo, se as sequências dadas forem

   $$X = (b, d, c, a, b, a) \text{ e } Y = (a, b, c, b, d, a, b),$$

   quer (b, a, b), quer (b, c, a, b), são sub-sequências comuns a X e Y, a primeira de dimensão 3 e a segunda de dimensão 4. Neste caso, 4 é a dimensão das maiores sub--sequências comuns a X e Y.

   Dadas duas sequências não vazias de caracteres, X e Y, pretende-se descobrir a dimensão das maiores sub-sequências comuns a X e Y. Esse valor pode ser calculado pelo seguinte algoritmo (onde *LCS* denota *Longest Common Subsequence*).

```
public static int LCS( char[] seq1, char[] seq2 )
{
    return LCS(seq1, seq2, seq1.length, seq2.length);
}

private static int LCS( char[] seq1, char[] seq2, int length1, int length2 )
{
    if ( length1 == 0 || length2 == 0 )
        return 0;
    else if ( seq1[length1 - 1] == seq2[length2 - 1] )
        return 1 + LCS(seq1, seq2, length1 - 1, length2 - 1);
    else
        return Math.max( LCS(seq1, seq2, length1, length2 - 1),
                         LCS(seq1, seq2, length1 - 1, length2) );
}
```

   O problema é que a complexidade temporal deste algoritmo é exponencial. Dados dois vectores de caracteres, *seq1* e *seq2*, no cálculo de *LCS(seq1,seq2)*, executam-se muitas chamadas recursivas iguais. Repare que, em todas elas, os dois primeiros argumentos são iguais.

   Apresente um algoritmo polinomial que calcule o mesmo valor, aplicando a técnica da função-memória ao algoritmo dado. Determine as complexidades temporal e espacial do seu algoritmo, no melhor caso e no pior caso, justificando todos os cálculos com muita clareza.

6. (Normal 2009) A função *fun*, definida para números inteiros não negativos, tem complexidade exponencial. Apresente um algoritmo polinomial que calcule o mesmo valor, aplicando a técnica da função-memória ao algoritmo dado.

```
public static double fun( int n ) throws NegativeNumberException
{
    if ( n < 0 )
        throw new NegativeNumberException();
    return funRec(n);
}

private static double funRec( int n )
{
    if ( n < 3 )
        return n;
    else
        return Math.sqrt(n) + funRec(n - 1) * funRec(n - 3);
}
```

Determine as complexidades temporal e espacial do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

7. (Recurso 2010) A função *noName*, definida para números inteiros não negativos, tem complexidade exponencial. Apresente um algoritmo polinomial que calcule o mesmo valor, aplicando a técnica da função-memória ao algoritmo dado.

```
public static long noName( int n ) throws NegativeNumberException
{
    if ( n < 0 )
        throw new NegativeNumberException();
    return noNameRec(n);
}

private static long noNameRec( int n )
{
    if ( n == 0 || n == 1 )
        return 2 * n + 1;
    else
    {
        long prod = 1;
        for ( int i = 0;  i < n / 2;  i++ )
            prod = prod * ( noNameRec(n - i - 1) % noNameRec(i) + 1 );
        return prod;
    }
}
```

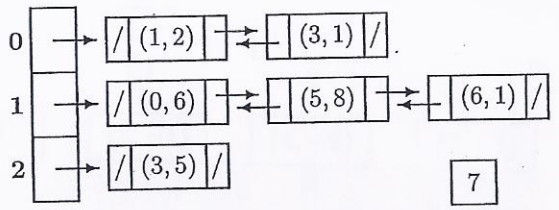
Determine as complexidades temporal e espacial do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

8. (Normal 2010) Uma matriz diz-se *esparsa* quando um grande número de elementos é zero. Para representar uma matriz deste género, basta guardar os elementos que são diferentes de zero. A classe *SparseMatrix* implementa matrizes esparsas de elementos do tipo E da seguinte maneira (ilustrada de forma muito simplificada na figura).

   - O número de colunas é guardado num inteiro.
   - As linhas da matriz são guardadas num vector (com capacidade igual ao número de linhas). Em cada posição do vector, há uma lista duplamente ligada (com cabeça e cauda) que guarda os elementos dessa linha, por ordem crescente de coluna. Cada célula da lista ligada possui uma entrada cuja chave (do tipo *Integer*) indica a coluna e cujo valor (do tipo E) é o elemento da matriz (na linha e na coluna correspondentes).

   Para simplificar, considera-se que as linhas e as colunas começam em zero, ou seja, o elemento da matriz do canto superior esquerdo está na linha zero e na coluna zero.

$$\begin{bmatrix} 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 8 & 1 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \end{bmatrix}$$



O método *transpose*, quando aplicado a uma matriz M, cria e devolve a matriz transposta de M, sem alterar M (ou seja, M.transpose() retorna M^T). Recorde que a primeira linha de M é a primeira coluna de M^T, a segunda linha de M é a segunda coluna de M^T, etc. No nosso exemplo,

$$M = \begin{bmatrix} 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 8 & 1 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 \end{bmatrix} \quad \text{e} \quad M^T = \begin{bmatrix} 0 & 6 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 5 \\ 0 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

```
public class SparseMatrix<E>
{
   protected int numberOfColumns;

   // Every List is implemented with a DoublyLinkedList.
   protected List<Entry<Integer,E>>[] lines;
   .....

   // Creates and returns the transpose of this matrix.
   public SparseMatrix<E> transpose( );
}
```

Implemente o método *transpose*, programando todos os métodos públicos ou auxiliares da classe *SparseMatrix* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

DoubleLL
Árvore binária de pesquisa | Sei lá outra manipulação

9. (Recurso 2009) Um polinómio de coeficientes inteiros pode ser implementado através de uma lista de pares do tipo <Integer, Integer>. As componentes do par representam, respectivamente, o grau e o coeficiente do termo do polinómio. Se se percorrer os elementos da lista pela ordem natural (ou seja, das posições menores para as maiores), os termos do polinómio ocorrem por ordem estritamente crescente de grau. Não existem coeficientes nulos (nem termos com o mesmo grau).

   A figura seguinte exemplifica esta implementação com três polinómios. Do lado esquerdo, o polinómio está representado na forma usual e, do lado direito, apresenta-se a sequência de pares guardada na lista. A posição de cada par na lista está escrita a *bold* por baixo do par.

$4+2x+5x^4-x^7$	$(0,4)$ **0**	$(1,2)$ **1**	$(4,5)$ **2**	$(7,-1)$ **3**
$-2x-x^2-8x^4$	$(1,-2)$ **0**	$(2,-1)$ **1**	$(4,-8)$ **2**	
$4-x^2-3x^4-x^7$	$(0,4)$ **0**	$(2,-1)$ **1**	$(4,-3)$ **2**	$(7,-1)$ **3**

   Repare que o último polinómio pode ser obtido somando os dois primeiros.

   Considere a classe *Polynomial*, dos polinómios de coeficientes inteiros implementados através de uma lista duplamente ligada, com cabeça e cauda. Ou seja, o atributo *list* é um objecto da classe *DoublyLinkedList*.

   O método *sum* constrói e retorna o polinómio soma, que se obtém somando o polinómio passado em argumento ao polinómio que recebe a mensagem. O polinómio soma não partilha posições de memória com os polinómios somados, que não podem ser alterados. Para simplificar, assuma que nenhum dos polinómios é 0 (zero).

```
package myMath;

public class Polynomial
{
    // The polynomial is stored in a list.
    protected List<Pair<Integer, Integer>> list;

    ..........................................

    // Builds and returns the sum of the specified polynomial
    // with this polynomial.
    public Polynomial sum( Polynomial polynomial );
}
```

   Implemente o método *sum* e programe todos os métodos auxiliares da classe *Polynomial* de que necessitar. Note que a classe *Polynomial* não pertence ao pacote *dataStructures*. Calcule a complexidade temporal do seu algoritmo, no caso esperado, justificando.

Eduardo

10. (Normal 2007) Descobriu que os algoritmos de inserção e de remoção das listas duplamente ligadas não estão a afectar correctamente o apontador para o nó anterior. Felizmente, a cabeça, a cauda e o número de elementos da lista, bem como todos os apontadores para o nó seguinte, estão correctos.

    Relembre então as classes *DListNode*, dos nós das listas duplamente ligadas de elementos do tipo E, e *DoublyLinkedList*, das listas duplamente ligadas com cabeça e cauda de elementos do tipo E. As duas classes pertencem ao pacote *dataStructures*.

```
class DListNode<E>
{
    private E element;                  // Element stored in the node.
    private DListNode<E> previous;      // (Pointer to) the previous.
    private DListNode<E> next;          // (Pointer to) the next node.

    public DListNode( E theElement );
    public DListNode( E theElement, DListNode<E> thePrevious,
                      DListNode<E> theNext );
    public E getElement( );
    public DListNode<E> getPrevious( );
    public DListNode<E> getNext( );
    public void setElement( E newElement );
    public void setPrevious( DListNode<E> newPrevious );
    public void setNext( DListNode<E> newNext );
}

public class DoublyLinkedList<E> implements List<E>
{
    protected DListNode<E> head;        // Node at the head of the list.
    protected DListNode<E> tail;        // Node at the tail of the list.
    protected int currentSize;          // Number of elements in the list.
    ..........................................
    protected void restoreAllPreviousPointers( );
}
```

    Implemente o método *restoreAllPreviousPointers*, que restabelece a coerência da lista, e programe todos os métodos auxiliares da classe *DoublyLinkedList* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

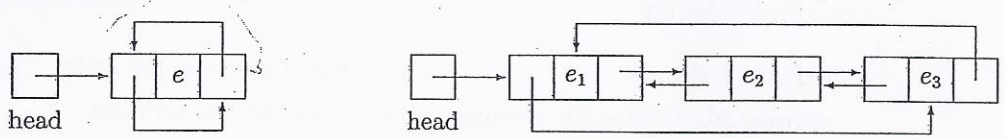
```
DListNode<E> node = head;
DListNode<E> aux = null;
while (node != null) {
    node.setPrevious(aux);
    aux = node;
    node = node.getNext(); }
```

complexidade
O(n)

11. (Normal 2009) Uma **lista circular duplamente ligada com cabeça** é uma estrutura de dados muito semelhante a uma lista duplamente ligada com cabeça. A única diferença é que os nós da lista circular nunca têm *null*. Mais precisamente:

- o anterior do primeiro nó da lista aponta para o último nó da lista; e
- o seguinte do último nó da lista aponta para o primeiro nó da lista.

A figura ilustra a forma de duas listas circulares duplamente ligadas com cabeça, uma com um elemento e a outra com três elementos.



Relembre então a classe *DListNode*, dos nós das listas duplamente ligadas de elementos do tipo E, e considere a classe *CircularDoublyLinkedList*, das listas circulares duplamente ligadas com cabeça de elementos do tipo E. As duas classes pertencem ao pacote *dataStructures*.

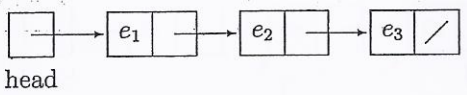
```
class DListNode<E>
{
  private E element;                  // Element stored in the node.
  private DListNode<E> previous;      // (Pointer to) the previous.
  private DListNode<E> next;          // (Pointer to) the next node.

  public DListNode( E theElem );
  public DListNode( E theElem, DListNode<E> thePrev, DListNode<E> theNext );
  public E getElement( );
  public DListNode<E> getPrevious( );
  public DListNode<E> getNext( );
  public void setElement( E newElement );
  public void setPrevious( DListNode<E> newPrevious );
  public void setNext( DListNode<E> newNext );
}

public class CircularDoublyLinkedList<E>
{
  protected DListNode<E> head;     // Node at the head of the list.
  ..........................................

  // Inserts the specified element at the last position in the list.
  public void addLast( E element );

  // Removes and returns the element at the last position in the list.
  public E removeLast( ) throws EmptyListException;
}
```

Implemente os métodos *addLast* e *removeLast*, programando todos os métodos auxiliares da classe *CircularDoublyLinkedList* de que necessitar. Note que a classe só tem uma variável de instância, chamada *head*. Calcule as complexidades temporais dos seus dois algoritmos, no melhor caso, no pior caso e no caso esperado, justificando.

12. (Recurso 2010) Considere a classe *OrderedSimplyLinkedList*, das listas simplesmente ligadas com cabeça, ordenadas e sem elementos repetidos, de elementos do tipo E. Esta classe tem apenas um atributo (*head*) que guarda o primeiro nó da lista.

Como a lista é:

- *simplesmente ligada*, cada nó da lista tem apenas o elemento E e o apontador para o nó seguinte;
- *ordenada e sem elementos repetidos*, os elementos estão por ordem estritamente crescente (ou seja, $e_1 < e_2 < e_3$, no exemplo ilustrado na figura).

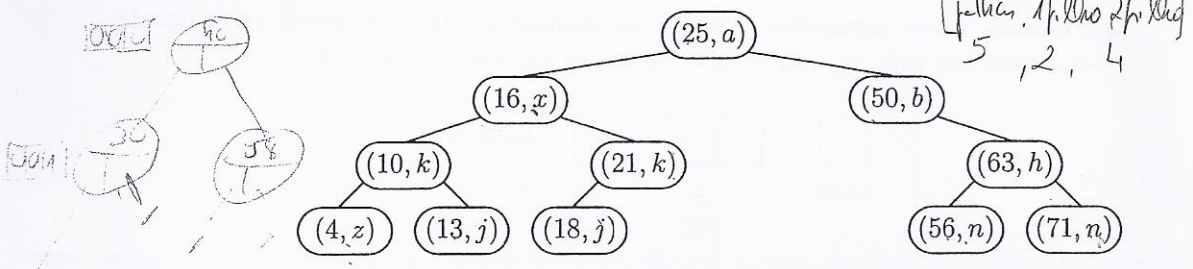


A classe *SListNode*, dos nós das listas simplesmente ligadas de elementos do tipo E, e a classe *OrderedSimplyLinkedList* pertencem ao pacote *dataStructures*.

```
class SListNode<E>
{
    private E element;                  // Element stored in the node.
    private SListNode<E> next;          // (Pointer to) the next node.

    public SListNode( E theElement );
    public SListNode( E theElement, SListNode<E> theNext );

    public E getElement( );
    public SListNode<E> getNext( );
    public void setElement( E newElement );
    public void setNext( SListNode<E> newNext );
}

public class OrderedSimplyLinkedList<E extends Comparable<E>>
{
    protected SListNode<E> head;        // Node at the head of the list.
    ..........................................

    // Inserts the specified element in the linked list
    // if it is not already present.
    public void insert( E element ) throws RepeatedElementException;
}
```

Implemente o método *insert*, programando todos os métodos auxiliares da classe *OrderedSimplyLinkedList* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo E, é sempre constante.

13. (Recurso 2010) O método *statistics*, da classe *BinarySearchTree*, retorna um vector com três inteiros: a posição zero do vector tem o número de folhas da árvore; a posição um do vector tem o número de nós da árvore com exactamente um filho; e a posição dois do vector tem o número de nós da árvore com dois filhos. No caso da árvore esquematizada na figura, esses inteiros seriam, respectivamente, 5, 2 e 4. Repare que, se a árvore for vazia, os três números são zero.



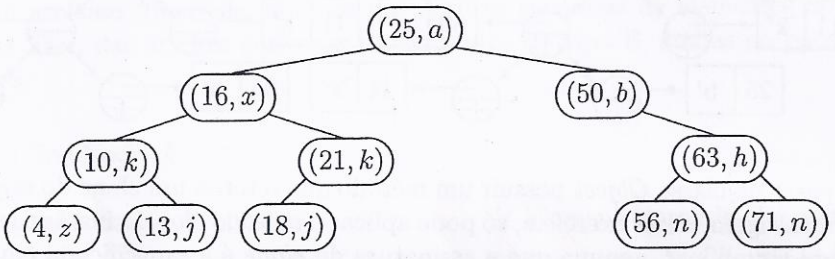

```
class BSTNode<K,V>
{
    private EntryClass<K,V> entry;       // Entry stored in the node.
    private BSTNode<K,V> leftChild;      // (Pointer to) the left child.
    private BSTNode<K,V> rightChild;     // (Pointer to) the right child.

    public BSTNode( K key, V value, BSTNode<K,V> left, BSTNode<K,V> right )
    public BSTNode( K key, V value );
    public EntryClass<K,V> getEntry( );
    public K getKey( );
    public V getValue( );
    public BSTNode<K,V> getLeft( );
    public BSTNode<K,V> getRight( );
    public void setEntry( EntryClass<K,V> newEntry );
    public void setEntry( K newKey, V newValue );
    public void setKey( K newKey );
    public void setValue( V newValue );
    public void setLeft( BSTNode<K,V> newLeft );
    public void setRight( BSTNode<K,V> newRight );
    public boolean isLeaf( );
}

public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    protected BSTNode<K,V> root;         // The root of the tree.
    protected int currentSize;           // Number of entries in the tree.
    ..........................................

    public int[] statistics( );
}
```

Implemente o método *statistics*, programando todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

14. (Normal 2010) O método *pairsOfSiblingsWithTheSameValue*, da classe *BinarySearchTree*, retorna o número de pares de nós irmãos que possuem o mesmo valor. No caso da árvore esquematizada na figura, só há quatro pares de nós irmãos e:

    - os irmãos cujas chaves são 16 e 50 têm valores distintos (x e b);
    - os irmãos cujas chaves são 10 e 21 têm ambos valor k;
    - os irmãos cujas chaves são 4 e 13 têm valores distintos (z e j); e
    - os irmãos cujas chaves são 56 e 71 têm ambos valor n.

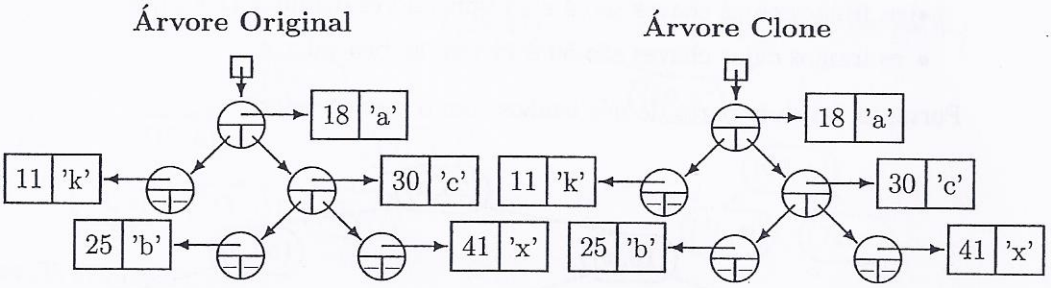
    Portanto, há **dois** pares de nós irmãos com o mesmo valor.




```
public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    // The root of the tree.
    protected BSTNode<K,V> root;

    // Number of entries in the tree.
    protected int currentSize;

    .....

    public int pairsOfSiblingsWithTheSameValue( );
}
```

Implemente o método *pairsOfSiblingsWithTheSameValue* e programe todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do teste à igualdade de dois valores (do tipo V) é sempre constante.

15. (Normal 2009) Considere um novo construtor da classe *BinarySearchTree*, que constrói um clone (uma cópia independente) da árvore binária de pesquisa que é passada em argumento. Como se ilustra na figura, as duas árvores têm exactamente a mesma forma e o mesmo conteúdo, mas são independentes (não partilham posições de memória). A árvore clonada (a original) não é alterada.



Apesar da classe *Object* possuir um método que retorna um clone do objecto que recebe a mensagem, neste exercício, só pode aplicar o método *clone* a instâncias de *EntryClass*. Para simplificar, assuma que a assinatura de *clone* é a especificada em baixo.

```
class EntryClass<K,V> implements Entry<K,V>
{
    private K key;                          // Key stored in the entry.
    private V value;                        // Value stored in the entry.

    public EntryClass( K theKey, V theValue );
    public K getKey( );
    public V getValue( );
    public void setKey( K newKey );
    public void setValue( V newValue );
    // Creates and returns a clone of this entry.
    public EntryClass<K,V> clone( );
}

public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    protected BSTNode<K,V> root;    // The root of the tree.
    protected int currentSize;      // Number of entries in the tree.
    .....

    public BinarySearchTree( BinarySearchTree<K,V> tree );
}
```

Implemente esse construtor e programe todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método *clone*, quando aplicado a instâncias de *EntryClass*, é sempre constante.

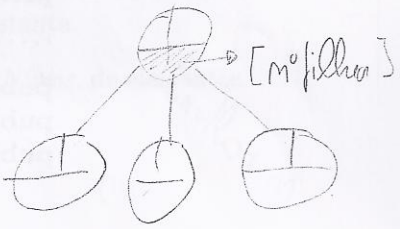
16. (Recurso 2009) Numa árvore genérica, cada nó possui um número arbitrário de filhos. Neste exercício, os filhos estão guardados num vector. Mais especificamente, se um nó tem k filhos (com $k \geq 0$), os filhos estão guardados nas posições $0, 1, \ldots, k-1$ do vector.

O *grau* de uma árvore não vazia é o maior número de filhos que algum nó da árvore possui. A noção de grau não está definida para árvores vazias.

Por exemplo, o grau de uma árvore binária só pode ser: zero (quando a árvore tem um único nó); um (quando a árvore tem pelo menos dois nós e todo o nó da árvore ou é folha ou só tem um filho); ou dois (quando pelo menos um nó da árvore tem dois filhos).

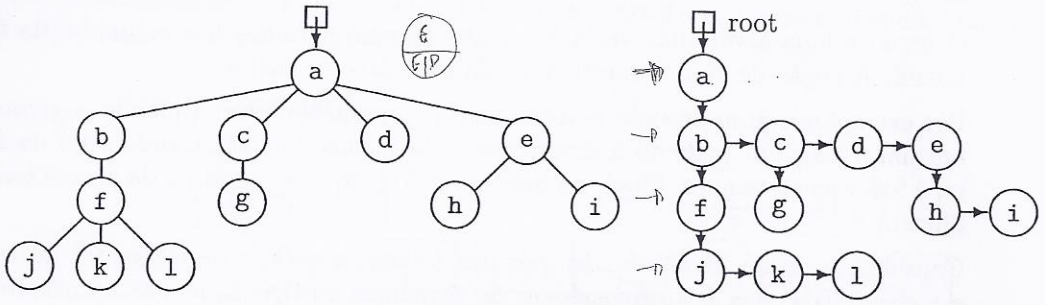
Considere a classe *TreeNode*, dos nós das árvores genéricas de elementos do tipo E, e a classe *Tree*, das árvores genéricas de elementos do tipo E, ambas no pacote *dataStructures*.

```
class TreeNode<E>
{
    protected E element;                    // Element stored in the node.
    protected TreeNode<E>[] children;       // (Pointers to) the children.
    protected int childrenSize;             // Number of children.

    .....

    // Returns the element in the node.
    public E getElement( );

    // Returns the number of children of the node.
    public int getNumberOfChildren( );

    // Returns the child stored at the specified position of
    // the node's array of children.
    public TreeNode<E> getChild( int position )
        throws InvalidPositionException;
}

public class Tree<E>
{
    protected TreeNode<E> root;             // The root of the tree.

    .....

    // Returns the degree of the tree.
    public int getDegree( ) throws EmptyTreeException;
}
```



Implemente o método *getDegree*, que calcula o grau da árvore, programando todos os métodos auxiliares da classe *Tree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

17. (Normal 2006) Relembre que, para implementar árvores cujos nós têm um número arbitrário de filhos, podem-se utilizar células com três campos: *elemento*, *filho esquerdo* e *irmão direito*. Por exemplo, a figura da direita ilustra a implementação da árvore representada na figura da esquerda.



Considere a classe *TreeNode*, dos nós das árvores genéricas de elementos do tipo E, e a classe *Tree*, das árvores genéricas de elementos do tipo E, ambas no pacote *dataStructures*.

```
class TreeNode<E>
{
    private E element;                  // Element stored in the node.
    private TreeNode<E> child;          // (Pointer to) the left child.
    private TreeNode<E> sibling;        // (Pointer to) the right sibling.
    ..........................................

    public E getElement( );
    public TreeNode<E> getChild( );
    public TreeNode<E> getSibling( );
}

public class Tree<E>
{
    protected TreeNode<E> root;         // The root of the tree.
    protected int currentSize;          // Number of elements in the tree.
    protected int currentHeight;        // Height of the tree.
    ..........................................

    public int[] levelSizes( ) throws EmptyTreeException;
}
```

A função *levelSizes* retorna um vector com dimensão igual à altura da árvore. A posição i desse vector contém o número de nós da árvore no nível $i + 1$.

Denotando a árvore representada acima por t, $t.levelSizes()$ retornaria um vector de dimensão quatro, com os inteiros 1, 4, 4 e 3 (nas posicões 0, 1, 2 e 3, respectivamente).

Implemente o método *levelSizes*, programando todos os métodos auxiliares da classe *Tree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. (**Sugestão:** pense numa solução recursiva.)

- Não é possível calcular complexidades com estruturas de dados. só com TADs.

- Descrição - resumo da operação

18. (Recurso 2007) Considere o tipo abstracto de dados *Fila de Espera com Categorias*, de categorias do tipo R e elementos do tipo E, caracterizado pela interface *QueueWithRanks*.

```
public interface QueueWithRanks<R extends Comparable<R>, E>
{
    // Inserts the specified element, with the specified rank,
    // at the rear of the queue.
    void enqueue( R rank, E element );

    // Removes and returns the first element in the queue
    // whose rank is the specified rank.
    E dequeue( R rank ) throws NoSuchElementException;

    // Removes and returns the first element in the queue
    // whose rank is the highest rank of an element in the queue.
    E dequeue( ) throws EmptyQueueWithRanksException;
}
```

& - E = & - insere o elemento vermelho

- entra vermelho, pai o primeiro elemento vermelho na fila

retorna o primeiro do elemento (com a categoria mais alta).

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *QueueWithRanks*, descreva brevemente como implementaria as três operações e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo R, é sempre constante.

19. (Normal 2008) Considere o tipo abstracto de dados *Fila com Dois Níveis*, de elementos do tipo E, caracterizado pela interface *TwoLevelQueue*.

```
public interface TwoLevelQueue<E>
{
    // Inserts the specified element at the rear of the queue,
    // classifying it as a level one element.
    void enqueueLevel1( E element );

    // Inserts the specified element at the rear of the queue,
    // classifying it as a level two element.
    void enqueueLevel2( E element );

    // Removes and returns the element at the front of the queue.
    E dequeue( ) throws EmptyQueueException;

    // Returns an iterator of the level one elements in the queue
    // (in proper sequence).
    Iterator<E> iteratorLevel1( );

    // Returns an iterator of the level two elements in the queue
    // (in proper sequence).
    Iterator<E> iteratorLevel2( );
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *TwoLevelQueue*, descreva brevemente como implementaria as cinco

operações e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando.

20. (Normal 2009) Um domínio D diz-se *auto-representável* se existir uma função

$$\text{rep} : D \longrightarrow D$$

que atribui, a cada elemento $x \in D$, um elemento $\text{rep}(x) \in D$, chamado *o representante* de x. Os elementos $x, y \in D$ *pertencem à mesma família*, se $\text{rep}(x) = \text{rep}(y)$. Qualquer conjunto $C \subseteq D$ é um *subconjunto auto-representável*.

Suponha que os objectos representáveis por elementos do tipo E são caracterizados pela interface *Representable* e considere o tipo abstracto de dados *Subconjunto Auto-Representável*, de elementos do tipo E, definido pela interface *SelfRepresentableSubset*.

```
public interface Representable<E>
{
    // Returns the representative of this object.
    E representative( );
}

public interface SelfRepresentableSubset<E extends Representable<E>>
{
    // Returns the number of elements in the set.
    int size( );

    // Returns true iff the specified element belongs to the set.
    boolean contains( E element );

    // Inserts the specified element in the set.
    void insert( E element ) throws ExistingElementException;

    // Removes the specified element from the set.
    void remove( E element ) throws NoSuchElementException;

    // Returns an iterator of all elements in the set
    // that belong to the same family of the specified element.
    Iterator<E> sameFamily( E element );
}
```

Dic (Representante, Dic (G,C)

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *SelfRepresentableSubset*, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no caso esperado, justificando. Pode assumir que os elementos do domínio são comparáveis entre si (E extends Comparable<E>), com complexidade constante, e que a complexidade do método *representative* é sempre constante.

melhor forma de implementar dicionário (falta de dispersão)
↓
Problema Reach

21. (Recurso 2009) Considere o tipo abstracto de dados *Fila com Prioridade Inteligente*, de chaves do tipo K e valores do tipo V, caracterizado pela interface *SmartMinPriorityQueue*.

```
public interface SmartMinPriorityQueue<K extends Comparable<K>, V>
{
    // Returns the number of entries in the priority queue.
    int size( );

    // Returns an entry with the smallest key in the priority queue.
    Entry<K,V> minEntry( ) throws EmptyPriorityQueueException;

    // If the priority queue contains an entry of the form (otherKey, value),
    // with otherKey ≤ key, this method does nothing except returning false;
    // otherwise,
    // inserts the entry (key, value) in the priority queue and returns true.
    boolean insert( K key, V value );

    // Removes an entry with the smallest key from the priority queue
    // and returns that entry.
    Entry<K,V> removeMin( ) throws EmptyPriorityQueueException;
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *SmartMinPriorityQueue*, descreva brevemente como implementaria as quatro operações e calcule as suas complexidades temporais, no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo K, é sempre constante. Pode assumir que os valores também são comparáveis entre si (V extends Comparable<V>), com complexidade constante.

22. (Normal 2010) Considere o tipo abstracto de dados *Dicionário Ordenado com Informação por Valor*, de chaves do tipo K e valores do tipo V, caracterizado pela interface *OrderedDictionaryWithValueInfo*.

```
public interface OrderedDictionaryWithValueInfo<K extends
    Comparable<K>, V> extends OrderedDictionary<K,V>
{
    // Returns the number of entries in the ordered dictionary
    // whose value is the specified value.
    int entriesWithValue( V value );
}

public interface OrderedDictionary<K extends Comparable<K>, V>
    extends Dictionary<K,V>
{
    Entry<K,V> minEntry( ) throws EmptyDictionaryException;
    Entry<K,V> maxEntry( ) throws EmptyDictionaryException;
}

public interface Dictionary<K,V>
{
    // Returns true iff the dictionary contains no entries.
    boolean isEmpty( );
    // Returns the number of entries in the dictionary.
    int size( );
    // Returns an iterator of the entries in the dictionary.
    Iterator<Entry<K,V>> iterator( );
    // If there is an entry in the dictionary whose key is the specified key,
    // returns its value; otherwise, returns null.
    V find( K key );
    // If there is an entry in the dictionary whose key is the specified key,
    // replaces its value by the specified value and returns the old value;
    // otherwise, inserts the entry (key, value) and returns null.
    V insert( K key, V value );
    // If there is an entry in the dictionary whose key is the specified key,
    // removes it from the dictionary and returns its value;
    // otherwise, returns null.
    V remove( K key );
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *OrderedDictionaryWithValueInfo*, descreva brevemente como implementaria as nove operações e calcule as suas complexidades temporais, no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo K, é sempre constante. Pode assumir que os valores também são comparáveis entre si (V extends Comparable<V>), com complexidade constante.

23. (Recurso 2010) Considere o tipo abstracto de dados *Pilha com Máximo*, de elementos do tipo E, caracterizado pela interface *StackWithMax*.

```
public interface StackWithMax<E extends Comparable<E>>
    extends Stack<E>
{
    // Returns the greatest element in the stack.
    E maximum( ) throws EmptyStackException;
}

public interface Stack<E>
{
    // Returns true iff the stack contains no elements.
    boolean isEmpty( );

    // Returns the number of elements in the stack.
    int size( );

    // Returns the element at the top of the stack.
    E top( ) throws EmptyStackException;

    // Inserts the specified element onto the top of the stack.
    void push( E element );

    // Removes and returns the element at the top of the stack.
    E pop( ) throws EmptyStackException;
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *StackWithMax*, descreva brevemente como implementaria as seis operações e calcule as suas complexidades temporais, no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo E, é sempre constante.

25. (Recurso 2010) Pretende-se construir uma aplicação para gerir um parque virtual de diversões. Para visitar o parque, é necessário realizar uma *inscrição* com um *endereço de e-mail*, que identificará sem ambiguidade o futuro visitante. Considera-se que um *visitante existe* quando o respectivo endereço foi dado numa inscrição.

No parque, os visitantes podem participar simultaneamente em diferentes *actividades* e podem, numa mesma *data* (dia, mês e ano), participar mais do que uma vez em cada actividade. A participação na actividade pode durar apenas alguns segundos ou vários dias. Cada *actividade* tem um *nome* (único, que a identifica), é de um determinado *tipo* e tem capacidade ilimitada.

A aplicação permite gerir as inscrições, as actividades e as participações dos visitantes nas actividades do parque virtual, através das seguintes operações.

(a) Inscrever um visitante no parque, dando o seu endereço de *e-mail*.
A operação só será efectuada se não existir um visitante com esse endereço.

(b) Adicionar uma actividade ao parque, dando o nome e o tipo da actividade.
A operação só será efectuada se não existir uma actividade com esse nome.

(c) Registar a entrada de um visitante no parque, dando o seu endereço.
A operação só será efectuada se o visitante existir e não estiver dentro do parque.

(d) Registar o início da participação (na data corrente) de um visitante numa actividade do parque, dando o endereço do visitante e o nome da actividade. Assuma que a data corrente será obtida chamando uma função da biblioteca.
A operação só será efectuada se o visitante estiver dentro do parque, mas não estiver a participar nessa actividade, e a actividade existir.

(e) Registar o fim da participação de um visitante numa actividade do parque, dando o endereço do visitante e o nome da actividade.
A operação só será efectuada se o visitante estiver a participar nessa actividade.

(f) Registar a saída de um visitante do parque, dando o seu endereço.
Esta operação só será efectuada se o visitante estiver dentro do parque e não estiver a participar em nenhuma actividade.

(g) Listar as participações (em actividades) de um dado visitante, dando o seu endereço.
Cada linha da listagem deve ter o *tipo* e o *nome* da actividade, a *data* de início e o *número de participações* do visitante nessa actividade que tiveram início nessa data.
A listagem deve estar ordenada alfabeticamente por tipo de actividade, dentro de cada tipo de actividade, por nome de actividade e, para cada actividade, decrescentemente por data.
A operação só será efectuada se o visitante existir.

(h) Listar os números totais de participações numa actividade, dando o nome da actividade.
Cada linha da listagem deve ter a *data* de início e o *número de participações* na actividade que tiveram início nessa data.
A listagem deve estar ordenada decrescentemente por data.
A operação só será efectuada se a actividade existir.

24. (Recurso 2009) Para simplificar este exercício, assuma que **todas as entradas de uma fila com prioridade têm chaves distintas**. Neste contexto, pode-se afirmar que o método *minEntry* da interface *MinPriorityQueue* retorna a entrada com a menor chave. Suponha que se acrescenta um novo método àquela interface (chamado *2ndMinEntry*), que devolve a entrada com a segunda menor chave.

Por exemplo, se a fila com prioridade tivesse as entradas (onde a chave é o primeiro elemento do par)

$$(3, \text{'z'}),\ (4, \text{'a'}),\ (7, \text{'b'})\ \text{ e }\ (10, \text{'j'}),$$

minEntry e *2ndMinEntry* retornariam $(3, \text{'z'})$ e $(4, \text{'a'})$, respectivamente.

```
public interface Entry<K,V>
{
    K getKey( );
    V getValue( );
}

public class MinHeap<K extends Comparable<K>, V>
    implements MinPriorityQueue<K,V>
{
    // Default capacity of the priority queue.
    public static final int DEFAULT_CAPACITY = 100;

    // The growth factor of the extendable array.
    public static final int GROWTH_FACTOR = 2;

    // Memory of the priority queue: an extendable array.
    protected Entry<K,V>[] array;

    // Number of entries in the priority queue.
    protected int currentSize;

    .....

    // Returns the entry with the second smallest key in the priority queue.
    public Entry<K,V> 2ndMinEntry( ) throws NoSuchEntryException;
}
```

Implemente o método *2ndMinEntry* na classe *MinHeap* e programe todos os métodos auxiliares desta classe de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo K, é sempre constante.

27. (Normal 2010) Pretende-se construir uma aplicação para gerir os atletas da União Europeia (UE) federados em modalidades olímpicas, durante um dado ano civil.

Cada *atleta* tem um *nome* único que o identifica, um *sexo* e uma *data de nascimento*, pratica uma só *modalidade* olímpica e pertence a uma *região* de um *país* da UE. A dimensão da região depende do país: tanto pode ser uma província, como na vizinha Espanha, como um distrito de Portugal.

Os nomes dos atletas, as modalidades, as regiões e os países são sequências de caracteres. Considere que as sequências que identificam as regiões e os países são todas distintas entre si (e nenhuma delas é "União Europeia").

A aplicação permite obter os números totais de atletas de cada modalidade, por sexo, de uma *zona* da UE. A zona será uma qualquer região, um qualquer país ou "União Europeia". Por exemplo, se a zona fosse "Portugal", como se ilustra na tabela, existiriam 5413 mulheres portuguesas federadas em atletismo, independentemente do distrito a que pertenciam, e 0 homens portugueses federados em ginástica artística.

modalidade	**F**	**M**
atletismo	5413	9560
badminton	0	0
basquetebol	64	271
.....	...	...
ginástica artística	1255	0
.....	...	...

Quando a aplicação é criada, indica-se o número de modalidades e as modalidades desse ano.

(a) Criar a aplicação, dando o número de modalidades e as modalidades.

Depois, podem-se efectuar as seguintes operações, por qualquer ordem.

(b) Inserir um atleta, dando o nome, o sexo, a data de nascimento, a modalidade, a região e o país.
A operação só não será efectuada se existir um atleta com o mesmo nome ou se não existir essa modalidade.

(c) Obter todos os dados de um atleta, dando o seu nome.

(d) Obter os nomes (ordenados alfabeticamente) de todos os atletas de uma região, dando a região.

(e) Listar os números totais de atletas de cada modalidade, por sexo, de uma zona da UE, dando a zona. A listagem deve estar ordenada alfabeticamente por modalidade e, para cada modalidade, primeiro o número de atletas do sexo feminino e, depois, o número de atletas do sexo masculino.
A listagem para o exemplo dado seria "5413, 9560, 0, 0, 64, 271, ..., 1255, 0, ...".

Como sabe, existem actualmente 27 países na UE. Considere que, no total, não deverão existir mais de 4 500 regiões, nem mais de 150 000 atletas federados na UE. Há cerca de meia centena de modalidades olímpicas (considerando os jogos de Verão e os de Inverno).

Assuma que há cerca de meio milhão de visitantes e cerca de 2 000 actividades. A probabilidade de um visitante participar em mais de cem actividades simultaneamente é inferior a 10^{-7}.

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar esta aplicação, descreva sumariamente os algoritmos para efectuar as oito operações (enumeradas de (a) a (h)) e calcule (justificando) as suas complexidades temporais, no caso esperado.

26. (Recurso 2009) Pretende-se criar um sistema para inventariar automóveis. Cada *automóvel* é identificado por uma *marca* e um *modelo*; tem uma *capacidade* do depósito de combustível, medida em litros; e possui uma *autonomia*, que é o número máximo de kilómetros que pode percorrer com um depósito cheio. Assuma que a capacidade e a autonomia são números inteiros.

    A *eficiência* de um automóvel, que indica o número de kilómetros percorridos por litro de combustível, é definida por $\frac{A}{C}$, onde A é a autonomia e C é a capacidade do automóvel. É óbvio que podem existir automóveis com a mesma eficiência, mas com autonomias e capacidades diferentes.

    O sistema deve permitir efectuar as seguintes operações.

    (a) Inserir um automóvel, dando a marca, o modelo, a capacidade e a autonomia.
    A operação não será efectuada se existir um automóvel com a mesma marca e o mesmo modelo.

    (b) Alterar as características de um automóvel, dando a marca e o modelo (que se mantêm) e os novos valores da capacidade e da autonomia.

    (c) Remover um automóvel, dando a marca e o modelo.

    (d) Listar, por ordem alfabética, todos os modelos dos automóveis com uma dada marca.

    (e) Listar, por uma ordem arbitrária, todos os automóveis que têm a máxima eficiência.
    Informação por automóvel: marca, modelo, capacidade e autonomia.

    Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar este sistema, descreva sumariamente os algoritmos para efectuar as cinco operações (enumeradas de (a) a (e)) e calcule (justificando) as suas complexidades temporais, no caso esperado.