Exame de Algoritmos e Estruturas de Dados

Departamento de Informática

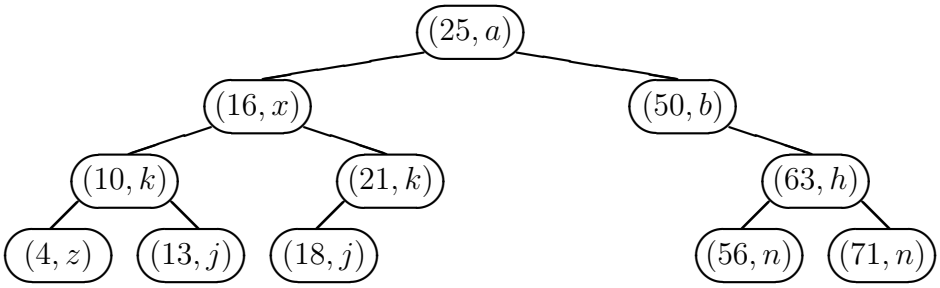
Universidade Nova de Lisboa

18 de Janeiro de 2010

1. [3.5 valores] 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 elements in the tree.
    protected int currentSize;

    . . . . . .

    public int pairsOfSiblingsWithTheSameValue( );
}
```

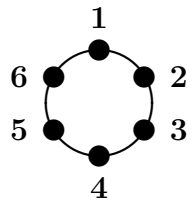
Implemente o método *pairsOfSiblingsWithTheSameValue* e programe todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do teste à igualdade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

2. [3.5 valores] 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( );

        // Returns the value in the dictionary whose key is the specified key;
        // or null if no such entry exists.
        V find( K key );

        // Inserts the entry (key, value) in the dictionary.
        // If the dictionary already contained an entry with the specified key,
        // returns the old value (which is replaced by the specified value);
        // otherwise, returns null.
        V insert( K key, V value );

        // Removes the entry whose key is the specified key from the dictionary
        // and returns the associated value, if such entry exists.
        // 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. Se quiser, pode assumir que os valores são comparáveis entre si (V extends Comparable<V>).

3. [3.5 valores] 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.

```
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
        return 2 * josephRec(n / 2) + 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.

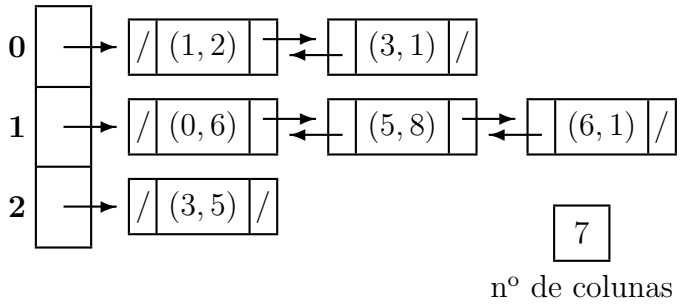
(Continue, porque o exame tem mais duas perguntas.)

4. [4.5 valores] 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.

5. [5 valores] 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).

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