Algoritmos e Estruturas de Dados
Departamento de Informática, Universidade Nova de Lisboa
Segundo teste - 10 de Dezembro de 2015

Atenção: Os Anexos ao teste poderão ser-lhe úteis.

1. Considere a árvore AVL apresentada na Figura 1. Em cada nó está apenas representada a chave do mesmo.

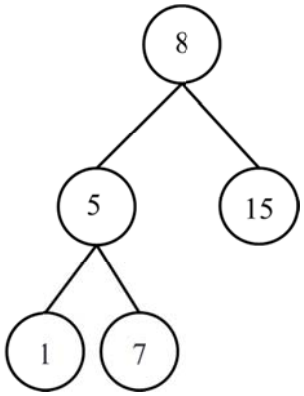

Figura 1

   a) Desenhe a árvore AVL resultante de inserir um elemento com a chave 3 na árvore apresentada;
   b) Desenhe a árvore AVL resultante de remover o elemento com a chave 5 da árvore apresentada na Figura 1. Atenção que a remoção deve ser feita na **árvore original,** representada na Figura.

2. Considere a tabela de dispersão aberta apresentada na Figura 2. Esta tabela constitui uma implementação do tipo abstrato de dados (TAD) Dicionário (Dictionary<K,V>), tal como apresentado nas aulas de AED. A tabela foi criada para conter no máximo 7 entradas (*maxSize*) e é constituída por um vetor de 7 posições (*arraySize*) onde, em cada posição, existe uma lista duplamente ligada, ordenada crescentemente pela chave das entradas, que guarda as colisões (entradas cujo resultado da aplicação da função de dispersão à chave da entrada é o mesmo). A função de dispersão da tabela apresentada é definida da seguinte forma:

```
// Returns the hash value of the specified key.
 protected int hash( int key )
 {
     return Math.abs( key ) % arraySize;
  }
```

   Neste momento, a tabela contém três entradas (*currentSize*). As entradas são do tipo Entry<Integer, Character>. A posição de cada entrada é encontrada pela aplicação da função de dispersão hash à chave da entrada. Assim, a entrada (1,'V') foi guardada na lista existente no índice 1 e a entrada (10,'3'), na lista pertencente ao índice 3.

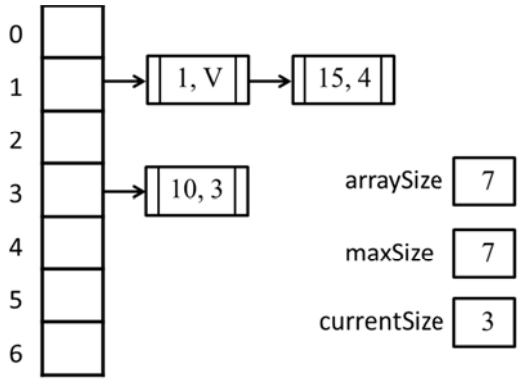

Figura 2

Dada a tabela de dispersão aberta apresentada na **Figura 2**, desenhe a tabela resultante de executar todas as operações pedidas abaixo (desenhe apenas a tabela final):

- Inserção da entrada (8, '7');
- Inserção da entrada (10,'7');
- Remoção da entrada com chave 4.

Com a tabela, apresente também os valores finais das variáveis de instância arraySize, maxSize e currentSize.

3. Considere uma implementação de uma Árvore Binária de Pesquisa que contém entradas do tipo Entry<K,K> sendo o tipo K comparável (`Comparable`). Considere um exemplo deste tipo de árvore na Figura 3, em que o tipo K considerado é `Integer`.

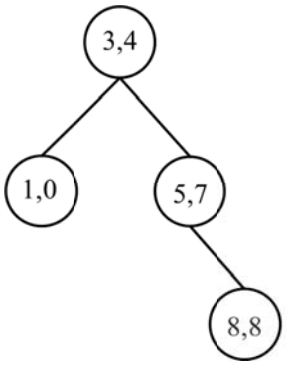

Figura 3

Considere que se pretende agora implementar um método **recursivo**, denominado `genBooleanTree()` que recebe, como parâmetro, a raiz de uma árvore deste tipo e que devolve a raiz de uma nova Arvore Binária de Pesquisa com a mesma forma, mas onde as entradas guardadas serão do tipo `Entry<K,Boolean>`. Nesta árvore, o valor `Boolean` de cada entrada será `true` caso, na árvore original, a chave da entrada seja de ordem menor que o valor da mesma entrada (ou seja, caso, para uma `Entry<K,K>` de nome `entry`, o resultado da chamada `entry.getKey().compareTo(entry.getValue())` seja menor que zero) e `false` no caso contrário.

No caso do exemplo da Figura 3, a árvore resultante da chamada ao método genBooleanTree() seria a árvore apresentada na Figura 4.

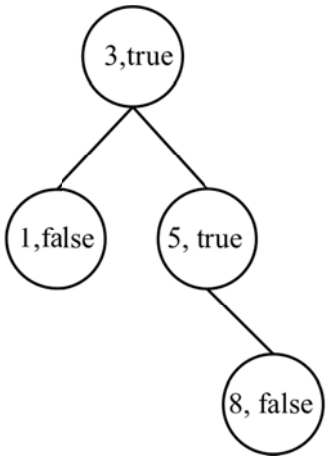

Figura 4

A assinatura do método **recursivo** a desenvolver será a apresentada abaixo:

```
private BSTNode<K,Boolean> genBooleanTree(BSTNode<K,K> node){

}
```

Calcule ainda a complexidade temporal do método `genBooleanTree()` no melhor caso, no pior caso e no caso esperado, justificando a sua resposta. Para esse efeito, pode assumir que a chamada ao método `compareTo()` associado ao tipo K é constante.

4. Pretendemos desenvolver um sistema de gestão de clientes e veículos associados a um sistema de pagamentos automatizados de portagens, com base num sensor que é colocado dentro do carro e que permite a identificação do veículo, ao passar na zona de portagem. A empresa ViaRápida mantém a seguinte informação sobre cada um dos seus *clientes*: número de identificação fiscal (NIF), nome completo, morada, número de telefone. Um cliente poderá ser um indivíduo ou uma empresa. Desta forma, um cliente poderá associar vários veículos ao seu registo na ViaRápida, não havendo limite para o número de veículos pertencentes a um mesmo cliente (na realidade poderão ser da ordem das centenas). Relativamente aos *veículos*, a informação guardada pelo sistema será: matrícula, marca, modelo e estado do sensor de identificação. Para cada veículo, deverá ainda ser possível saber qual o dono do mesmo.
   Importante: Não se pretende, para o sistema a desenvolver, guardar informação de registo de viagens, mas apenas conseguir identificar as relações entre os clientes e os veículos, assim como gerir o estado dos sensores de identificação.
   O sistema deverá permitir as seguintes operações:

   a) *Inserir cliente*, recebendo a seguinte informação: número de identificação fiscal (NIF), nome completo, morada e número de telefone. A inserção só tem sucesso se o NIF ainda não existir no sistema;
   b) *Inserir um novo veículo*, recebendo: matrícula, marca, modelo, estado do sensor de identificação (*em funcionamento* ou *avariado*) e NIF do dono do veículo. A Inserção só tem sucesso se o dono do veículo já existe no sistema e se a matrícula ainda não existe;
   c) *Remover veículo,* dando a matrícula. Para a operação ter sucesso, a matrícula tem de existir no sistema;

d) *Consultar dados de cliente,* dando o NIF. Esta operação só terá sucesso se o NIF existir no sistema e deve devolver os seguintes dados, associados ao cliente: nome completo, morada e número de telefone;
e) *Alterar estado de sensor de identificação do veículo*, dando a matrícula do veículo. O estado será alterado para o estado contrário aquele em que o sensor está no momento. Esta operação só terá sucesso se a matrícula do veículo existir no sistema;
f) *Consultar dados de Veículo,* dando a matrícula do veículo. Esta operação só tem sucesso se a matrícula existir no sistema e deverá devolver os seguintes dados: marca, modelo e estado do sensor de identificação;
g) *Listar veículos de um cliente,* dando o NIF do mesmo, ordenados alfabeticamente por matrícula. Esta operação só terá sucesso se o NIF existir no sistema e deverá listar, para cada veículo: a matrícula, a marca, o modelo e o estado do sensor de identificação.

Espera-se que o número de clientes e de veículos seja da ordem dos milhares. Tal como já mencionado, um cliente poderá ter centenas de veículos guardados no sistema. Com base nesta especificação:

- Explicite detalhadamente as estruturas de dados e as variáveis de instância mais adequadas para implementar esta aplicação;
- Descreva sumariamente os algoritmos para efetuar as sete operações (enumeradas de (a) a (g)) e calcule (justificando) as suas complexidades temporais, no caso esperado.

Anexo A – Interfaces e Classes de Apoio

```
public interface Comparable<T>{
   int compareTo( T object );
}
public interface Stack<E>{
   boolean isEmpty( );
   int size( );
   E top( ) throws EmptyStackException;
   void push( E element );
   E pop( ) throws EmptyStackException;
}
public interface Queue<E> {
   boolean isEmpty( );
   int size( );
   void enqueue( E element );
   E dequeue( ) throws EmptyQueueException;
}
public interface Iterator<E> {
    boolean hasNext( );
    E next( ) throws NoSuchElementException;
    void rewind( );
}

public interface List<E> {
    boolean isEmpty( );
    int size( );
   //( List<E> continua na página 6)
```

```
    //continuação do interface List<E> (página 5).

    Iterator<E> iterator( );
    E getFirst( ) throws EmptyListException;
    E getLast( ) throws EmptyListException;
    E get( int position ) throws InvalidPositionException;
    int find( E element );
    void addFirst( E element );
    void addLast( E element );
    void add( int position, E element )
       throws InvalidPositionException;
    E removeFirst( ) throws EmptyListException;
    E removeLast( ) throws EmptyListException;
    E remove( int position ) throws InvalidPositionException;
    boolean remove( E element );
}
public interface Entry<K,V>{
   K getKey( );
   V getValue( );
}
public interface Comparable<T>{
   int compareTo( T object );
}
public interface Dictionary<K,V>{
   boolean isEmpty( );
   int size( );
   Iterator<Entry<K,V>> iterator( );
   V find( K key );
   //interface Dictionary<K,V> continua na página 7.
```

```
    //continuação do interface Dictionary<K,V> (página 6).
    V insert( K key, V value );
    V remove( K key );
}
public interface OrderedDictionary<K extends Comparable<K>, V>
        extends Dictionary<K,V>{
   Entry<K,V> minEntry( ) throws EmptyDictionaryException;
   Entry<K,V> maxEntry( ) throws EmptyDictionaryException;
}
class BSTNode<K,V>{
   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( );
}
```

Anexo B - Recorrências

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(n-1) + c & n \geq 1 \end{cases} \qquad \begin{matrix} \\ \\ n = 1 \\ \\ n \geq 2 \end{matrix}$$

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

com $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

Recorrência 2a)

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(\frac{n}{2}) + O(1) & n \geq 1 \end{cases} \qquad \begin{matrix} \\ \\ n = 1 \\ \\ n \geq 2 \end{matrix}$$

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

com $a \geq 0, \quad b = 1, 2$ **constantes**

Recorrência 2b)

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ bT(\frac{n}{c}) + O(n) & n \geq 1 \end{cases} \qquad \begin{matrix} \\ \\ n = 1 \\ \\ n \geq 2 \end{matrix}$$

com $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$