Algoritmos e Estruturas de Dados
Segundo Teste – 2013/2014
Departamento de Informática, Universidade Nova de Lisboa
12 de Dezembro de 2013

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.

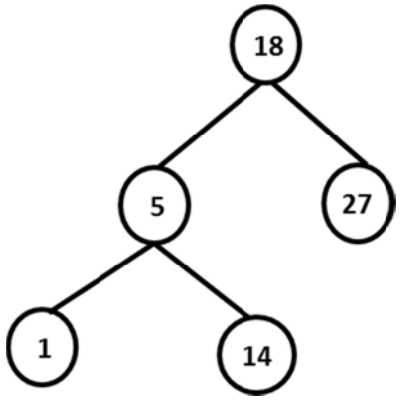

Figura 1

   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 os Tipos Abstractos de Dados representados pelos interfaces abaixo. O interface Person permite a consulta de informação relativa a uma pessoa (primeiro nome, nome de família e ano de nascimento). O interface Family permite a consulta de dados relativos a uma família. Cada elemento da família é também representado através de uma implementação do interface Person.

```
public interface Person {

   //Devolve o primeiro nome da pessoa.
   String getFirstName();

   //Devolve o nome de família da pessoa.
   String getLastName();

   //Devolve o ano de nascimento da pessoa.
   int getYearOfBirth();

}
```

(Continua na página 2)

```
public interface Family {

   //Devolve o pai da família.
   Person getFather();

   //Devolve a mãe da família.
   Person getMother();

   //devolve o numero de filhos da família.
   int numberOfChildren();

   //devolve um iterador dos filhos da família.
   Iterator<Person> childrenIterator();
}
```

Dados estes interfaces, implemente o iterador KidsIterator, um iterador de objetos do tipo Person que recebe, no construtor, um iterador de objetos do tipo Family, e que implementa a iteração de todos os filhos associados às famílias contidas no iterador original. A ordem de iteração dos filhos deve ser a ordem das famílias contidas no iterador de famílias.

Defina as variáveis de instância do iterador e implemente todos métodos auxiliares de que necessitar para completar a classe. Para cada método apresentado calcule (justificando) as suas complexidades temporais, no caso esperado. Para este efeito pode considerar que os filhos de uma família estão guardados em Listas duplamente ligadas (DoublyLinkedList<Person>).

```
public class KidsIterator implements Iterator<Person> {

   //Cria um iterador dos filhos de todas as famílias contidas no
   //iterador it.
   public KidsIterator(Iterator<Family> it);

   //Inicializa a iteração.
   public void rewind();

   //Devolve true se a iteração ainda não tiver chegado ao fim.
   public boolean hasNext();

   //Devolve o próximo filho na iteração, se esta não tiver chegado ao fim.
   //Requires: hasNext()
   public Person next() throws NoSuchElementException;
}
```

(Continua na página 3)

3. Pretendemos desenvolver um Sistema de gestão de um clube de vídeo online que disponibiliza filmes pelo método de streaming. O sistema deverá permitir a gestão dos filmes em aluguer no clube, assim como dos clientes inscritos no clube.
   Cada filme é identificado de forma única por um código de registo, e deverá conter também um título, uma duração (em minutos) e um género, assim como o URL de localização do filme. Assume-se que o título de cada de filme é único, condição que não precisa de ser verificada. Um filme está inicialmente em estado ativo e poderá ser desativado se deixar de estar disponível para aluguer.
   Um cliente do clube será identificado univocamente pelo seu endereço de e-mail, e deverá disponibilizar a seguinte informação: nome, morada e telefone. Deverá ainda ser possível listar todos os filmes já alugados por cada cliente.
   O sistema deverá permitir as seguintes operações:

   a) *Inserir um filme*, recebendo a seguinte informação: código de registo, título, duração em minutos, género, e URL de localização do filme. A inserção só tem sucesso se o código de registo do filme ainda não existir no sistema. O filme é inserido com o estado de ativo;
   b) *Inserir um Cliente*, recebendo: endereço de e-mail, nome, morada e telefone. A Inserção só tem sucesso se o endereço de e-mail não existir no sistema;
   c) *Consultar Cliente*, dando o endereço de e-mail. A informação a consultar será nome, morada e telefone do cliente, assim, como todos os títulos dos filmes já alugados pelo mesmo, ordenados alfabeticamente por título. A operação só tem sucesso se o endereço de e-mail do cliente existir no sistema;
   d) *Desativar filme,* dando o código de registo. Esta operação só tem sucesso se o código de registo existir no sistema. Depois de desativado, o filme não pode ser alugado;
   e) *Alugar um filme,* dando o código de registo do filme e o e-mail do cliente. Para a operação ter sucesso, o código de registo e o e-mail têm de existir no sistema e o filme tem de estar ativo. Cada cliente só pode ser alugar um determinado filme uma vez.

   Espera-se que o número de filmes seja da ordem dos milhares. O número de clientes será da ordem das centenas de milhares.

   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 cinco operações (enumeradas de (a) a (e)) e calcule (justificando) as suas complexidades temporais, no caso esperado.

   Não deve desenvolver código em java, apenas fazer uma descrição da implementação das operações de acordo com a sua escolha de estruturas de dados.

(Anexo na página 4)

Anexo – Interfaces e Classes de Apoio

```
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( );
    Iterator<E> iterator( );
    E getFirst( ) throws EmptyListException;
    E getLast( ) throws EmptyListException;
    E get( int position ) throws InvalidPositionException;
    //interface List<E> continua na página 5.
```

```
    //continuação do interface List<E> (página 4).
    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 );
   V insert( K key, V value );
   V remove( K key );
}
```

(continua na página 6)

```
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( );
}
```