Algoritmos e Estruturas de Dados
Exame de Recurso
Departamento de Informática, Universidade Nova de Lisboa
9 de Janeiro de 2017

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

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

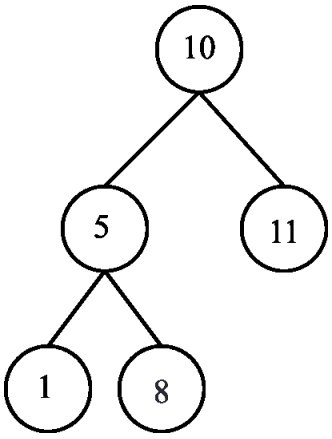

Figura 1

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

2. O Algoritmo de ordenação Bucket Sort, apresentado nas aulas teóricas de AED, faz parte de um conjunto de algoritmos de ordenação que, suportados por técnicas de indexação, permitem ordenar um conjunto de elementos em tempo linear. Neste tipo de algoritmos, as chaves dos elementos a ordenar devem fazer parte de um conjunto limitado, de pequena dimensão. Desta forma, pedimos-lhe que implemente o algoritmo Bucket Sort para um conjunto de entradas do tipo `Entry<Integer,V>` com chaves entre **0** e ***n***. Este algoritmo recebe um iterador de entradas deste tipo (desordenadas) e devolve também um iterador de entradas, já ordenadas por ordem da chave. As chaves não são únicas e a permutação das entradas, gerada pelo algoritmo, deve ser **estável**. Para executar a ordenação, o algoritmo usa um vetor de ***n+1*** posições, de listas de entradas, que irá receber, na posição i, todas as entradas com chave i, tal como apresentado na Figura 2.

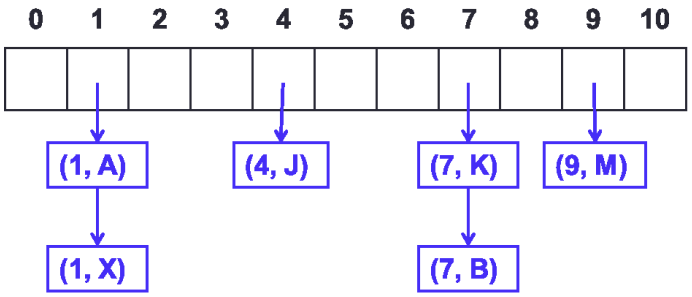

Figura 2

No exemplo apresentado na figura, o resultado do método será um iterador de uma lista entradas que conterá as mesmas pela seguinte ordem: **(1, A) (1, X) (4, J) (7,K) (7,B) (9, M)**. Considere que o iterador passado como parâmetro no método está implementado sobre uma lista duplamente ligada, assim como o iterador que é devolvido como resultado, no final da execução do método.

Importante: Para a implementação do método ser considerada para avaliação, o algoritmo de ordenação deve obrigatoriamente utilizar o vetor de listas apresentado na figura 2, contribuindo esta estrutura de dados para uma complexidade temporal linear.

A assinatura do método a implementar é apresentada a seguir:

```
Iterator<Entry<Integer,V>> bucketSort( Iterator<Entry<Integer,V>> entries,
                                       int numberOfKeys ) {

   //Implementação do algoritmo de ordenação Bucket Sort.
}
```

3. Considere uma Arvore Binária de Pesquisa que contém, no valor associado à entrada de um nó, a altura do nó em questão. Nesta árvore, as entradas `Entry<K,V>` guardadas no nó têm as seguintes características: o tipo `K` é comparável e o tipo `V`, que contém a altura do nó, um inteiro. Implemente um método booleano **recursivo** que recebe uma árvore deste tipo (a sua raiz) e determina se a árvore em questão é ou não uma árvore equilibrada. Deve relembrar a definição dada nas aulas teóricas de árvore equilibrada. Para o ajudar podemos dizer-lhe que uma árvore AVL é uma árvore binária de pesquisa equilibrada. A assinatura do método a desenvolver (nomeado `isBalanced()`) é apresentada abaixo. Implemente todos os métodos auxiliares de que necessitar. Calcule ainda a complexidade temporal do método no pior caso, justificando.

```
boolean isBalanced( BSTNode<K,Integer> treeRoot ){

   //True se a árvore (nó) treeRoot for equilibrada.
 }
```

4. Pretendemos implementar um sistema de gestão de textos de grande dimensão. Um texto é constituído por várias linhas e cada linha por várias palavras. Não estão definidos máximos para o número de linhas de cada texto, nem para o número de palavras existentes em cada linha. Cada linha tem um número de ordem atribuído e pode ocorrer repetição de palavras ao longo do texto. Abaixo pode consultar a interface Java definida para o tipo abstrato de dados (TAD) `LineText`. Este contém operações que permitem adicionar linhas ao texto (sempre no final do mesmo), saber o número de ocorrências de cada palavra existente no texto, consultar os números das linhas onde uma determinada

palavra ocorre e consultar uma determinada linha do texto, inserindo o seu número de ordem.

```
public interface LineText {

   //Adiciona linha de texto no final do mesmo
   void addLine( String line );

   //Devolve o número de ocorrências atual de uma palavra no texto
   int numberOfOccurrences( String word ) throws NonExistingWord;

   //Devolve um iterador dos números de ordem das linhas onde a palavra
   //word é mencionada no texto. Este iterador deve iterar os números de
   //linha crescentemente.
   Iterator<Integer> whereIsWord( String word ) throws NonExistingWord;

   //Devolve a linha com o número de ordem lineNumber
   String showLine( int lineNumber ) throws NonExistingLines;

}
```

Relativamente ao problema proposto:

- Explicite detalhadamente as estruturas de dados e as variáveis de instância mais adequadas para implementar a interface dada. Se a sua tarefa for facilitada pela criação de TADs auxiliares à sua solução, deverá também descrever as variáveis de instância e estruturas de dados associadas a estes na sua resposta;
- Descreva brevemente como implementaria todas as operações e calcule as suas complexidades temporais, no caso esperado, justificando. **Não deve desenvolver código em java, apenas fazer uma descrição da implementação das operações de acordo com a sua escolha de estruturas de dados e variáveis de instância.**

5. Uma empresa de formação pretende desenvolver um sistema de disponibilização de cursos online, iniciando a implementação com o sub-sistema de gestão de subscrição de cursos e especializações. Os *Cursos* estão organizados em *Especializações* que podem ser subscritas por *Entidades* oficiais, registadas no site da empresa. Por sua vez os colaboradores destas entidades podem frequentar os cursos associados às especializações subscritas. Não vamos, no âmbito deste exercício, debruçarmo-nos sobre os colaboradores, mas apenas sobre as entidades. As especializações são conjuntos de cursos com um tema em comum e podem ser alteradas frequentemente, com inserções e remoções dos cursos. Um curso só poderá, no máximo, fazer parte de uma especialização.

   O sistema deverá permitir as seguintes operações:

   a) *Registar Entidade*, recebendo a seguinte informação: número de identificação fiscal (NIPC), nome, morada, telefone e email de contacto. A inserção só tem sucesso se o NIPC ainda não existir no sistema;
   b) *Remover Entidade*, dando o NIPC. Esta operação só terá sucesso se uma entidade com este NIPC já existir no sistema;

c) *Adicionar curso,* recebendo a seguinte informação: código (único) do curso e nome. Esta operação só terá sucesso se o código do curso ainda não existir no sistema. O curso, no momento da sua criação, não está associado a especializações;
d) *Criar Especialização,* recebendo a seguinte informação: identificador único da especialização, Título, e códigos de todos os cursos que serão associados à especialização. Esta operação só terá sucesso se não existir ainda nenhuma especialização com o identificador dado. A possibilidade dos cursos fazerem parte da especialização deve também ser confirmada e no caso negativo a operação não tem sucesso;
e) *Subscrever/Remover subscrição de especialização,* dando o identificador da entidade e o identificador da especialização. Esta operação permite a uma entidade subscrever todos os cursos que façam parte de uma especialização e desistir da mesma subscrição. A operação só terá sucesso se tanto o identificador da entidade como o identificador da especialização já existirem no sistema;
f) *Consultar entidade,* dando o NIPC. Esta operação só terá sucesso se o NIPC já existir no sistema e deverá devolver o nome da entidade, morada, telefone e email, assim como uma listagem do código e nome de todos os cursos por ela subscritos. Esta listagem deverá estar ordenada por identificador de especialização e, para todos os cursos da mesma especialização, por código do curso;
g) *Listar todos os cursos de uma especialização,* dando o identificador da especialização. A operação só terá sucesso se o identificador da especialização já existir no sistema. Esta listagem deverá ser ordenada por código de curso.

Espera-se que o número de entidades e cursos seja da ordem dos milhares. As especializações poderão ser da ordem das centenas e poderão conter centenas de cursos. Com base nesta especificação:

- Explicite detalhadamente as estruturas de dados e as variáveis de instância mais adequadas para implementar esta aplicação; Se a sua tarefa for facilitada pela criação de TADs auxiliares à sua solução, deverá também descrever as variáveis de instância e estruturas de dados associadas a estes na sua resposta.
- Descreva sumariamente os algoritmos para efetuar as 7 operações (enumeradas de (a) a (g)) 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 A - 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**

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

Anexo B – 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 7)
```

```
    //continuação do interface List<E> (página 6).

    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 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 );
}
```

```
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 DListNode<E> implements Serializable {
    public DListNode( E theElement, DListNode<E> thePrevious,
       DListNode<E> theNext );

    public DListNode( E theElement );

    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> {

    public boolean isEmpty( );

    public int size( );

    public Iterator<E> iterator( );

    public E getFirst( ) throws EmptyListException;

    public E getLast( ) throws EmptyListException;

    protected DListNode<E> getNode( int position );

    //( DoublyLinkedList<E> continua na página 9)
```

```
   //continuação de DoublyLinkedList<E> (página 8).

   public E get( int position ) throws InvalidPositionException;

   public int find( E element );

   public void addFirst( E element );

   public void addLast( E element );

   protected void addMiddle( int position, E element );

   public void add( int position, E element )
      throws InvalidPositionException;

   protected void removeFirstNode( );

   public E removeFirst( ) throws EmptyListException;

   protected void removeLastNode( );

   public E removeLast( ) throws EmptyListException;

   protected void removeMiddleNode( DListNode<E> node );

   public E remove( int position )
      throws InvalidPositionException;

   protected DListNode<E> findNode( E element );

   public boolean remove( E element );

   public void append( DoublyLinkedList<E> list )

}

class BSTNode<K,V>{

   public BSTNode( K key, V value, BSTNode<K,V> left,

                   BSTNode<K,V> right );

   public BSTNode( K key, V value );

//( BSTNode<K,V> continua na página 10)
```

```
    //continuação de BSTNode<K,V> (página 9).

    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( );
}
```