Algoritmos e Estruturas de Dados
Exame de Recurso
Departamento de Informática, Universidade Nova de Lisboa
8 de Janeiro de 2018

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

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

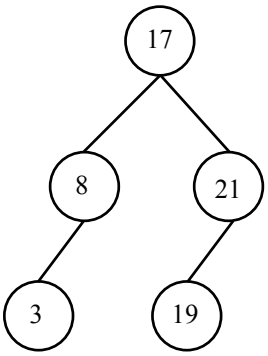

Figura 1

i) Desenhe a árvore AVL resultante de remover o elemento com a chave 17 da árvore apresentada;

ii) Desenhe a árvore AVL resultante de inserir um elemento com chave 1 na árvore apresentada na Figura 1. **Atenção** que a inserção deve ser feita na **árvore original,** representada na Figura.

1. b) 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 quatro 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 (6,'6'), na lista pertencente ao índice 3.

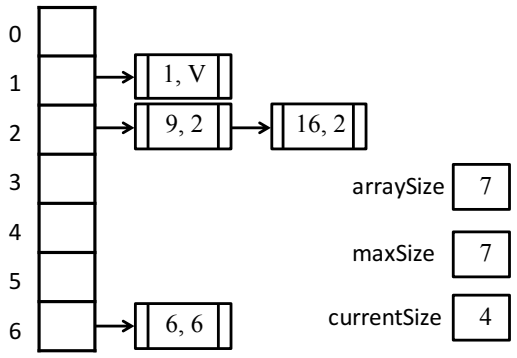

Figura 2

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

- Inserção da entrada (6,'7');
- Inserção da entrada (4,'V');
- Remoção da entrada com chave 2.

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

2. Considere a Classe `BinarySearchTree` apresentada nas aulas (ver anexo). Pedimos-lhe que desenvolva um construtor **recursivo** para esta classe, que cria uma nova árvore a partir de uma árvore existente. A nova árvore deverá ser em tudo igual à árvore recebida como parâmetro, em forma e conteúdo. No entanto, a nova árvore não deverá partilhar nós com a árvore original. Implemente todos os métodos auxiliares de que necessite.

   Calcule ainda a complexidade temporal do mesmo método, no melhor caso, pior caso e no caso esperado, justificando. Apenas as versões recursivas do construtor serão avaliadas.

```
public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V> {

  // a raiz da árvore.
  protected BSTNodeB<K,V> root;
      ...

  // Construtor que cria uma nova árvore igual à
  // árvore otherTree
 public BinarySearchTree ( BinarySearchTree <K,V> otherTree ){
      ...
  }

      ...
}
```

(O exame continua na página 3)

3. Considere uma implementação de um dicionário ordenado (`OrderedDictionary<K,V>`) denominado `OrderedDicInList<K,V>` sendo que tanto `K` como `V` implementam o interface `Comparable`. Pedimos-lhe que desenvolva um método protegido de verificação dos dados contidos no dicionário. O objetivo é identificar se existem mais do que uma entrada com o mesmo valor (tipo `V`) no dicionário. Este método (`repeatedValues`) deverá devolver todas as chaves (com exceção da primeira) que estiverem associadas ao mesmo valor (`value`), passado como parâmetro no método (ver assinatura do método abaixo). Se o dicionário estiver vazio, ou se o valor não existir no mesmo, o método deverá devolver `null`. Caso contrário, o método devolverá um iterador de todas as chaves encontradas, com exceção da primeira, pela ordem em que foram sendo encontradas. O iterador com os resultados do método deverá ser implementado sobre uma lista duplamente ligada (`DoublyLinkedList<K>`). Note que esta lista poderá ser vazia, se o valor procurado existir apenas uma vez no dicionário (não havendo repetições do valor).

   Abaixo apresentamos-lhe a assinatura do método a implementar. Desenvolva todos os métodos auxiliares de que necessite. Calcule ainda a complexidade temporal do método `repeatedValues`, no melhor caso, no pior caso e no caso esperado, justificando. Para este efeito poderá assumir que o dicionário ordenado está implementado em lista duplamente ligada.

```
//Devolve as chaves das entradas do dicionário ordenado cujos valores são
 //iguais a value
 protected Iterator<K> repeatedValues(V value){
   ...
 }
```

4. Um histograma é um gráfico de frequências de valores associados a uma determinada variável. Pretendemos disponibilizar uma implementação do Tipo Abstrato de Dados `GradeHistogram` (apresentado abaixo) que irá guardar os dados necessários para a criação de um histograma de notas de estudantes de uma determinada unidade curricular (UC).

   As operações do TAD incluem: *inserir notas* de alunos (uma nota, no máximo, para cada estudante), *consultar a nota* de um estudante, *consultar a frequência* de uma nota (quantos alunos obtiveram a mesma), *consultar a Moda dos dados* (qual a nota com a maior frequência) e *listar os dados do histograma* (nota e frequência) por ordem crescente da nota.

   O estudante é representado por um identificador único. Os dados do histograma serão apresentados através de entradas `Entry<Integer,Integer>` que conterão para cada nota, o número de estudantes que obtiveram a mesma nota (frequência).

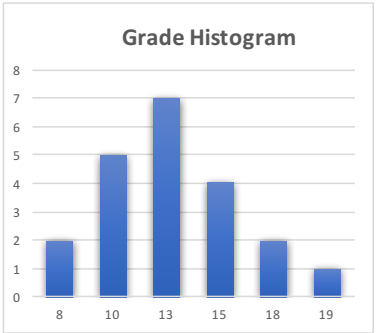

Figura 3

Na Figura 3, apresenta-se um exemplo de um histograma de notas. Os dados deste histograma seriam devolvidos pelo método `histogramData()` pela seguinte ordem: (8, 2) (10, 5) (13, 7) (15, 4) (18, 2) (19, 1).

```
public interface GradeHistogram {

   //Adiciona nota de estudante
   void addGrade( String idStudent, int grade )
     throws StudentAlreadyHasGrade, NonValidGrade;

   //Devolve a nota de um estudante
   int getStudentGrade( String idStudent ) throws NonExistingStudent;

   //Devolve a frequência de uma nota
   int getGradeFrequency( int grade ) throws NonExistingGrade;

   //Devolve iterador das frequências de cada nota atribuída
   //aos estudantes. A iteração é executada por ordem crescente
   //das notas. Cada iteração devolve uma entrada composta pela nota e
   //pela frequência da mesma no conjunto dos estudantes
   Iterator<Entry<Integer, Integer>> histogramData( );

    //Devolve a Entrada que contém a Moda do conjunto de dados (a nota com
    //a maior frequência, e a respetiva frequência)
   Entry<Integer,Integer> getMode( ) throws EmptyDataSet;
}
```

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 do TAD apresentado 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.**

(O exame continua na página 5)

5. Nesta pergunta pedimos-lhe considere o desenvolvimento de um sistema para classificação de ficheiros por palavras-chave (tags). Um *ficheiro* é adicionado ao sistema acompanhado de alguma (meta) informação. Num momento posterior, um ficheiro sem classificação pode ser classificado através de uma lista (pequena) de tags. Após esta operação, numa pesquisa posterior, este ficheiro será sempre apresentado a um utilizador interessado numa das tags em questão. A pesquisa por uma tag no sistema deverá devolver a lista de ficheiros classificados com a mesma tag, ordenados por **nome** de ficheiro.
   A classificação de um ficheiro pode ser alterada mais tarde, apenas através da remoção de uma tag da sua lista de classificação. A remoção completa de uma tag do sistema também é possível. Se ao fim de várias alterações no sistema, um ficheiro tiver uma lista de classificação vazia, este poderá receber uma nova lista de tags.

   Assim, o sistema deverá permitir as seguintes operações:
   a) *Inserir ficheiro*, recebendo a seguinte informação: identificador único do ficheiro, nome, URL e dimensão em Mbytes. A inserção só tem sucesso se o identificador do ficheiro ainda não existir no sistema;
   b) *Classificar ficheiro,* recebendo o identificador do ficheiro e uma lista de palavras-chave (tags). Esta operação só terá sucesso se o identificador do ficheiro já existir no sistema, se a lista de tags do ficheiro identificado estiver vazia e se a lista de tags passada na operação não for vazia;
   c) *Consultar dados de ficheiro,* recebendo o identificador do ficheiro. Esta operação só terá sucesso se o identificador do ficheiro existir no sistema e deverá apresentar nome, URL e dimensão do ficheiro, assim como a lista das tags que classificam o ficheiro, sem ordem pré-definida;
   d) *Remover tag da classificação de um ficheiro*, dando o identificador do ficheiro e a tag a remover da lista de classificação do ficheiro. Esta operação só terá sucesso se o identificador do ficheiro já existir no sistema e se a tag fizer parte da lista de classificação do ficheiro em questão;
   e) *Remover tag do sistema*, recebendo a tag. Esta operação só terá sucesso se esta tag existir no sistema;
   f) *Listar os dados de todos os ficheiros classificados com uma tag,* dando a tag. A operação só terá sucesso se a tag existir no sistema. Esta listagem deverá ser ordenada por nome do ficheiro e os dados a apresentar por ficheiro serão nome, URL e dimensão.

   O sistema deverá ser pensado tendo em conta que podem existir milhares de ficheiros, cada um classificado com uma lista pequena de tags. O número de tags totais existentes no sistema será também da ordem dos milhares. 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 6 operações (enumeradas de (a) a (f)) 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( );
    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 );
    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 );
     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( );
}
```