Algoritmos e Estruturas de Dados
Exame de Recurso
Departamento de Informática, Universidade Nova de Lisboa
11 de Janeiro de 2014

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

1. Considere a tabela de dispersão aberta apresentada na Figura 1. 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, as entradas (1,T) e (15,A) foram guardadas na lista existente no índice 1 da tabela e a entrada (11,X), na lista pertencente ao índice 4.

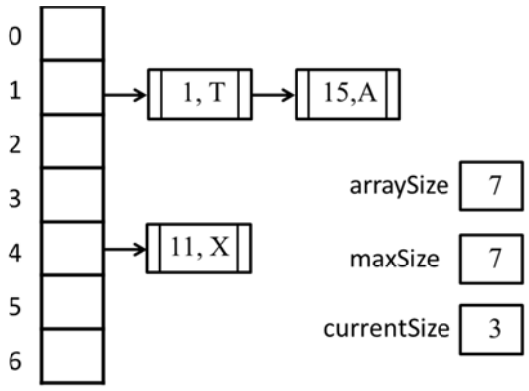

Figura 1

Dada a tabela de dispersão aberta apresentada na **Figura 1**, desenhe a tabela resultante de executar as operações pedidas abaixo:

- Remoção da entrada (4,X);
- Inserção da entrada (8,A);
- Inserção da entrada (11,B).

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

2. Considere o conceito de profundidade de uma árvore, tal como apresentado nas aulas teóricas de AED. A **profundidade** de uma árvore não vazia é o máximo das profundidades das suas folhas, sendo que a **profundidade de um nó** de uma árvore é o número de nós que compõem o (único) caminho (sempre descendente) da raiz ao nó. Em particular, a profundidade da raiz de uma árvore é 1. Considere o exemplo da Figura 2: A profundidade da árvore é 3; a profundidade do nó com chave 82 é 2 e a profundidade do nó com chave 30 é 3.

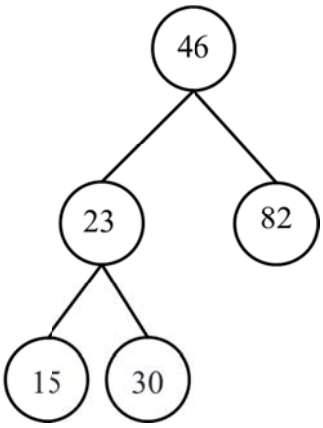

Figura 2

Considere a Classe BinarySearchTree apresentada nas aulas (ver anexo). Pedimos-lhe que desenvolva um método **recursivo** que devolva a profundidade da árvore. Implemente todos os métodos auxiliares de que necessite. Calcule ainda a complexidade temporal do mesmo método, no caso esperado, justificando.

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

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

   // Método público que devolve a profundidade da árvore.
   public int depth(){

      ...
   }
```

(O exame continua na página 3)

3. Considere a Classe `DoublyLinkedList<E>` (ver anexo) apresentada nas aulas teóricas e assuma, para este exercício, que o genérico `E` estende o interface `Comparable<E>`. Tendo isto em consideração, implemente o método `removeNotOrdered()` que remove da lista todos os nós cujos elementos fazem com que a mesma não seja uma lista ordenada crescentemente. Como exemplo, considere a lista L que contém 6 elementos inteiros, pela seguinte ordem: 2, 6, 6, 5, 7, 1. O resultado de aplicar o método `removeNotOrdered()` à mesma lista fará com que os elementos 5 e 1 sejam removidos e, após a execução do método, a lista conterá apenas 4 elementos pela seguinte ordem: 2, 6, 6, 7. Após a execução do método, a lista é uma lista ordenada.

   Deve desenvolver todos os métodos necessários para a completa implementação de `removeNotOrdered()` e calcule a sua complexidade temporal **no melhor caso, no pior caso e no caso esperado**, justificando.

```
public class DoublyLinkedList<E extends Comparable<E>>
   implements List<E> {

   // Node at the head of the list.
   protected DListNode<E> head;

   // Node at the tail of the list.
   protected DListNode<E> tail;

   // Number of elements in the list.
   protected int currentSize;

   ...

   // remove unordered elements from list.
   public void removeNotOrdered(){
   ...
   }

}
```

4. Nos sistemas de gestão de conteúdos, a classificação dos mesmos é de grande importância, dado que facilita a pesquisa e o acesso a dados relevantes. Vimos assim pedir-lhe que desenvolva uma implementação para o Tipo Abstrato de Dados (TAD) `TaggedPhotos` que permite fazer a gestão da classificação de uma base de fotografias. A informação relevante para cada fotografia será apenas o seu nome (poderá ser um título único que descreva a fotografia) e uma lista de palavras-chave (*tags*) que descrevem o seu conteúdo. O comportamento associado ao TAD, além do armazenamento dos dados associados às fotografias, incluirá avaliar o peso de uma determinada tag na base de fotografias. No exemplo da tabela 1, existem 3 fotos e 5 tags (“lisboa”, “cidade”, “rio”, “ponte” e “barragem”). O número total de menções a tags é de 8, sendo que a tag com maior peso na base é “rio” (com três menções). A implementação do TAD deverá permitir saber qual o peso (rácio) que uma determinada tag tem no total de menções de tags existentes na base. No caso da tag “lisboa” do exemplo, o seu peso na base é de 0.25 (2 menções da tag num total de 8 menções de tags). **(cont.)**

Tabela 1

Nome da Foto	Tags
Lisboa à noite	lisboa, cidade, rio
Ponte 25 de Abril	lisboa, ponte, rio
Barragem do Alqueva	rio, barragem

```
public interface TaggedPhotos {

   //Requires: name não existe no sistema.
   //Insere a foto no sistema e associa-lhe a lista de tags.
   void insertPhoto(String name, List<String> tags)
      throws ExistingPhotoName;

   //Remove a foto do sistema e atualiza as menções das tags que lhe
   //estão associadas. Devolve true se a foto foi removida e false se a
   // foto não existia no sistema.
   boolean removePhoto(String name);

   //Devolve o número total de tags existentes no sistema.
   int getTotalTags();

   //Devolve o número total de menções a tags existentes no sistema.
   int getTotalTagMentions();

   //Devolve o número de menções existentes no sistema, para a tag taag.
   int getTagMentions(String taag);

   //Devolve o peso da tag taag no sistema.
   double getTagWeight(String taag);
}
```

Baseando-se na descrição apresentada e no interface fornecido, explicite detalhadamente as estruturas de dados mais adequadas e as variáveis de instância mais adequadas para implementar o mesmo, descreva brevemente como implementaria as seis 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.**

(O exame continua na página 5)

5. Considere o desenvolvimento de um sistema de gestão de equipas de trabalho numa empresa de grande dimensão. O sistema mantém informação sobre os funcionários da empresa e sobre as equipas que os integram. Neste contexto, será possível inserir e remover funcionários, e criar novas equipas, assim como cancelá-las. Deverá também ser possível alterar a composição das equipas, adicionando e removendo membros às mesmas. Cada funcionário deverá integrar, no máximo, uma equipa. A informação associada ao funcionário é a seguinte: código de funcionário (único na empresa), nome, categoria e equipa a que pertence. No ato de criação do funcionário, este não deve ainda pertencer a nenhuma equipa. Relativamente à equipa, a informação que lhe é associada é um nome (único) e o conjunto de funcionários que a compõem. **Nota:** deverá ser possível listar a composição de uma equipa por ordem alfabética do nome dos seus membros. Pode assumir, sem necessidade de validação, que os nomes dos funcionários são únicos.

   As operações suportadas pelo sistema a desenvolver são as seguintes:

   a) *Inserir um Funcionário*, dando o seu código, nome e categoria. A operação só tem sucesso se o código de funcionário ainda não existir no sistema;
   b) *Remover Funcionário*, dando o seu código. A operação só tem sucesso se o código de funcionário já existir no sistema;
   c) *Consultar dados de Funcionário*, dando o seu código. Esta operação só tem sucesso se o código de funcionário já existir no sistema e deverá devolver todos os dados do funcionário, incluindo o nome da equipa a que pertence, caso o funcionário esteja integrado numa equipa;
   d) *Criar Equipa*, dando o seu nome. A equipa é criada sem membros mas a operação só terá sucesso se o nome da equipa ainda não existir no sistema.
   e) *Alterar equipa,* dando o seu nome e código de funcionário relevante. Esta operação só terá sucesso se o nome da equipa e o código do funcionário já existirem no sistema e se o funcionário não estiver já integrado numa equipa diferente. Esta operação pode ter dois efeitos diversos. Se o funcionário já fizer parte da equipa em questão, será removido da equipa. Se o funcionário não estiver integrado em nenhuma equipa, será adicionado à equipa;
   f) *Remover Equipa,* dando o seu nome. Esta operação só terá sucesso se o nome da equipa existir no sistema. A remoção da equipa implica a remoção de todas as referências à mesma no sistema, nomeadamente, os seus membros deixarão de estar integrados numa equipa;
   g) *Listar equipa,* dando o seu nome. Esta operação só terá sucesso se o nome da equipa existir no sistema e deve devolver os nomes dos membros da equipa, ordenados alfabeticamente.

   Espera-se que o número de funcionários e equipas sejam da ordem dos milhares. A equipas são geralmente de curta duração e sua composição varia bastante ao longo do tempo.
   Com base nesta especificação, explicite detalhadamente as estruturas de dados mais adequadas e as variáveis de instância mais adequadas para implementar esta aplicação, 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.**