Algoritmos e Estruturas de Dados

Departamento de Informática, Universidade Nova de Lisboa

Segundo teste - 13 de dezembro de 2017

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

1. Considere as Árvores AVL apresentadas nas Figuras 1 e 2. O valor apresentado dentro do nó de cada árvore refere a chave do mesmo nó. Tendo em conta estes dados, efetue as seguintes operações:
   a) Desenhe a árvore binária de pesquisa resultante de inserir um nó com chave 22 na árvore da Figura 1, sem efetuar rotações. Considera que a árvore resultante é uma árvore AVL? Se a sua resposta for negativa, efetue a rotação necessária para transformá-la em AVL.
   b) Desenhe agora a árvore binária de pesquisa resultante de remover o nó com chave 18 da AVL apresentada na Figura 2, sem efetuar rotações. Considera que a árvore resultante é uma árvore AVL? Se a sua resposta for negativa, efetue a rotação necessária para transformá-la em AVL.

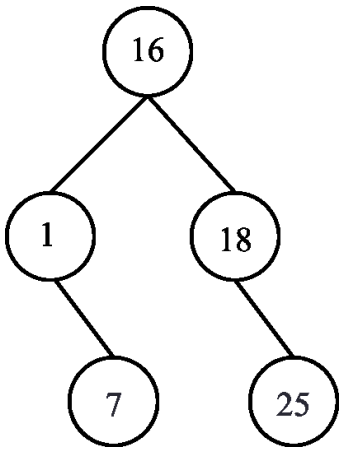

Fig. 1

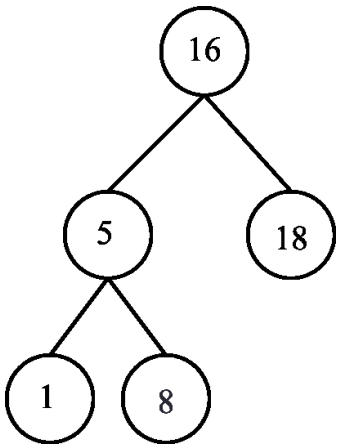

Fig. 2

(O enunciado continua na página 2)

2. Considere a classe BinarySearchTree apresentada nas aulas teóricas de AED. Pedimos-lhe que implemente o método sameShapeTree, um método recursivo que verifica se duas árvores binárias de pesquisa (ABP) são iguais. Para este exercício, duas ABP serão consideradas iguais se contiverem as mesmas entradas (o que deve ser verificado através das chaves das entradas) e se tiverem formas idênticas. O método público recebe uma árvore como parâmetro e vai compará-la com a árvore corrente (this). Na verdade, o método público irá chamar o método privado recursivo com o mesmo nome, que recebe dois nós BSTNode<K,V> e que verifica se as sub-árvores passadas como parâmetros são iguais. O cabeçalho do método recursivo é apresentado abaixo. Calcule ainda a complexidade temporal do método no pior caso, justificando.

```
public boolean sameShapeTree(BinarySearchTree<K,V> otherTree) {
   if (this.size()!=otherTree.size())
     return false;
   return sameShapeTree(this.root, otherTree.root);
}

//Devolve true se as sub-árvores com raiz em node1 e node2
//forem iguais
private boolean sameShapeTree(BSTNode<K,V> node1, BSTNode<K,V> node2){
         ...

}
```

3. O Sistema de Gestão de Autores da Sociedade Portuguesa de Autores (SPA) armazena informação sobre todos os autores portugueses, incluindo os prémios que estes ganharam. A informação sobre cada Autor é estruturada através do Tipo Abstrato de Dados (TAD) Author, apresentado abaixo.

```
public interface Author {

   //Nome do Autor
   String getAuthorName();

   //Data (em String) de nascimento do Autor
   String getAuthorBirth();

   //Nacionalidade do Autor
   String getAuthorNationality();

   //Lista de prémios obtidos pelo autor,
   //ordenada por ordem decrescente de importância
   Iterator<String> listAwards();

}
```

O Sistema da SPA deverá permitir a gestão integrada de vários autores. Tal como se pode ver na interface Java do TAD Author, uma implementação do mesmo TAD deverá conter a lista de prémios ganhos pelo autor.

(O exercício 3 continua na página seguinte)

Pedimos-lhe que implemente o Iterador `BestAwardsIterator` que recebe, no seu construtor, um Iterador de autores e que implementa a iteração dos autores laureados (autores que já receberam prémios no passado) devolvendo, em cada iteração, o nome do autor e o melhor prémio por ele recebido. Para o desenvolvimento do iterador poderá utilizar a definição apresentada abaixo e considerar o seguinte:

- Os autores não premiados não são devolvidos por este iterador;
- O iterador `BestAwardsIterator` é um iterador de Entradas do tipo `Entry<K,V>` (`Entry<String, String>`), onde `K` será o nome do autor e `V` será o título do prémio de maior importância atribuído ao mesmo autor.

Para facilitar o desenvolvimento do iterador poderá assumir a implementação da `EntryClass<K,V>` como disponível de forma pública. Pode implementar todos os métodos auxiliares de que necessitar.

Determine a complexidade temporal de todos os métodos desenvolvidos, no pior caso, **justificando**. Para este efeito pode assumir que a estrutura de dados usada para guardar os autores, assim como para a lista de prémios de um autor, é uma lista duplamente ligada, tal como a apresentada nas aulas teóricas de AED.

```
public class BestAwardsIterator implements Iterator<Entry<String,String>>{

   //Construtor
   public BestAwardsIterator(Iterator<Author> authorIt);

   //(Re-)inicia a iteração.
   public void rewind();

   //Devolve true se e só se a iteração ainda não terminou.
   public boolean hasNext();

   //Devolve o próximo laureado e o prémio mais importante que recebeu
   public Entry<String,String> next() throws NoSuchElementException;
}
```

4. Uma grande cadeia de lojas pretende implementar um sistema de gestão de clientes a partir das fichas de clientes de cada loja. Pedimos-lhe que apoie o desenvolvimento do processo mensal de atualização dos dados. Mensalmente, cada loja envia a sua lista de clientes para o sistema central, sob a forma de um iterador de clientes. Cada *Cliente* contém a seguinte informação: identificador fiscal único (NIF), nome completo, morada e gasto mensal (em Euros) do cliente. Nesta altura, o sistema global é atualizado, com alguns cuidados. Os dados dos clientes já existentes devem ser atualizados, e os novos clientes devem ser inseridos no sistema. Este deve estar preparado para, em qualquer momento, disponibilizar a listagem de todos os novos clientes, ou seja, de todos os clientes que ainda não estavam no sistema no mês anterior. Esta primeira listagem deve ser ordenada pelo nome do cliente. Uma outra listagem que deve estar sempre preparada é a dos melhores clientes do mês. Os melhores clientes do mês são todos o que gastaram o valor máximo de compras (em Euros), despendido até ao momento. A cadeia de lojas envia, no final do mês, um presente aos melhores clientes do mês que vai terminar.

(O exercício 4 continua na página seguinte)

É importante ter em conta que, se num determinado momento, um cliente ultrapassa o máximo atual, todos os clientes que faziam parte da lista de melhores clientes do mês, perdem esse estatuto. Esta listagem não tem uma ordem pré-definida. No final de cada mês, os valores de compras despendidos no mês que termina são inicializados e os novos clientes deixam de o ser.

Assim, o sistema deverá permitir as seguintes operações:

a) *Atualizar Dados do sistema central*, recebendo os dados dos clientes de uma determinada loja. Nesta operação, o sistema recebe um iterador de Clientes, em que cada cliente contém a seguinte informação: número de contribuinte (NIF), nome completo, morada e valor mensal (em Euros) gasto na loja. Cada loja envia estes dados apenas uma vez por mês e esta assiduidade não precisa de ser controlada. No entanto, os dados recebidos devem ser usados para apoiar a geração, quando necessário, das listagens das alíneas b) e c). Se duas lojas enviarem dados relativos ao mesmo cliente, os dados deverão ser sempre atualizados e o valor despendido pelo cliente dever ser incrementado;
b) *Listar dados dos melhores clientes do mês, até ao momento,* sem qualquer ordem definida;
c) *Listar dados dos novos clientes do mês que decorre,* ordenados pelo nome do cliente. Pode assumir que o nome do cliente é único;
d) *Iniciar mês*. Quando o mês termina, os dados guardados no sistema devem ser preparados para o novo mês que se inicia.

Espera-se que o número de clientes seja da ordem dos milhares. A cadeia é composta de centenas de lojas. Com base nesta especificação:

- 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. Para este efeito, pode assumir que o iterador de clientes recebido na alínea a) está implementado sobre uma lista duplamente ligada. **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.**

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 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 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**

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