Recurso de Algoritmos e Estruturas de Dados

Departamento de Informática

Universidade Nova de Lisboa

28 de Janeiro de 2011

1. [3.5 valores] Uma árvore *degenerada* é uma árvore cujos nós têm (todos), no máximo, um filho. Portanto, qualquer árvore vazia é degenerada.

```
public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    // The root of the tree.
    protected BSTNode<K,V> root;

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

    .....

    // Returns true if every node of the binary search tree has
    // at most one child.
    public boolean isDegenerate( );
}
```

   Implemente o método *isDegenerate*, programando todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso e no pior caso, justificando.

2. [3 valores] (**Sugestão:** Deixe esta pergunta (em particular, a alínea (b)) para o fim.)
   Relembre que uma árvore vermelha e preta é uma árvore binária de pesquisa cujos nós são coloridos de forma a satisfazer as seguintes propriedades:

   1. cada nó ou é preto ou é vermelho;
   2. a raiz da árvore é preta;
   3. os filhos dos nós vermelhos são pretos; e
   4. todos os caminhos da raiz a uma sub-árvore vazia têm o mesmo número de nós pretos.

   Chamemos p ao número de nós pretos de um (qualquer) caminho da raiz a uma sub-árvore vazia.

   (a) Qual é o número mínimo de nós de uma árvore vermelha e preta quando $p = 2$? Qual é o número máximo de nós de uma árvore vermelha e preta quando $p = 2$? Justifique claramente as suas respostas, desenhando árvores com as características especificadas.

   (b) Generalize o seu raciocínio para um qualquer p positivo. Qual é o número mínimo de nós de uma árvore vermelha e preta, em função de p? Qual é o número máximo de nós de uma árvore vermelha e preta, em função de p? Justifique os seus cálculos de forma precisa e concisa.

3. [4 valores] Considere o tipo abstracto de dados *Ranking de Vendas*, de itens do tipo I, caracterizado pela interface *SalesRanking*.

```
public interface SalesRanking<I extends Comparable<I>>
{
    void addSale( I item, int quantity ) throws NonPositiveQuantityException;

    int size( );

    int getQuantity( I item ) throws NoSuchItemException;

    Iterator<Entry<I, Integer>> iterator( );

    Iterator<I> topItems( );
}
```

A operação *addSale*(i, q) regista uma venda (já realizada) de q exemplares do item i. Esta operação só não é efectuada se $q \leq 0$. Para exemplificar, assuma que se registam as seguintes vendas num *ranking* de vendas `sales`, com itens do tipo carácter, acabado de criar (e sem nenhuma venda registada):

- `sales.addSale('a', 6)` — registo da venda de seis exemplares do item `'a'`;
- `sales.addSale('c', 7)` — registo da venda de sete exemplares do item `'c'`;
- `sales.addSale('a', 1)` — registo da venda de um exemplar do item `'a'`;
- `sales.addSale('b', 4)` — registo da venda de quatro exemplares do item `'b'`.

Neste momento, já se venderam sete exemplares dos itens `'a'` e `'c'`, e quatro exemplares do item `'b'`. Vejamos os resultados das restantes operações.

- *size* retorna o número de itens distintos vendidos. Portanto,

  `sales.size()` retornaria três.

- *getQuantity*(i) retorna o número total de exemplares vendidos do item i. Por exemplo,

  `sales.getQuantity('a')` retornaria sete.

- *iterator*() retorna um iterador de entradas (que preserva a ordem estritamente crescente dos itens), onde cada entrada tem um item vendido e o número total de exemplares vendidos desse item. No nosso caso,

  `sales.iterator()` geraria a sequência (`'a'`, 7), (`'b'`, 4) e (`'c'`, 7), por esta ordem.

- *topItems*() retorna um iterador dos itens mais vendidos, ou seja, dos que estão no primeiro lugar do *ranking*. Assim,

  `sales.topItems()` permitiria obter os itens `'a'` e `'c'`, por qualquer ordem.

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *SalesRanking*, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo I, é sempre constante.

4. [4.5 valores] Considere pares (e, b), onde e é um elemento do tipo E e b é um valor booleano. O elemento e diz-se *verdadeiro*, se o booleano b for *true*; e diz-se *falso*, no outro caso.

   Dada uma sequência S daqueles pares, pretende-se obter uma permutação S' das primeiras coordenadas dos pares de S com as seguintes características:

   - os elementos verdadeiros ocorrem antes dos elementos falsos;
   - a ordem dos elementos verdadeiros é a ordem original;
   - a ordem dos elementos falsos é a ordem inversa da original.

   Por exemplo:

   - se $S = <(1,T),\ (2,T),\ (3,F),\ (4,T),\ (5,T),\ (6,F),\ (7,F)>$, então $S' = <1,\ 2,\ 4,\ 5,\ 7,\ 6,\ 3>$;
   - se $S = <(a,T),\ (x,T),\ (a,T),\ (y,T)>$, então $S' = <a,\ x,\ a,\ y>$;
   - se $S = <(a,F),\ (x,F),\ (x,F)>$, então $S' = <x,\ x,\ a>$;
   - se $S = <(3,F),\ (2,F),\ (5,F),\ (5,T)>$, então $S' = <5,\ 5,\ 2,\ 3>$;
   - se S é a sequência vazia, então S' é a sequência vazia.

   Suponha que ambas as sequências são geradas por iteradores. A classe *StrangeIterator* tem um construtor que recebe um iterador da sequência S (original) e devolve um iterador da sequência S'. Defina as suas variáveis de instância, implemente o construtor e os dois[1] métodos públicos e programe todos os métodos auxiliares (da classe *StrangeIterator*) de que necessitar.

```
public interface Pair<F,S>
{
    // Returns the first coordinate of the pair.
    F getFirst( );

    // Returns the second coordinate of the pair.
    S getSecond( );
}

public class StrangeIterator<E> implements Iterator<E>
{
    public StrangeIterator( Iterator<Pair<E, Boolean>> sequence );

    // Returns true iff the iteration has more elements.
    // In other words, returns true if next would return an element
    // rather than throwing an exception.
    public boolean hasNext( );

    // Returns the next element in the iteration.
    public E next( ) throws NoSuchElementException;
}
```

   Calcule (justificando) as complexidades temporais do construtor e dos dois métodos públicos, no melhor caso e no pior caso.

[1] Para simplificar o exercício, a operação *rewind* foi abolida da classe *StrangeIterator*.

5. [5 valores] Uma agência imobiliária pretende construir uma aplicação para gerir os imóveis que tem para transaccionar.

   Os dados associados a cada *imóvel* são: um *número-chave* único que o identifica, o *tipo* (apartamento, moradia, etc.), o *número de assoalhadas*, o *ano* de construção, a *morada*, o *preço*, uma *planta*, uma *fotografia*, o *estado de conservação* (em construção, a estrear, recente, remodelado ou a necessitar de obras), o *estado de venda* (disponível, reservado ou sinalizado) e o nome do *proprietário*. Cada *proprietário* tem um *nome* (que se assume ser único, para simplificar), uma *morada*, um *número de telefone* e um *número de contribuinte*.

   O sistema permite efectuar as seguintes operações.

   (a) Inserir um proprietário, dando o nome, a morada, o número de telefone e o número de contribuinte.

   A operação só será efectuada se não existir um proprietário com esse nome.

   (b) Inserir um imóvel, dando todos os seus dados.

   A operação só será efectuada se não existir um imóvel com esse número-chave e se existir um proprietário com esse nome.

   (c) Remover um imóvel, dando o seu número-chave.

   A operação só será efectuada se existir um imóvel com esse número-chave.

   (d) Obter todos os dados de um imóvel, dando o seu número-chave.

   (e) Obter a morada, o número de telefone, o número de contribuinte e os números-chave de todos os imóveis de um proprietário, dando o seu nome.

   Os números-chave deverão aparecer pela ordem por que foram inseridos no sistema.

   (f) Listar todos os imóveis com um determinado número de assoalhadas que se encontrem dentro de um intervalo de preços.

   A listagem deve estar ordenada crescentemente por preço e, em caso de igualdade no preço, por ordem crescente de número-chave.

   Informação por imóvel: número-chave, tipo, morada, preço, estado de conservação e estado de venda.

   Prevê-se que o número de imóveis e o número de proprietários não ultrapassem 100 000. Regra geral, em cada momento, cada proprietário não deverá ter mais de dez imóveis. Assuma que qualquer imóvel tem entre uma e vinte assoalhadas e que cerca de 95% dos imóveis transaccionados pela imobiliária têm preços entre 50 000 e 800 000 euros, com uma distribuição uniforme dentro desta gama de valores.

   Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar esta aplicação, descreva sumariamente os algoritmos para efectuar as seis operações (enumeradas de (a) a (f)) e calcule (justificando) as suas complexidades temporais, no caso esperado.