

Exame de Algoritmos e Estruturas de Dados

Departamento de Informática

Universidade Nova de Lisboa

7 de Janeiro de 2011

1. [3.5 valores] Um *multiconjunto* é uma colecção de elementos que admite repetições. Por exemplo, $\{a, b, a, d, a, c, c\}$ é um multiconjunto com sete elementos (o “*size*” é 7). Se se inserisse ‘a’, o multiconjunto passaria a ser $\{a, b, a, d, a, c, c, a\}$. Se, depois, se removesse ‘c’, o multiconjunto resultante seria $\{a, b, a, d, a, c, a\}$. Note que a ordem de apresentação dos elementos é irrelevante. É fácil concluir que um multiconjunto é um conjunto quando não tem elementos repetidos.

Considere o tipo abstracto de dados *Multiconjunto*, de elementos do tipo E, caracterizado pela interface *Multiset*.

```
public interface Multiset<E>
{
    // Returns the number of elements in the multiset.
    int size( );

    // Returns true iff the multiset contains the specified element.
    boolean contains( E element );

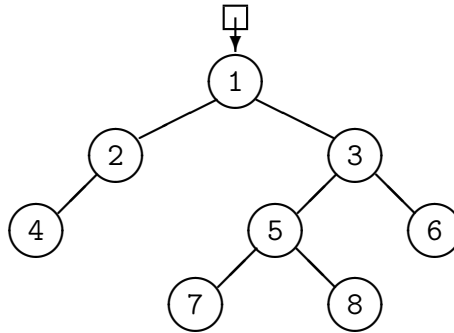
    // Inserts the specified element in the multiset.
    void insert( E element );

    // Removes one occurrence of the specified element from the multiset
    // and returns true, if the multiset contains the element.
    // Otherwise, returns false.
    boolean remove( E element );

    // Returns true iff the multiset is a set.
    boolean isASet( );
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *Multiset*, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no caso esperado, justificando. Se quiser, pode assumir que os elementos são comparáveis entre si (E extends Comparable<E>), com complexidade constante.

2. Numa árvore binária, cada nó pode ser identificado pelo *caminho* da raiz ao nó. Esse caminho é descrito por uma sequência de booleanos, onde *false* indica que se prossegue pela sub-árvore esquerda e *true* indica que se prossegue pela sub-árvore direita.



Para exemplificar, considere a árvore esquematizada na figura. O caminho:

- que identifica a raiz da árvore (com o número 1) é vazio;
- “*false*” identifica o nó com o número 2;
- “*false false*” identifica o nó com o número 4;
- “*true false true*” identifica o nó com o número 8.

A sequência “*false true*” não é um caminho, porque não corresponde a qualquer nó da árvore. Note que não há caminhos nas árvores vazias.

```

public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    protected BSTNode<K,V> root;    // The root of the tree.
    protected int currentSize;       // Number of elements in the tree.
    .....

    public Iterator<Boolean> maxPath( ) throws EmptyTreeException;

    public Entry<K,V> entryAt( Iterator<Boolean> path )
        throws NoSuchElementException;
}

```

- (a) [4 valores] O método *maxPath* retorna um (qualquer) dos maiores caminhos da árvore. No nosso exemplo, esse caminho poderia ser “*true false false*” ou “*true false true*”. Implemente o método *maxPath* e programe todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.
- (b) [4 valores] O método *entryAt* retorna a entrada do nó identificado pelo caminho passado em argumento. Por exemplo, se o caminho for vazio e a árvore não for vazia, a entrada retornada é a que se encontra na raiz da árvore. Implemente o método *entryAt* e programe 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.

3. [5 valores] Pretende-se construir uma aplicação para gerir um sistema de apostas parecido com o Euromilhões, com sorteios diários realizados à meia noite GMT (*Greenwich Mean Time*).

Quer as *apostas*, quer as *chaves sorteadas*, são constituídas por cinco números distintos (entre 1 e 50) e por duas estrelas distintas (entre 1 e 9).

Cada aposta custa dois euros, mas o valor do *prémio* (diário) é igual ao número de apostas feitas nesse dia, em euros. Se nenhuma aposta feita nesse dia for exactamente igual à chave sorteadada, o valor do prémio transita para o dia seguinte; nos restantes casos, o prémio é igualmente dividido pelos jogadores que acertaram na chave sorteadada. Para simplificar, assuma que esta divisão dá sempre resto zero e que, em cada dia, as apostas de cada jogador são todas distintas.

Cada *jogador* tem um *código* único que o identifica, um *nome*, um *número de contribuinte* e um *saldo*, que é usado para pagar as apostas. Os prémios ganhos pelo jogador aumentam o saldo.

O sistema permite efectuar as seguintes operações.

- (a) Inserir um jogador no sistema (com saldo zero), dando o código, o nome e o número de contribuinte do jogador.

A operação só será efectuada se não existir um jogador com esse código.

- (b) Alterar o saldo de um jogador, dando o seu código e o valor (negativo ou positivo) a somar ao seu saldo.

A operação só será efectuada se existir um jogador com esse código e se o respectivo saldo actual somado ao valor dado não for negativo.

- (c) Registrar uma aposta (no sorteio desse dia), dando o código do jogador e a aposta.

A operação só será efectuada se existir um jogador com esse código, se o respectivo saldo for superior ou igual a dois euros e se, nesse dia, o jogador ainda não tiver feito uma aposta igual.

- (d) Inserir a chave sorteadada (no sorteio desse dia), atribuindo os prémios.

- (e) Listar todos os jogadores premiados num determinado dia, dando o dia.

Informação por jogador: código, nome e número de contribuinte.

A listagem deve estar ordenada por código de jogador.

- (f) Listar todos os prémios ganhos por um jogador, dando o seu código.

Informação por prémio: dia e valor do prémio recebido nesse dia.

A listagem deve estar ordenada cronologicamente.

Assuma que o dia corrente é obtido chamando uma função da biblioteca e que as chaves sorteadas são sempre inseridas à meia noite GMT.

Prevê-se que o sistema contenha cerca de meio milhão de jogadores e que muitos deles registem várias apostas diariamente.

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.

4. [3.5 valores] Pretende-se aplicar a técnica da função-memória ao cálculo da distância de Levenshtein entre duas sequências de caracteres.

A *distância de Levenshtein* entre duas sequências de caracteres indica o número mínimo de *operações de edição* necessárias para transformar a primeira sequência na segunda. As operações de edição são de três tipos: inserir um carácter; remover um carácter; e substituir um carácter por outro carácter.

Por exemplo, a distância de Levenshtein entre “a passar” e “ao nadar” é 4. É possível transformar a primeira sequência na segunda, efectuando uma inserção, duas substituições e uma remoção. A inserção de ‘o’ a seguir ao primeiro ‘a’ permite obter “ao”. As substituições de ‘p’ por ‘n’ e do primeiro ‘s’ por ‘d’ transformam “passar” em “nadsar”. Finalmente, com a remoção de ‘s’ obtém-se “nadar”. Note que há outros conjuntos de operações de edição que atingem o mesmo objectivo, mas nenhum possui menos de quatro elementos.

```
public static int levenshteinDistance( char[] seq1, char[] seq2 )
{
    return dist(seq1, seq2, seq1.length, seq2.length);
}

private static int dist( char[] x, char[] y, int lenX, int lenY )
{
    if ( lenX == 0 )
        return lenY;
    else if ( lenY == 0 )
        return lenX;
    else
        return minOf3( dist(x, y, lenX-1, lenY-1) + diff(x[lenX-1], y[lenY-1]),
                        dist(x, y, lenX-1, lenY) + 1,
                        dist(x, y, lenX, lenY-1) + 1 );
}

private static int minOf3( int value1, int value2, int value3 )
{
    return Math.min( Math.min(value1, value2), value3 );
}

private static int diff( char elem1, char elem2 )
{
    return ( elem1 == elem2 ? 0 : 1 );
}
```

O problema é que a complexidade temporal deste algoritmo é exponencial. Dados dois vectores de caracteres (completamente preenchidos), *seq1* e *seq2*, no cálculo de *levenshteinDistance(seq1, seq2)*, executam-se muitas chamadas recursivas iguais. Repare que, em todas elas, os dois primeiros argumentos são iguais.

Apresente um algoritmo polinomial que calcule o mesmo valor, aplicando a técnica da função-memória ao algoritmo dado. Determine as complexidades temporal e espacial do seu algoritmo, no melhor caso e no pior caso, justificando todos os cálculos com muita clareza.