Recurso de Algoritmos e Estruturas de Dados

Departamento de Informática

Universidade Nova de Lisboa

5 de Fevereiro de 2009

1. [3.5 valores] Numa árvore genérica, cada nó possui um número arbitrário de filhos. Neste exercício, os filhos estão guardados num vector. Mais especificamente, se um nó tem k filhos (com $k \geq 0$), os filhos estão guardados nas posições $0, 1, \ldots, k-1$ do vector.

   O *grau* de uma árvore não vazia é o maior número de filhos que algum nó da árvore possui. A noção de grau não está definida para árvores vazias.

   Por exemplo, o grau de uma árvore binária só pode ser: zero (quando a árvore tem um único nó); um (quando a árvore tem pelo menos dois nós e todo o nó da árvore ou é folha ou só tem um filho); ou dois (quando pelo menos um nó da árvore tem dois filhos).

   Considere então a classe *TreeNode*, dos nós das árvores genéricas de elementos do tipo E, e a classe *Tree*, das árvores genéricas de elementos do tipo E, ambas no pacote *dataStructures*. Implemente o método *getDegree* (na classe *Tree*), que calcula o grau da árvore, e programe todos os métodos auxiliares desta classe de que necessitar.

```
class TreeNode<E>
{
    protected E element;                    // Element stored in the node.
    protected TreeNode<E>[] children;       // (Pointers to) the children.
    protected int childrenSize;             // Number of children.

    ..................................

    // Returns the element in the node.
    public E getElement( );

    // Returns the number of children of the node.
    public int getNumberOfChildren( );

    // Returns the child stored at the specified position of
    // the node's array of children.
    public TreeNode<E> getChild( int position ) throws InvalidPositionException;
}

public class Tree<E>
{
    protected TreeNode<E> root;             // The root of the tree.

    ..................................

    // Returns the tree's degree.
    public int getDegree( ) throws EmptyTreeException;
}
```

   Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

2. [4.5 valores] Um polinómio de coeficientes inteiros pode ser implementado através de uma lista de pares do tipo <Integer, Integer>. As componentes do par representam, respectivamente, o grau e o coeficiente do termo do polinómio. Se se percorrer os elementos da lista pela ordem natural (ou seja, das posições menores para as maiores), os termos do polinómio ocorrem por ordem estritamente crescente de grau. Não existem coeficientes nulos (nem termos com o mesmo grau).

   A figura seguinte exemplifica esta implementação com três polinómios. Do lado esquerdo, o polinómio está representado na forma usual e, do lado direito, apresenta-se a sequência de pares guardada na lista. A posição de cada par na lista está escrita a *bold* por baixo do par.

$4 + 2\,x + 5\,x^4 - x^7$	$(0, 4)$	$(1, 2)$	$(4, 5)$	$(7, -1)$
	0	**1**	**2**	**3**
$-2\,x - x^2 - 8\,x^4$	$(1, -2)$	$(2, -1)$	$(4, -8)$	
	0	**1**	**2**	
$4 - x^2 - 3\,x^4 - x^7$	$(0, 4)$	$(2, -1)$	$(4, -3)$	$(7, -1)$
	0	**1**	**2**	**3**

   Repare que o último polinómio pode ser obtido somando os dois primeiros.

   Considere a classe *Polynomial*, dos polinómios de coeficientes inteiros implementados através de uma lista duplamente ligada, com cabeça e cauda. Ou seja, o atributo *list* é um objecto da classe *DoublyLinkedList*.

   O método *sum* constrói e retorna o polinómio soma, que se obtém somando o polinómio passado em argumento ao polinómio que recebe a mensagem. O polinómio soma não partilha posições de memória com os polinómios somados, que não podem ser alterados. Para simplificar, assuma que nenhum dos polinómios é 0 (zero).

```
package myMath;

public class Polynomial
{
    // The polynomial is stored in a list.
    protected List<Pair<Integer, Integer>> list;

    ..........................................

    // Builds and returns the sum of the specified polynomial
    // with this polynomial.
    public Polynomial sum( Polynomial polynomial );
}
```

   Implemente o método *sum* (na classe *Polynomial*) e programe todos os métodos auxiliares desta classe de que necessitar. Note que a classe *Polynomial* não pertence ao pacote *dataStructures* (ao contrário das interfaces e das classes do apêndice). Calcule a complexidade temporal do seu algoritmo, no caso esperado, justificando.

3. [3.5 valores] Para simplificar este exercício, assuma que **todas as entradas de uma fila com prioridade têm chaves distintas**. Neste contexto, pode-se afirmar que o método *minEntry* da interface *MinPriorityQueue* retorna a entrada com a menor chave. Suponha que se acrescenta um novo método àquela interface (chamado *2ndMinEntry*), que devolve a entrada com a segunda menor chave.

   Por exemplo, se a fila com prioridade tivesse as entradas (onde a chave é o primeiro elemento do par)

   $$(3, \text{'z'}),\ (4, \text{'a'}),\ (7, \text{'b'})\ \text{ e }\ (10, \text{'j'}),$$

   minEntry e *2ndMinEntry* retornariam $(3, \text{'z'})$ e $(4, \text{'a'})$, respectivamente.

```
public interface Entry<K,V>
{
    K getKey( );
    V getValue( );
}

public class MinHeap<K extends Comparable<K>, V>
    implements MinPriorityQueue<K,V>
{
    // Default capacity of the priority queue.
    public static final int DEFAULT_CAPACITY = 1000;

    // Memory of the priority queue: an array.
    protected Entry<K,V>[] array;

    // Number of entries in the priority queue.
    protected int currentSize;

    .....

    // Returns the entry with the second smallest key in the priority queue.
    public Entry<K,V> 2ndMinEntry( ) throws NoSuchEntryException;
}
```

   Implemente o método *2ndMinEntry* na classe *MinHeap* e programe todos os métodos auxiliares desta classe de que necessitar. Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo K, é sempre constante.

(Continue, porque o exame tem mais duas perguntas.)

4. [3.5 valores] Considere o tipo abstracto de dados *Fila com Prioridade Inteligente*, de chaves do tipo K e valores do tipo V, caracterizado pela interface *SmartMinPriorityQueue*.

```
public interface SmartMinPriorityQueue<K extends Comparable<K>, V>
{
    // Returns the number of entries in the priority queue.
    int size( );

    // Returns an entry with the smallest key in the priority queue.
    Entry<K,V> minEntry( ) throws EmptyPriorityQueueException;

    // If the priority queue contains an entry of the form (otherKey, value),
    // with otherKey ≤ key, this method does nothing except returning false;
    // otherwise, inserts the entry (key, value) in the priority queue and returns true.
    boolean insert( K key, V value );

    // Removes an entry with the smallest key from the priority queue
    // and returns that entry.
    Entry<K,V> removeMin( ) throws EmptyPriorityQueueException;
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar este tipo abstracto de dados, descreva brevemente como implementaria as quatro operações e calcule as suas complexidades temporais, no caso esperado, justificando. Se quiser, pode assumir que o tipo V é comparável.

5. [5 valores] Pretende-se criar um sistema para inventariar automóveis. Cada *automóvel* é identificado por uma *marca* e um *modelo*; tem uma *capacidade* do depósito de combustível, medida em litros; e possui uma *autonomia*, que é o número máximo de kilómetros que pode percorrer com um depósito cheio. Assuma que a capacidade e a autonomia são números inteiros.

   A *eficiência* de um automóvel, que indica o número de kilómetros percorridos por litro de combustível, é definida por $\frac{A}{C}$, onde A é a autonomia e C é a capacidade do automóvel. É óbvio que podem existir automóveis com a mesma eficiência, mas com autonomias e capacidades diferentes.

   O sistema deve permitir efectuar as seguintes operações.

   (a) Inserir um automóvel, dando a marca, o modelo, a capacidade e a autonomia.
   A operação não será efectuada se existir um automóvel com a mesma marca e o mesmo modelo.

   (b) Alterar as características de um automóvel, dando a marca e o modelo (que se mantêm) e os novos valores da capacidade e da autonomia.

   (c) Remover um automóvel, dando a marca e o modelo.

   (d) Listar, por ordem alfabética, todos os modelos dos automóveis com uma dada marca.

   (e) Listar, por uma ordem arbitrária, todos os automóveis que têm a máxima eficiência.
   Informação por automóvel: marca, modelo, capacidade e autonomia.

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

Anexo — Parte do Pacote *dataStructures*

```
public interface Iterator<E>
{
    boolean hasNext( );
    E next( ) throws NoSuchElementException;
    void rewind( );
}
public interface Pair<F,S>
{
    F getFirst( );
    S getSecond( );
    void setFirst( F newFirst );
    void setSecond( S newSecond );
}
public class PairClass<F,S> implements Pair<F,S>
{
    public PairClass( F theFirst, S theSecond );
    .....
}
public interface List<E>
{
    boolean isEmpty( );
    int size( );
    Iterator<E> iterator( );
    E getFirst( ) throws EmptyListException;
    E getLast( ) throws EmptyListException;
    // Range of valid positions: 0, ..., size()−1.
    E get( int position ) throws InvalidPositionException;
    int find( E element );
    void addFirst( E element );
    void addLast( E element );
    // Range of valid positions: 0, ..., size().
    void add( int position, E element ) throws InvalidPositionException;
    E removeFirst( ) throws EmptyListException;
    E removeLast( ) throws EmptyListException;
    // Range of valid positions: 0, ..., size()−1.
    E remove( int position ) throws InvalidPositionException;
    boolean remove( E element );
}
public class DoublyLinkedList<E> implements List<E>
{
    public DoublyLinkedList( );
    .....
}
```