Exame de Algoritmos e Estruturas de Dados I

Departamento de Informática

Universidade Nova de Lisboa

4 de Julho de 2007

1. [3.5 valores] Descobriu que os algoritmos de inserção e de remoção das listas duplamente ligadas não estão a afectar correctamente o apontador para o nó anterior. Felizmente, a cabeça, a cauda e o número de elementos da lista, bem como todos os apontadores para o nó seguinte, estão correctos. Implemente o método *restoreAllPreviousPointers* (na classe *DoublyLinkedList*), que restabelece a coerência da lista, e calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

```
public class DoublyLinkedList<E> implements List<E>
{
    .....

    protected void restoreAllPreviousPointers( );
}
```

2. [3.5 valores] Considere o tipo abstracto de dados *Dicionário Reversível*, de chaves do tipo K e valores do tipo V, caracterizado pela interface *ReversibleDictionary*.

```
public interface ReversibleDictionary<K,V>
{
    // Returns the value in the dictionary whose key is the specified key;
    // or null if no such entry exists.
    V find( K key );

    // Returns the key in the dictionary whose value is the specified value;
    // or null if no such entry exists.
    K findReverse( V value );

    // Inserts the entry (key, value) in the dictionary and returns true,
    // if the dictionary contains neither an entry with the specified key,
    // nor an entry with the specified value. Otherwise, returns false.
    boolean insert( K key, V value );

    // Removes the entry whose key is the specified key from the dictionary
    // and returns the associated value, if such an entry exists.
    // Otherwise, returns null.
    V remove( K key );

    // Removes the entry whose value is the specified value from the dictionary
    // and returns the associated key, if such an entry exists.
    // Otherwise, returns null.
    K removeReverse( V value );
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar este tipo abstracto de dados, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando. Se quiser, pode assumir que os tipos K e V são comparáveis.

3. [3.5 valores] Considere a função recursiva $f_s(i, v)$, onde $s = (x_0\, x_1\, \cdots\, x_{n-1})$ é uma sequência não vazia de números inteiros positivos, i é um inteiro entre 0 e $n-1$, e v é um inteiro não negativo.

$$f_s(i,v) = \begin{cases} x_0^2, & \text{se } i = 0 \text{ e } v \neq x_0; \\ 1, & \text{se } i = 0 \text{ e } v = x_0; \\ \max(\ f_s(i-1, v-x_i-1), & \\ \qquad f_s(i-1, v-x_i)\ ), & \text{se } i > 0 \text{ e } v - x_i > 0; \\ x_i + f_s(i-1, v), & \text{nos restantes casos.} \end{cases}$$

O método *fMem*(seq, value) calcula o valor da função $f_{\text{seq}}(\text{seq.length} - 1, \text{value})$, quando a sequência está implementada num vector (ocupando as posições $0, \ldots, \text{seq.length} - 1$).

```
// Calcula o valor de f_seq(seq.length − 1, value).
// Assuma que o vector tem seq.length inteiros positivos, que seq.length > 0
// e que value > 0.
static int fMem( int[] seq, int value );
```

Implemente *fMem*, aplicando a técnica da função memória, e calcule a complexidade temporal e espacial do seu algoritmo, no melhor caso, no pior caso e no caso esperado.

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

4. [5 valores] Pretende-se desenvolver uma aplicação para simular entradas e saídas de veículos num enorme parque de estacionamento. O parque tem E entradas (identificadas pelos números $1, \ldots, E$), S saídas (identificadas pelos números $1, \ldots, S$) e capacidade para C veículos (onde E, S e C são inteiros positivos).

   Como em todos os parques de estacionamento, nem sempre é possível satisfazer imediatamente os pedidos de entrada de veículos. Quando o condutor prime o botão da respectiva cancela, a cancela abre-se de imediato se o parque não estiver cheio (ou seja, se o número de veículos que entraram e ainda não saíram for inferior a C). Mas, se o parque estiver lotado, o condutor tem que esperar que saia um veículo para poder entrar.

   Quando há vários condutores à espera (um em cada cancela), a regra geral é que entra o que primeiro tiver premido o respectivo botão. No entanto, a política para atrair clientes frequentes é atribuir-lhes uma bonificação no tempo do pedido de entrada. Se as bonificações derem origem a igualdades, entra primeiro o que efectivamente tiver chegado primeiro.

   Por exemplo, se estiverem três veículos à espera:

   - um, na cancela 10, que premiu o botão às 16h 12 e não tem bonificações;
   - outro, na cancela 52, que premiu o botão às 16h 17 e tem uma bonificação de 10 minutos; e
   - outro, na cancela 4, que premiu o botão às 16h 22 e tem uma bonificação de 15 minutos,

   entra primeiro o da cancela 52, depois o da cancela 4 e, só depois, o da cancela 10. Note que, para este efeito, é como se os veículos nas cancelas 52 e 4 tivessem chegado às 16h 07. Este empate é resolvido a favor do da cancela 52, porque, de facto, ele chegou antes do da cancela 4.

   A aplicação arranca com um parque vazio, lê os dados de um ficheiro e escreve os resultados noutro. Ambos os ficheiros estão ordenados cronologicamente (por instante). As operações são de três tipos.

   (a) **Entrada de Veículo** (Note que $t' \geq t$.)
   - Dados: no instante t, ocorreu um pedido de entrada do veículo com a matrícula m e a bonificação b, efectuado na cancela de entrada e.
   - Resultados: no instante t', o veículo com a matrícula m entrou pela cancela e.

   (b) **Saída de Veículo**
   - Dados: no instante t, ocorreu a saída do veículo com a matrícula m, na cancela s.
   - Resultados: no instante t, o veículo com a matrícula m saíu, tendo pago v.

   (c) **Contagem dos Veículos no Parque**
   - Dados: no instante t, quantos veículos estavam no parque?
   - Resultados: no instante t, estavam n veículos no parque.

   Assuma que o ficheiro de entrada está correcto e que os tempos são medidos com uma precisão tal que nunca há dois instantes iguais no ficheiro dos dados. Considere que dispõe de uma função que, dados os instantes de entrada e de saída do veículo no parque, calcula o valor pago. Pode comparar, somar e subtrair instantes.

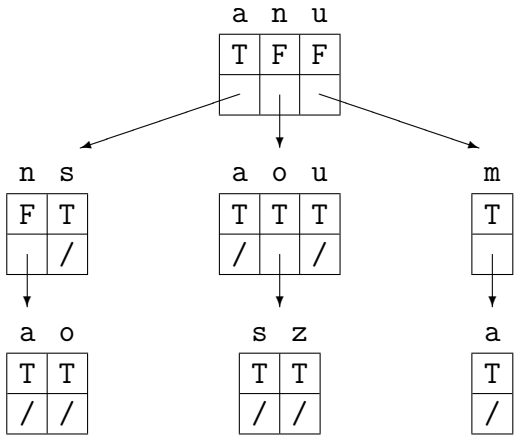
   Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar esta aplicação, descreva sumariamente o algoritmo de simulação e os três sub-algoritmos (de entrada, saída e contagem de veículos) e calcule as suas complexidades temporais, no caso esperado.

5. [4.5 valores] Uma das estruturas de dados usadas para implementar prontuários ortográficos (conjuntos de palavras de uma língua) é uma árvore (*DictTree*) cujos nós (do tipo *DictNode*) possuem um dicionário. No exemplo da figura, o dicionário está representado por uma espécie de vector indexado pela chave. Por exemplo, o dicionário do nó raiz:

   - associa a letra `a` (a chave) a um par cuja primeira componente é o valor booleano *true* e cuja segunda componente é um nó da árvore (do tipo *DictNode*);
   - associa a letra `n` a um par cuja primeira componente é o valor booleano *false* e cuja segunda componente é outro nó da árvore;
   - não tem qualquer entrada cuja chave seja a letra `z`.

   Como estamos a analisar o nó raiz, significa que, no prontuário da figura, todas as palavras começam por `a`, `n` ou `u`. O valor booleano indica que `a` é uma palavra, mas `n` e `u` não são. Olhando para o filho da raiz associado à letra `a`, concluimos que há palavras que começam por `an` e `as`, mas só `as` é palavra.

   Uma sequência de letras é uma palavra do prontuário, se existir um percurso na árvore (a começar na raiz) cuja concatenação das letras é igual à palavra e cujo valor booleano associado à última letra da palavra é *true*.



Palavras do Prontuário:

`a`, `ana`, `ano`, `as`,

`na`, `no`, `nos`, `noz`, `nu`,

`um` e `uma`.

   Implemente o método *containsWord* (na classe DictTree), que indica se uma palavra não vazia (gerada por um iterador) pertence ao prontuário. Para isso, considere o seguinte contexto de interfaces e classes, que generaliza o tipo da chave (no nosso exemplo, K é *Character*).

```
public interface Pair<F,S>   {   F getFirst();   S getSecond();   }
public interface DictNode<K>
{     // Returns the pair in the node's dictionary whose key is the specified key;
      // or null if no such entry exists.
      Pair<Boolean, DictNode<K>> getPair( K key );
}
public class DictTree<K>
{     protected DictNode<K> root;        // The root of the tree.
      // Assume that word is not empty.
      public boolean containsWord( Iterator<K> word );
}
```

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

Anexo A — Interfaces do Pacote *dataStructures*

```
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 MinPriorityQueue<K extends Comparable<K>, V>
{
    boolean isEmpty( );
    int size( );
    Entry<K,V> minEntry( ) throws EmptyPriorityQueueException;
    void insert( K key, V value );
    Entry<K,V> removeMin( ) throws EmptyPriorityQueueException;
}

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 interface Dictionary<K,V>
{
    boolean isEmpty( );
    int size( );
    Iterator<Entry<K,V>> iterator( );
    // Returns the value in the dictionary whose key is the specified key;
    // or null if no such entry exists.
    V find( K key );
    // Inserts the entry (key, value) in the dictionary.
    // If the dictionary already contained an entry with the specified key,
    // returns the old value (which is replaced by the specified value);
    // otherwise, returns null.
    V insert( K key, V value );
    // Removes the entry whose key is the specified key from the dictionary
    // and returns the associated value, if such entry exists.
    // Otherwise, returns null.
    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;
}

public interface Iterator<E>
{
    boolean hasNext( );
    E next( ) throws NoSuchElementException;
    void rewind( );
}

public interface TwoWayIterator<E> extends Iterator<E>
{
    boolean hasPrevious( );
    E previous( ) throws NoSuchElementException;
    void fullForward( );
}

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

Anexo B — Classes do Pacote *dataStructures*

```
public class StackInArray<E> implements Stack<E>
{
    public static final int DEFAULT_CAPACITY = 1000;
    protected E[] array;              // Memory of the stack: an array.
    protected int top;                // Index of the element at the top of the stack.
    ..........................................
}

public class StackInList<E> implements Stack<E>
{
    protected List<E> list;           // Memory of the stack: a list.
    ..........................................
}

public class QueueInArray<E> implements Queue<E>
{
    public static final int DEFAULT_CAPACITY = 1000;
    protected E[] array;              // Memory of the queue: a circular array.
    protected int front;              // Index of the element at the front of the queue.
    protected int rear;               // Index of the element at the rear of the queue.
    protected int currentSize;        // Number of elements in the queue.
    ..........................................
}

public class QueueInList<E> implements Queue<E>
{
    protected List<E> list;           // Memory of the queue: a list.
    ..........................................
}

public class MinHeap<K extends Comparable<K>, V>
    implements MinPriorityQueue<K,V>
{
    public static final int DEFAULT_CAPACITY = 1000;
    protected Entry<K,V>[] array;     // Memory of the priority queue: an array.
    protected int currentSize;        // Number of entries in the priority queue.
    ..........................................
}
```

```
class DListNode<E>
{
    private E element;                  // Element stored in the node.
    private DListNode<E> previous;      // (Pointer to) the previous node.
    private DListNode<E> next;          // (Pointer to) the next node.

    public DListNode( ) { ... }
    public DListNode( E theElement, DListNode<E> thePrevious,
        DListNode<E> theNext ) { ... }
    public E getElement( ) { ... }
    public DListNode<E> getPrevious( ) { ... }
    public DListNode<E> getNext( ) { ... }
    public void setElement( E newElement ) { ... }
    public void setPrevious( DListNode<E> newPrevious ) { ... }
    public void setNext( DListNode<E> newNext ) { ... }
}

public class DoublyLinkedList<E> implements List<E>
{
    protected DListNode<E> head;    // Node at the head of the list.
    protected DListNode<E> tail;    // Node at the tail of the list.
    protected int currentSize;      // Number of elements in the list.

    public DoublyLinkedList( ) { ... }
    // Pre-condition: position == 0, ..., currentSize−1.
    protected DListNode<E> getNode( int position ) { ... }
    .....
}

public class DoublyLLIterator<E> implements TwoWayIterator<E>
{
    // Node with the first element in the iteration.
    protected DListNode<E> firstNode;
    // Node with the last element in the iteration.
    protected DListNode<E> lastNode;
    // Node with the next element in the iteration.
    protected DListNode<E> nextToReturn;
    // Node with the previous element in the iteration.
    protected DListNode<E> prevToReturn;
    .....
}
```

```
class BSTNode<K,V>
{
    private CEntry<K,V> entry;              // Entry stored in the node.
    private BSTNode<K,V> leftChild;         // (Pointer to) the left child.
    private BSTNode<K,V> rightChild;        // (Pointer to) the right child.

    public BSTNode( K key, V value ) { ... }
    public BSTNode( K key, V value, BSTNode<K,V> left, BSTNode<K,V> right ) { ... }
    public CEntry<K,V> getEntry( ) { ... }
    public K getKey( ) { ... }
    public V getValue( ) { ... }
    public BSTNode<K,V> getLeft( ) { ... }
    public BSTNode<K,V> getRight( ) { ... }
    public void setEntry( CEntry<K,V> newEntry ) { ... }
    public void setEntry( K newKey, V newValue ) { ... }
    public void setValue( V newValue ) { ... }
    public void setLeft( BSTNode<K,V> newLeft ) { ... }
    public void setRight( BSTNode<K,V> newRight ) { ... }
    public boolean isLeaf( ) { ... }
}

class AVLNode<K,V> extends BSTNode<K,V>
{
    // The balance factor of the tree rooted at the node, which is:
    // 'E' iff height( node.getLeft() ) = height( node.getRight() );
    // 'L' iff height( node.getLeft() ) = height( node.getRight() ) + 1;
    // 'R' iff height( node.getLeft() ) = height( node.getRight() ) − 1.
    private char balanceFactor;

    public AVLNode( K key, V value ) { ... }
    public AVLNode( K key, V value, char balance, AVLNode<K,V> left,
        AVLNode<K,V> right ) { ... }
    public char getBalance( ) { ... }
    public void setBalance( char newBalance ) { ... }
}

public class AVLTree<K extends Comparable<K>, V>
    extends BinarySearchTree<K,V>
{
    .....
}
```

```
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.

    protected static class PathStep<K,V>
    {
        public BSTNode<K,V> parent;  // The parent of the node.
        public boolean isLeftChild;  //The node is the left or the right child of parent.
        public PathStep( BSTNode<K,V> theParent, boolean toTheLeft ) {  ...  }
        public void set( BSTNode<K,V> newParent, boolean toTheLeft ) {  ...  }
    }

    // Returns the node whose key is the specified key;
    // or null if no such node exists.
    // Moreover, stores the last step of the path in lastStep.
    protected BSTNode<K,V> findNode( K key, PathStep<K,V> lastStep ) {  ...  }

    // Returns the node with the smallest key
    // in the tree rooted at the specified node.
    // Precondition: node != null.
    protected BSTNode<K,V> minNode( BSTNode<K,V> node ) {  ...  }

    // Returns the node with the largest key
    // in the tree rooted at the specified node.
    // Precondition: node != null.
    protected BSTNode<K,V> maxNode( BSTNode<K,V> node ) {  ...  }

    ..............................................
}


public abstract class HashTable<K,V> implements Dictionary<K,V>
{
    protected static final int DEFAULT_CAPACITY = 50;
    protected int currentSize;          // Number of entries in the hash table.
    protected int maxSize;              // Maximum number of entries.

    ..............................................
}


public class SepChainHashTable<K extends Comparable<K>, V>
    extends HashTable<K,V>
{
    protected Dictionary<K,V>[] table;      // The array of dictionaries.

    ..............................................
}
```