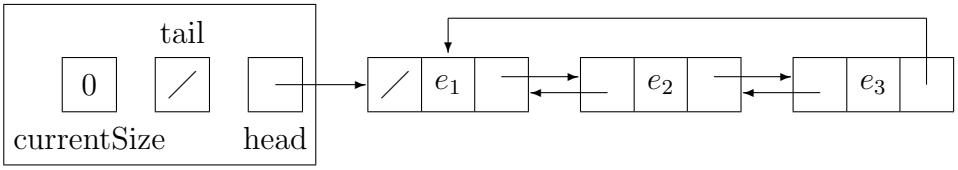
# Recurso de Algoritmos e Estruturas de Dados I

Departamento de Informática

Universidade Nova de Lisboa

21 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 três campos: o número de elementos da lista (que tem zero); a cauda da lista (que tem **null**); e o apontador do último nó da lista para o nó seguinte (que aponta para a cabeça da lista). Felizmente, todos os restantes campos estão correctos (como se ilustra na figura).




Implemente o método *restoreCoherence* (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 restoreCoherence( );
}
```

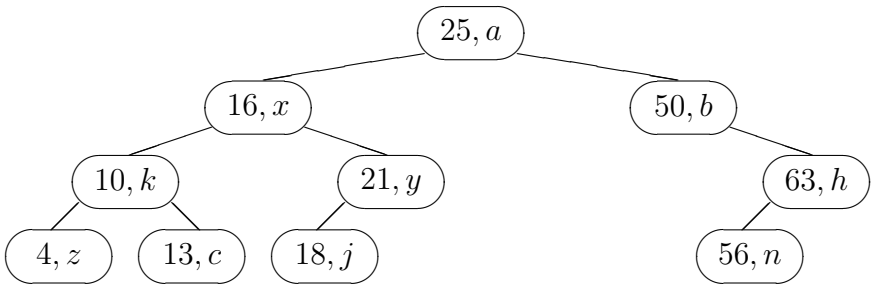
2. [3.5 valores] Considere o tipo abstracto de dados *Fila de Espera com Categorias*, de categorias do tipo R e elementos do tipo E, caracterizado pela interface *QueueWithRanks*.

```
public interface QueueWithRanks<R extends Comparable<R>, E>
{
    // Inserts the specified element, with the specified rank,
    // at the rear of the queue.
    void enqueue( R rank, E element );

    // Removes and returns the first element in the queue
    // whose rank is the specified rank.
    E dequeue( R rank ) throws NoSuchElementException;

    // Removes and returns the first element in the queue
    // whose rank is the highest rank of an element in the queue.
    E dequeue( ) throws EmptyQueueWithRanksException;
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar este tipo abstracto de dados, descreva brevemente como implementaria as três operações e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando.

3. [4.5 valores] Pretende-se acrescentar um conjunto de métodos à classe das árvores binárias de pesquisa, que permitam avaliar experimentalmente o desempenho dos algoritmos de pesquisa, inserção e remoção. Um desses métodos (chamado *minNodesVisitedInUnsuccessfulSearches*) calcula o número mínimo de nós da árvore visitados durante uma (qualquer) pesquisa sem sucesso.

   Por exemplo, para a árvore representada na figura, esse número é dois. Qualquer pesquisa cuja chave pertença ao conjunto $\{26, 27, 28 \ldots, 48, 49\}$ visita dois nós (cujas chaves são 25 e 50) e nenhuma pesquisa sem sucesso visita menos do que dois nós da árvore. Portanto, dois é o número mínimo de nós da árvore visitados durante uma pesquisa sem sucesso.




   Implemente o método *minNodesVisitedInUnsuccessfulSearches* (na classe *BinarySearchTree*), da forma mais eficiente possível. Assuma que, dadas duas quaisquer chaves $k_1$ e $k_2$ presentes na árvore, tais que $k_1 < k_2$, existe uma chave $k'$ que não se encontra na árvore e que verifica $k_1 < k' < k_2$.

```
public class BinarySearchTree<K extends Comparable<K>, V>
        implements OrderedDictionary<K,V>
{
        . . . . . .

        // Returns the minimum number of nodes visited during
        // an unsuccessful search.
        // A search is unsuccessful if the find method returns null.
        public int minNodesVisitedInUnsuccessfulSearches( );
}
```

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

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

4. [3.5 valores] A Transformada Rápida de Fourier, geralmente abreviada por *FFT* (de *Fast Fourier Transform*), tem inúmeras aplicações em áreas tão variadas como o processamento de sinal, o processamento de imagem ou a aritmética sobre polinómios. A FFT transforma um vector de números complexos noutro vector de números complexos (com a mesma dimensão). É necessário que a dimensão do vector a transformar seja uma potência de dois.

   O código que se segue, embora incompleto, esquematiza a implementação da FFT, onde *ComplexNumber* é uma interface que caracteriza um número complexo.

```
// Computes the FFT of the specified array, which has vec.length
// complex numbers (where vec.length is a power of 2).
public static void fft( ComplexNumber[] vec )
{
    permute(vec);
    fft(vec, 0, vec.length − 1);
}


private static void fft( ComplexNumber[] vec, int firstPos, int lastPos )
{
    if ( firstPos < lastPos )
    {
        int middle = ( firstPos + lastPos ) / 2;
        fft(vec, firstPos, middle);
        fft(vec, middle + 1, lastPos);
        combine(vec, firstPos, lastPos);
    }
}


// Performs the adequate permutation of the specified array.
// This method runs in O(vec.length) time,
// in the best-case and in the worst-case.
private static void permute( ComplexNumber[] vec );


// Performs the combination of two FFTs in the specified array,
// from the first specified position to the last specified position.
// This method runs in O(lastPos − firstPos + 1) time,
// in the best-case and in the worst-case.
private static void combine( ComplexNumber[] vec, int firstPos, int lastPos );
```

   Determine a complexidade temporal do método público *fft*, quando este é chamado com um vector de $n$ números complexos (onde $n$ é uma potência de dois), no melhor caso, no pior caso e no caso esperado. Justifique todos os cálculos com muita clareza.

**(Continue, porque o exame tem mais uma pergunta.)**

5. [5 valores] Pretende-se implementar um prontuário ortográfico (conjunto de palavras de uma língua) que ajude a fazer palavras-cruzadas.

   Na verdade, a ajuda só será significativa quando se conhecer a sequência de vogais de uma palavra. Para este efeito, assuma que não existem acentos nem cedilhas e que não se distinguem as maiúsculas das minúsculas, como é usual nos jogos de palavras-cruzadas.

   Para exemplificar, repare que as palavras "afasia", "alcatifa", "canalizar" e "papaia" têm a mesma sequência de vogais, que é "aaia". As palavras "afasia" e "papaia" têm comprimento seis, "alcatifa" tem comprimento oito e "canalizar" tem comprimento nove.

   A aplicação deve permitir efectuar as seguintes operações.

   (a) Inserir uma nova palavra $p$ no prontuário.

   (b) Descobrir se uma dada palavra $p$ pertence ao prontuário.

   (c) Calcular o número de palavras do prontuário com uma dada sequência $s$ de vogais.

   (d) Listar, por ordem lexicográfica, todas as palavras do prontuário com um determinado comprimento $c$ e com uma dada sequência $s$ de vogais.

   (e) De entre todas as palavras do prontuário com uma dada sequência $s$ de vogais, listar, por ordem lexicográfica, aquelas cujo comprimento for o menor possível.

   (Por exemplo, se o prontuário só tivesse as quatro palavras referidas acima e se a sequência de entrada fosse "aaia", a listagem teria as palavras "afasia" e "papaia", por esta ordem.)

   Assuma que o número de palavras do prontuário pode ser grande. Por exemplo, o ficheiro de palavras portuguesas *words*, do pacote *wportuguese* (versão 20060914-1) da distribuição Debian, tem 407 583 palavras distintas. Já o ficheiro homónimo do pacote *wspanish* (versão 1.0.19) da mesma distribuição tem apenas 86 016 palavras diferentes.

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

**(O exame terminou.)**

# 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.

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

# Anexo C — Recorrências

**Recorrência 1**

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ b\,T(n-1) + c & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

**com** $a \geq 0, \quad b \geq 1, \quad c \geq 1$ **constantes**

$$T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

**Recorrência 2**

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ b\,T(\frac{n}{c}) + f(n) & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

**com** $a \geq 0, \quad b \geq 1, \quad c > 1$ **constantes**

**e** $f(n) = O(n^k)$, **para algum** $k \geq 0$

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

**Recorrência 2 (a)**

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ b\,T(\frac{n}{2}) + d & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

**com** $a \geq 0, \quad b = 1, 2, \quad d \geq 1$ **constantes**

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

**Recorrência 2 (b)**

$$T(n) = \begin{cases} a & n = 0 \\ & \textbf{ou} \\ b\,T(\frac{n}{c}) + d\,n & n \geq 1 \end{cases} \qquad \begin{matrix} n = 1 \\ \\ n \geq 2 \end{matrix}$$

**com** $a \geq 0, \quad b \geq 1, \quad c > 1, \quad d > 0$ **constantes**

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