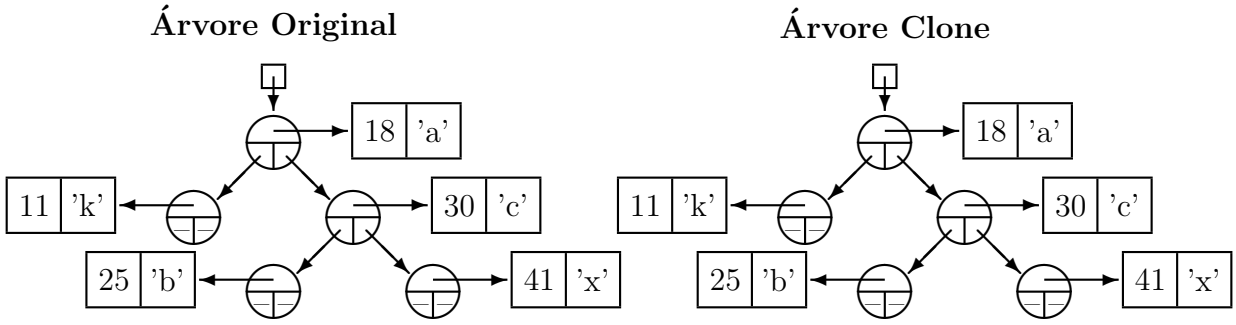
Exame de Algoritmos e Estruturas de Dados

Departamento de Informática

Universidade Nova de Lisboa

15 de Janeiro de 2009

1. [3.5 valores] Considere um novo construtor da classe *BinarySearchTree*, que constrói um clone (uma cópia independente) da árvore binária de pesquisa que é passada em argumento. Como se ilustra na figura, as duas árvores têm exactamente a mesma forma e o mesmo conteúdo, mas são independentes (não partilham posições de memória). A árvore clonada (a original) não é alterada.



Implemente esse construtor e programe todos os métodos auxiliares da classe *BinarySearchTree* de que necessitar. Apesar da classe *Object* possuir um método que retorna um clone do objecto que recebe a mensagem, neste exercício, só pode aplicar o método *clone* a instâncias da classe *EntryClass*. Para simplificar, assuma que a assinatura de *clone* é a especificada no anexo.

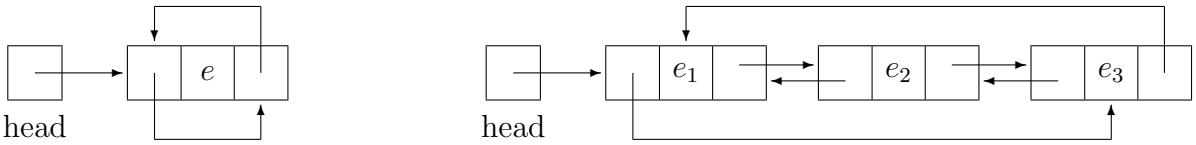
```
public class BinarySearchTree<K extends Comparable<K>, V>
    implements OrderedDictionary<K,V>
{
    // The root of the tree.
    protected BSTNode<K,V> root;

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

    .....

    // Creates a new binary search tree, which is a clone of the specified tree.
    // The two trees are independent.
    public BinarySearchTree( BinarySearchTree tree );
}
```

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 *clone*, quando aplicado a uma instância da classe *EntryClass*, é sempre constante.

2. [4.5 valores] Uma **lista circular duplamente ligada com cabeça** é uma estrutura de dados muito semelhante a uma lista duplamente ligada com cabeça. A única diferença é que os nós da lista circular nunca têm *null*. Mais precisamente:

   - o anterior do primeiro nó da lista aponta para o último nó da lista; e
   - o seguinte do último nó da lista aponta para o primeiro nó da lista.

   A figura ilustra a forma de duas listas circulares duplamente ligadas com cabeça, uma com um elemento e a outra com três elementos.



   Relembre então a classe *DListNode*, dos nós das listas duplamente ligadas de elementos do tipo E, e considere a classe *CircularDoublyLinkedList*, das listas circulares duplamente ligadas com cabeça de elementos do tipo E. As duas classes pertencem ao pacote *dataStructures*.

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

   public DListNode( E theElem );
   public DListNode( E theElem, DListNode<E> thePrev, 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 CircularDoublyLinkedList<E>
{
   protected DListNode<E> head;     // Node at the head of the list.
   ..........................................

   // Inserts the specified element at the last position in the list.
   public void addLast( E element );

   // Removes and returns the element at the last position in the list.
   public E removeLast( ) throws EmptyListException;
}
```

   Implemente os métodos *addLast* e *removeLast* (na classe *CircularDoublyLinkedList*) e programe todos os métodos auxiliares desta classe de que necessitar. Note que a classe só tem uma variável de instância, chamada *head*. Calcule as complexidades temporais dos seus dois algoritmos, no melhor caso, no pior caso e no caso esperado, justificando.

3. [3.5 valores] A função *funRec*, definida para números inteiros não negativos, tem complexidade exponencial. Apresente um algoritmo polinomial que calcule o mesmo valor, aplicando a técnica da função-memória ao algoritmo dado.

```
public static double funRec( int n ) throws NegativeNumberException
{
    if ( n < 0 )
        throw new NegativeNumberException();

    if ( n < 3 )
        return n;
    else
        return Math.sqrt(n) + funRec(n − 1) * funRec(n − 3);
}
```

Determine a complexidade temporal e espacial do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

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

4. [3.5 valores] Um domínio D diz-se *auto-representável* se existir uma função rep : $D \longrightarrow D$ que atribui, a cada elemento $x \in D$, um elemento rep$(x) \in D$, chamado *o representante* de x. Os elementos $x, y \in D$ *pertencem à mesma família*, se rep$(x) =$ rep(y). Qualquer conjunto $C \subseteq D$ é um *subconjunto auto-representável*.

   Suponha que, em termos de programação, os objectos representáveis por elementos do tipo E são caracterizados pela interface *Representable* e considere o tipo abstracto de dados *Subconjunto Auto-Representável* de elementos do tipo E, definido pela interface *SelfRepresentableSubset*.

```
public interface Representable<E>
{
    // Returns the representative of this object.
    E representative( );
}

public interface SelfRepresentableSubset<E extends Representable<E>>
{
    // Returns the number of elements in the set.
    int size( );

    // Returns true iff the specified element belongs to the set.
    boolean contains( E element );

    // Inserts the specified element in the set.
    void insert( E element ) throws ExistingElementException;

    // Removes the specified element from the set.
    void remove( E element ) throws NoSuchElementException;

    // Returns an iterator of all elements in the set
    // that belong to the same family of the specified element.
    Iterator<E> sameFamily( E element );
}
```

   Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar o TAD *Subconjunto Auto-Representável*, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no caso esperado, justificando. Pode assumir que os elementos do domínio são comparáveis entre si (E extends Comparable<E>) e que a complexidade do método *representative* é sempre constante.

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

5. [5 valores] Pretende-se construir uma aplicação para gerir a lista telefónica (páginas brancas) de uma região que terá, no máximo, 100 000 *números de telefone*. Cada número de telefone está associado a um *nome*, a uma *morada* e, possivelmente, a uma *subdivisão*.

   As subdivisões permitem agrupar números de telefone associados ao mesmo assinante. Por exemplo, se a Caixa Geral de Depósitos tiver três agências e o Dr. Luís Lopes Matias tiver um telefone na habitação e outro no consultório, o assinante Caixa Geral de Depósitos terá três subdivisões e o assinante Dr. Luís Lopes Matias terá duas.

   Os assinantes são então classificados em dois tipos. Um *assinante simples* tem um nome, uma morada e um número de telefone. Um *assinante estruturado* tem um nome e dois ou mais "triplos", em que cada triplo é composto por uma subdivisão, uma morada e um número de telefone. Os números de telefone são únicos, mas pode haver dois assinantes com o mesmo nome ou com moradas iguais.

   O exemplo seguinte ilustra estes conceitos.

Antunes, Josefa Ernestina – 12, 2º D, Rua do Sol, 9999-999 Aviais	652 856 745
Caixa Geral de Depósitos	
Agência de Almada – 2, Largo 26 de Abril, 7654-767 Almada	212 953 456
Agência de FCT/UNL – Quinta da Torre, 2829-516 Caparica	212 945 555
Agência de Setúbal – 3, Rua do Bocage, 2466-001 Setúbal	216 548 179
Matias, Luís Lopes	
Consultório – 25, Rua do Lá Vai Um, 8956-123 Caixinhas	553 457 880
Residência – 25, Rua do Lá Vão Dois, 8956-123 Caixinhas	553 457 889
Matias, Luís Lopes – 12, 2º D, Rua do Sol, 9999-999 Aviais	652 777 000

   A aplicação deve permitir efectuar as seguintes operações.

   (a) Inserir um assinante simples, dando todos os seus dados (nome, morada e número de telefone).
   A operação não será efectuada se o número de telefone existir.

   (b) Inserir um assinante estruturado, dando todos os seus dados (nome e triplos com subdivisão, morada e número de telefone).
   A operação não será efectuada se algum dos seus números de telefone existir.

   (c) Remover um assinante, dando um qualquer dos seus números de telefone.

   (d) Obter todos os dados de um assinante, dando um qualquer dos seus números de telefone.

   (e) Gerar a lista telefónica, ou seja, listar todos os assinantes e todos os seus dados, como se ilustrou no exemplo. A listagem deve aparecer por ordem alfabética de nome. Em caso de igualdade no nome, a ordem é arbitrária. A listagem das subdivisões de um assinante deve estar por ordem alfabética de subdivisão.

   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.

Anexo

```
class EntryClass<K,V> implements Entry<K,V>
{
    private K key;                          // Key stored in the entry.
    private V value;                        // Value stored in the entry.

    public EntryClass( K theKey, V theValue );

    public K getKey( );

    public V getValue( );

    public void setKey( K newKey );

    public void setValue( V newValue );

    // Creates and returns a clone of this entry.
    public EntryClass<K,V> clone( );
}


class BSTNode<K,V>
{
    private EntryClass<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 BSTNode( EntryClass<K,V> theEntry );

    public EntryClass<K,V> getEntry( );

    public K getKey( );

    public V getValue( );

    public BSTNode<K,V> getLeft( );

    public BSTNode<K,V> getRight( );

    public void setEntry( EntryClass<K,V> newEntry );

    public void setEntry( K newKey, V newValue );

    public void setKey( K newKey );

    public void setValue( V newValue );

    public void setLeft( BSTNode<K,V> newLeft );

    public void setRight( BSTNode<K,V> newRight );

    public boolean isLeaf( );
}
```