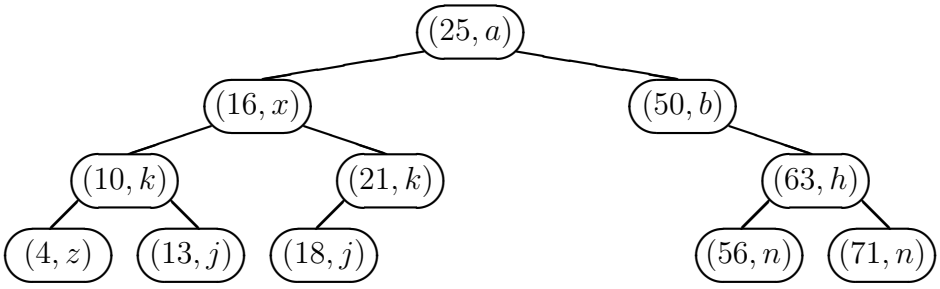
Recurso de Algoritmos e Estruturas de Dados

Departamento de Informática

Universidade Nova de Lisboa

4 de Fevereiro de 2010

1. [3.5 valores] O método *statistics*, da classe *BinarySearchTree*, retorna um vector com três inteiros: a posição zero do vector tem o número de folhas da árvore; a posição um do vector tem o número de nós da árvore com exactamente um filho; e a posição dois do vector tem o número de nós da árvore com dois filhos. No caso da árvore esquematizada na figura, esses inteiros seriam, respectivamente, 5, 2 e 4. Repare que, se a árvore for vazia, os três números são zero.




```
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;

    .....

    public int[] statistics( );
}
```

Implemente o método *statistics*, programando 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.

2. [3.5 valores] Considere o tipo abstracto de dados *Pilha com Máximo*, de elementos do tipo E, caracterizado pela interface *StackWithMax*.

```
public interface StackWithMax<E extends Comparable<E>>
    extends Stack<E>
{
    // Returns the greatest element in the stack.
    E maximum( ) throws EmptyStackException;
}

public interface Stack<E>
{
    // Returns true iff the stack contains no elements.
    boolean isEmpty( );

    // Returns the number of elements in the stack.
    int size( );

    // Returns the element at the top of the stack.
    E top( ) throws EmptyStackException;

    // Inserts the specified element onto the top of the stack.
    void push( E element );

    // Removes and returns the element at the top of the stack.
    E pop( ) throws EmptyStackException;
}
```

Explicite **detalhadamente as estruturas de dados** mais adequadas para implementar a interface *StackWithMax*, descreva brevemente como implementaria as seis operações e calcule as suas complexidades temporais, no caso esperado, justificando. Assuma que a complexidade temporal do método *compareTo*, quando aplicado a instâncias do tipo E, é sempre constante.

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

3. [3.5 valores] A função *noName*, 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 long noName( int n ) throws NegativeNumberException
{
    if ( n < 0 )
        throw new NegativeNumberException();

    return noNameRec(n);
}


private static long noNameRec( int n )
{
    if ( n == 0 || n == 1 )
        return 2 * n + 1;
    else
    {
        long prod = 1;
        for ( int i = 0;   i < n / 2;   i++ )
            prod = prod * ( noNameRec(n - i - 1) % noNameRec(i) + 1 );
        return prod;
    }
}
```

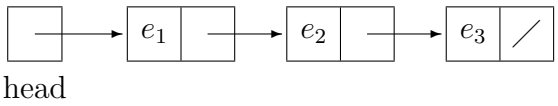
Determine a complexidade temporal e espacial do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

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

4. [4.5 valores] Considere a classe *OrderedSimplyLinkedList*, das listas simplesmente ligadas com cabeça, ordenadas e sem elementos repetidos, de elementos do tipo E. Esta classe tem apenas um atributo (*head*) que guarda o primeiro nó da lista.

   Como a lista é:

   - *simplesmente ligada*, cada nó da lista tem apenas o elemento E e o apontador para o nó seguinte;
   - *ordenada e sem elementos repetidos*, os elementos estão por ordem estritamente crescente (ou seja, $e_1 < e_2 < e_3$, no exemplo ilustrado na figura).



   A classe *SListNode*, dos nós das listas simplesmente ligadas de elementos do tipo E, e a classe *OrderedSimplyLinkedList* pertencem ao pacote *dataStructures*.

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

    public SListNode( E theElement );
    public SListNode( E theElement, SListNode<E> theNext );
    public E getElement( );
    public SListNode<E> getNext( );
    public void setElement( E newElement );
    public void setNext( SListNode<E> newNext );
}

public class OrderedSimplyLinkedList<E extends Comparable<E>>
{
    protected SListNode<E> head;        // Node at the head of the list.
    ..........................................

    // Inserts the specified element in the linked list
    // if it is not already present.
    public void insert( E element ) throws RepeatedElementException;
}
```

   Implemente o método *insert*, programando todos os métodos auxiliares da classe *OrderedSimplyLinkedList* 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 E, é sempre constante.

5. [5 valores] Pretende-se construir uma aplicação para gerir um parque virtual de diversões. Para visitar o parque, é necessário realizar uma *inscrição* com um *endereço de e-mail*, que identificará sem ambiguidade o futuro visitante. Considera-se que um *visitante existe* quando o respectivo endereço foi dado numa inscrição.

   No parque, os visitantes podem participar simultaneamente em diferentes *actividades* e podem, numa mesma *data* (dia, mês e ano), participar mais do que uma vez em cada actividade. A participação na actividade pode durar apenas alguns segundos ou vários dias. Cada *actividade* tem um *nome* (único, que a identifica), é de um determinado *tipo* e tem capacidade ilimitada.

   A aplicação permite gerir as inscrições, as actividades e as participações dos visitantes nas actividades do parque virtual, através das seguintes operações.

   (a) Inscrever um visitante no parque, dando o seu endereço de *e-mail*.
   A operação só será efectuada se não existir um visitante com esse endereço.

   (b) Adicionar uma actividade ao parque, dando o nome e o tipo da actividade.
   A operação só será efectuada se não existir uma actividade com esse nome.

   (c) Registar o início da participação (na data corrente) de um visitante numa actividade do parque, dando o endereço do visitante e o nome da actividade. Assuma que a data corrente será obtida chamando uma função da biblioteca.
   A operação só será efectuada se o visitante existir, mas não estiver a participar nessa actividade, e a actividade existir.

   (d) Registar o fim da participação de um visitante numa actividade do parque, dando o endereço do visitante e o nome da actividade.
   A operação só será efectuada se o visitante estiver a participar nessa actividade.

   (e) Listar as participações (em actividades) de um dado visitante, dando o seu endereço.
   Cada linha da listagem deve ter o *tipo* e o *nome* da actividade, a *data* de início e o *número de participações* (concluídas ou em curso) do visitante nessa actividade que tiveram início nessa data.
   A listagem deve estar ordenada alfabeticamente por tipo de actividade, dentro de cada tipo de actividade, por nome de actividade e, para cada actividade, decrescentemente por data.
   A operação só será efectuada se o visitante existir.

   (f) Listar os números totais de participações numa actividade, dando o nome da actividade.
   Cada linha da listagem deve ter a *data* de início e o *número de participações* (concluídas ou em curso) na actividade que tiveram início nessa data.
   A listagem deve estar ordenada decrescentemente por data.
   A operação só será efectuada se a actividade existir.

   Assuma que há cerca de meio milhão de visitantes e cerca de 2 000 actividades. A probabilidade de um visitante participar em mais de cem actividades simultaneamente é inferior a 10^{-7}.

   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.