Algoritmos e Estruturas de Dados 2018/19
Primeiro Teste – 29 de outubro de 2018
Departamento de Informática, Universidade Nova de Lisboa

Atenção:

- Os Anexos ao teste poderão ser-lhe úteis.
- Leia todas as indicações no seu caderno de teste.

Grupo I – Análise de algoritmos

Considere o método potIt apresentado abaixo.

```
//Requires: n > 0
public static double potIt(double x, long n){
   double result = 1;
   double currentX = x;
   long currentN = n;
   for (currentN = n; currentN > 0; currentN = currentN / 2){
      if ( currentN % 2 != 0)
         result = result * currentX;
      currentX = currentX * currentX;
   }
   return result;
}
```

Determine a complexidade temporal do método potIt, no melhor caso, no pior caso, e no caso esperado, **justificando**.

Grupo II – Programação em AED

1. Implemente um método recursivo que, dado um vetor de números inteiros completamente preenchido, devolve o número de ocorrências de um determinado número inteiro, passado como parâmetro ao método.
   numberOfOccurences (apresentado abaixo) deve constituir o acesso público ao método, podendo ser necessário implementar métodos auxiliares para concretizar a recursão. Determine ainda a complexidade temporal do método numberOfOccurences no melhor caso, no pior caso e no caso esperado, **justificando**.

   Nota: Implemente a solução em código Java. Apenas as soluções recursivas serão avaliadas.

```
public static int numberOfOccurences(int aim, int[] array) {
        ...
}
```

(A pergunta 2 do Grupo II está na página 2)

2. Considere a classe de lista duplamente ligada `DoublyLinkedList` apresentada nas aulas, concentrando-se no seu método público **`remove()`**. Este método recebe, como parâmetro, um número inteiro que contém a posição de um elemento do tipo genérico **E** a remover da lista, e remove o nó da lista que contém este mesmo elemento, se a posição recebida for válida.
   Implemente este método com a melhor complexidade temporal possível e tendo em conta todas as situações de exceção. Na sua implementação deverá reutilizar outros métodos que façam parte da classe em causa, tais como o **`removeFirst()`, `removeLast()`, `getNode()` e `removeMiddleNode()`, não sendo necessário implementar os mesmos (consulte o Anexo B para conhecer as assinaturas dos métodos em questão)**. Deve ainda determinar a complexidade temporal do método no melhor caso, no pior caso e no caso esperado, **justificando**.

```
package dataStructures;

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

   // Node at the head of the list.
   protected DListNode<E> head;
   // Node at the tail of the list.
   protected DListNode<E> tail;
   // Number of elements in the list.
   protected int currentSize;

. . .
   public E remove( int position ) throws InvalidPositionException{
   ...
   }

}
```

(O Grupo III está na página 3)

Grupo III – Tipos Abstratos de Dados e Estruturas de Dados

Neste grupo deve analisar o problema apresentado e conceber uma solução para o problema completo de acordo com os requisitos descritos. Deverá depois responder a cada uma das questões, de acordo com a solução que concebeu.
Nas suas respostas às questões 4 e 5, não deve desenvolver código em java, apenas fazer uma descrição da implementação das operações pedidas, de acordo com a sua escolha de estruturas de dados e variáveis de instância.

O Tipo Abstrato de Dados (TAD) SharingBike define as operações associadas à partilha de **uma única** bicicleta, num sistema gratuito de partilha de bicicletas, na cidade de Beja. Será necessário manter informação sobre a bicicleta (matrícula e ano de fabricação) assim, como dados relativos às várias partilhas da mesma bicicleta. Denominamos de partilhas os períodos, em minutos, durante os quais um cidadão da cidade, ou um turista, se movimenta pela cidade com a bicicleta. O sistema é gratuito, mas a autarquia pretende manter os períodos de partilha dentro dos 30 minutos, pelo que o acumulado de atrasos na entrega de cada bicicleta (para além destes períodos de 30 minutos) deve ser tido em conta. Também é importante conhecer a média (em minutos) dos períodos de partilha, para cada bicicleta. Finalmente, deve ser possível listar todos os períodos (em minutos) de partilha da bicicleta, ordenados cronologicamente, do mais recente para o mais antigo.

Apresenta-se abaixo o interface SharingBike:

```
import dataStructures.Iterator;
public interface SharingBike {

  //Devolve a matrícula da bicicleta.
  String getLicense();

  //Devolve o ano de fabricação da bicicleta.
  int getManufactureYear();

  //Adiciona um período de partilha da bicicleta (em minutos).
  void addShare(int minutes);

  //Devolve iterador (em minutos) dos períodos de partilha da bicicleta.
  //A iteração executada a partir deste iterador deve apresentar
  //os períodos de partilha da bicicleta por ordem cronológica,
  //iniciando na mais recente e terminando na mais antiga.
  //Requires: a bicicleta terá de ter sido partilhada pelo menos uma vez.
  Iterator<Integer> listSharing() throws NoSharingYetException;

  //Devolve o valor acumulado (em minutos) de atrasos (para além dos 30 minutos)
  //na entrega da bicicleta.
  int getAccumulatedDelay();

  //Devolve a média (em minutos) dos períodos de partilha da bicicleta.
  int getAverageSharing();
}
```

(O Grupo III continua na página 4)

Relativamente ao interface apresentado, reflita sobre a melhor implementação para o mesmo e responda às seguintes questões **(separadamente)**:

1. Proponha as variáveis de instância necessárias e respetivos tipos para apoiar a implementação dos métodos `getLicense()` e `getManufactureYear()`.
2. Proponha uma Estrutura de Dados para apoiar a implementação dos métodos `addShare()` e `listSharing()`. A sua resposta deve incluir:
   a) **Tipo Abstrato de Dados (TAD) genérico;**
   b) **Estrutura de Dados (ED) genérica que deverá ser usada para implementar o TAD;**
   c) **Tipo de dados (do problema) a guardar dentro da ED (Tipo do Elemento (`E`); ou tipos associados ao par `Entry<K,V>`) e respetivo significado. <u>Deve sempre justificar esta escolha</u>.**

   NOTA: Tenha em conta que, quando a escolha recai sobre um dicionário, será necessário escolher o tipo da Chave (`K`) e o tipo do Valor associado (`V`).
3. Proponha ainda, se achar conveniente, variáveis de instância adicionais, para apoiar a implementação dos métodos `getAccumulatedDelay()` e `getAverageSharing()`.
4. Considerando as EDs e variáveis de instância propostas em 1, 2 e 3, descreva brevemente como implementaria a operação `addShare()`, considerando que:
   - a operação `listSharing()` deverá devolver um iterador da ED proposta em 2;
   - a execução das operações `getAccumulatedDelay()` e `getAverageSharing()` deve ter complexidade mínima possível, em todos os casos.

   Estude ainda a complexidade temporal da operação `addShare()` no melhor caso, no pior caso e no caso esperado, **justificando**.
5. Com base nas EDs e variáveis de instância propostas em 1, 2 e 3, descreva brevemente como implementaria a operação `getAccumulatedDelay()`. Estude ainda a complexidade temporal desta operação no melhor caso, no pior caso e no caso esperado, **justificando**.
6. Com base nas EDs e variáveis de instância propostas em 1, 2 e 3, descreva brevemente como implementaria a operação `getAverageSharing()`. Estude ainda a complexidade temporal desta operação no melhor caso, no pior caso e no caso esperado, **justificando**.

(Os anexos ao teste estão nas páginas seguintes)

Anexo A - Recorrências

Recorrência 1

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

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

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

Recorrência 2a)

$$T(n) = \begin{cases} a & n = 0 \\ & \text{ou} \\ bT(\frac{n}{2}) + O(1) & n \geq 1 \end{cases} \qquad \begin{matrix} \\ \\ n = 1 \\ \\ n \geq 2 \end{matrix}$$

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

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

Recorrência 2b)

$$T(n) = \begin{cases} a & n = 0 \\ & \text{ou} \\ bT(\frac{n}{c}) + O(n) & n \geq 1 \end{cases} \qquad \begin{matrix} \\ \\ n = 1 \\ \\ n \geq 2 \end{matrix}$$

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

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

Anexo B – Interfaces e Classes de Apoio

```
public interface Comparable<T>{
   int compareTo( T object );
}
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 Iterator<E> {
   boolean hasNext( );
   E next( ) throws NoSuchElementException;
   void rewind( );
}
public interface List<E> {
   boolean isEmpty( );
   int size( );
   Iterator<E> iterator( );
   E getFirst( ) throws EmptyListException;
   E getLast( ) throws EmptyListException;
   E get( int position )
      throws InvalidPositionException;
   int find( E element );
   void addFirst( E element );
   void addLast( E element );
   void add( int position, E element )
      throws InvalidPositionException;
   E removeFirst( ) throws EmptyListException;
   E removeLast( ) throws EmptyListException;
   E remove( int position )
      throws InvalidPositionException;
   boolean remove( E element );
}
public interface Entry<K,V>{
   K getKey( );
   V getValue( );
}
public interface Dictionary<K,V>{
   boolean isEmpty( );
   int size( );
   Iterator<Entry<K,V>> iterator( );
   V find( K key );
   V insert( K key, V value );
   V remove( K key );
}
```

```
class DListNode<E> implements Serializable {
   public DListNode( E theElement, DListNode<E>
            thePrevious, DListNode<E> theNext );
   public DListNode( E theElement );
   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> {
   public boolean isEmpty( );
   public int size( );
   public Iterator<E> iterator( );
   public E getFirst( )
      throws EmptyListException;
   public E getLast( )
      throws EmptyListException;
   protected DListNode<E> getNode
                              ( int position );
   public E get( int position )
      throws InvalidPositionException;
   public int find( E element );
   public void addFirst( E element );
   public void addLast( E element );
   protected void addMiddle
                  ( int position, E element );
   public void add( int position, E element )
      throws InvalidPositionException;
   protected void removeFirstNode( );
   public E removeFirst( )
      throws EmptyListException;
   protected void removeLastNode( );
   public E removeLast( )
      throws EmptyListException;
   protected void removeMiddleNode
                        ( DListNode<E> node );
   public E remove( int position )
      throws InvalidPositionException;
   protected DListNode<E> findNode( E element );
   public boolean remove( E element );
   public void append
                  ( DoublyLinkedList<E> list );
}
```