

Algoritmos e Estruturas de Dados
2014/15
Perguntas Tipo
Departamento de Informática, Universidade Nova de Lisboa

Atenção: Os Anexos ao teste poderão ser-lhe útil.

1. No problema de Josephus, há n pessoas, numeradas de 1 a n , dispostas em círculo. Começando na pessoa 1, percorre-se o círculo (as vezes necessárias) eliminando pessoas, até ficar apenas uma que se salva. A pessoa a ser eliminada é sempre a segunda que se encontra no percurso circular (que não tem descontinuidades).

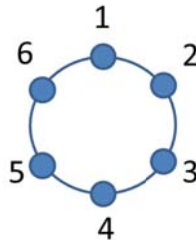


Figura 1

Por exemplo, se $n = 6$, na primeira volta pelo círculo, eliminam-se as pessoas 2, 4 e 6. Depois, (encontra-se a pessoa 1 e) é eliminada a 3, (encontra-se a pessoa 5 e) é eliminada a 1. Neste caso, salva-se a pessoa 5.

A função `josephus` calcula quem se salva, quando o número de pessoas é n .

```
public static int josephus( int n )  
    throws NonPositiveNumberException {  
    if ( n <= 0 )  
        throw new NonPositiveNumberException();  
    return josephRec(n);  
}  
  
private static int josephRec( int n ) {  
    if ( n == 1 )  
        return 1;  
    else if ( n % 2 == 0 )  
        return 2 * josephRec(n / 2) - 1;  
    else  
        return 2 * josephRec(n / 2) + 1;  
}
```

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

2. Considere a Classe de lista duplamente ligada `DoublyLinkedList` apresentada nas aulas, concentrando-se no seu método de remoção à cabeça, listado abaixo. Na sua folha de teste implemente o método auxiliar `removeFirstNode()` que falta para complementar o método público. Inclua comentários para cada linha, a explicar o objetivo de cada instrução do método.

```
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;
```

```
    . . .
```

```
    // Removes and returns the element at the first position in the list.
```

```
    public E removeFirst( ) throws EmptyListException
```

```
    {
```

```
        if ( this.isEmpty() )
```

```
            throw new EmptyListException();
```

```
        E element = head.getElement();
```

```
        this.removeFirstNode();
```

```
        return element;
```

```
    }
```

```
}
```

3. Implemente o método `sumOfDigits` que recebe, como parâmetro, um número inteiro n e que calcula, de forma recursiva, a soma de todos os dígitos que compõem o mesmo número. Por exemplo, para $n = 12345$, o método pedido devolve 15. Determine a complexidade temporal do método `sumOfDigits`, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.
4. Aproximam-se as eleições para o conselho de gestão do clube de Bairro "Os amigos da Troika". É preciso organizar a votação de forma a podermos saber o número de votos atribuídos a cada sócio. O clube valoriza os sócios mais antigos atribuindo, a cada sócio, tantos votos quanto o número de anos de filiação do mesmo no clube e adicionando um (1) a esse número. Relativamente a cada sócio sabemos sempre o seu nome e o ano em que se filiou no clube. Se soubermos qual o ano em que estamos, é simples saber o número de votos de um sócio. Por exemplo, o "Ti Manel" entrou para o clube em 1984. Isto significa que em 2014, ele terá direito a 31 votos. O "Toni Rodinhas" só tem direito a 11 votos porque entrou para o Clube em 2004. Já a bebé dele, a "Sissi" que nasceu no

primeiro dia de 2014, no feriado, já é sócia, com direito a 1 voto. É preciso desenvolver um sistema de suporte às eleições do clube. Pensámos num tipo abstrato de dados para suportar a resolução do problema, e pedimos-lhe que pense na melhor implementação para o mesmo. O TAD em causa é apresentado no interface Java abaixo:

```
public interface TroikaFriends {

    //Devolve o ano corrente
    int currentYear();

    //Adiciona um novo sócio ao clube, verificando que não existe já um
    //sócio com o mesmo nome. O ano de filiação do sócio é o ano em curso
    void addPartner(String partnerName) throws ExistingPartnerException;

    //Adiciona um novo sócio ao clube, verificando que não existe já um
    //sócio com o mesmo nome. O ano de filiação do sócio é registryYear
    //Requires: registryYear <= currentYear()
    void addPartner(String partnerName, int registryYear)
        throws ExistingPartnerException, InvalidYearException;

    //Atualiza o ano em curso para o ano seguinte ao corrente
    void newYear();

    //Devolve o número de votos atribuídos ao sócio partnerName
    int partnerVotes(String partnerName)
        throws NonExistingPartnerException;

    //Devolve o número de sócios do clube
    int numberOfPartners();

    //Devolve a soma de todos os votos atribuídos ao total de sócios
    //do clube
    int numberOfCurrentVotes();
}
```

Parta do princípio de que o construtor da classe que implementar o interface receberá, como parâmetro, o ano corrente.

Baseando-se na descrição apresentada e no interface fornecido, explicita detalhadamente as estruturas de dados mais adequadas para implementar o mesmo, descreva brevemente como implementaria as sete operações e calcule as suas complexidades temporais, no caso esperado, justificando.

5. Considere o tipo abstracto de dados Fila com Dois Níveis, de elementos do tipo E, caracterizado pela interface TwoLevelQueue.

```
public interface TwoLevelQueue<E> {  
  
    // Inserts the specified element at the rear of the queue,  
    // classifying it as a level one element.  
    void enqueueLevel1( E element );  
  
    // Inserts the specified element at the rear of the queue,  
    // classifying it as a level two element.  
    void enqueueLevel2( E element );  
  
    // Removes and returns the element at the front of the queue.  
    E dequeue( ) throws EmptyQueueException;  
  
    // Returns an iterator of the level one elements in the queue  
    // (in proper sequence).  
    Iterator<E> iteratorLevel1( );  
  
    // Returns an iterator of the level two elements in the queue  
    // (in proper sequence).  
    Iterator<E> iteratorLevel2( );  
}
```

Explicitamente detalhadamente as estruturas de dados mais adequadas para implementar a interface TwoLevelQueue, descreva brevemente como implementaria as cinco operações e calcule as suas complexidades temporais, no melhor caso, no pior caso e no caso esperado, justificando.

Anexo A - Recorrências

Recorrência 1

$$T(n) = \begin{cases} a & n = 0 & n = 1 \\ bT(n-1) + c & n \geq 1 & n \geq 2 \end{cases} \quad \text{ou} \quad T(n) = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

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

Recorrência 2a)

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

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

Recorrência 2b)

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

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 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;  
  
    //interface List<E> continua na página 7.
```

```

//continuação do interface List<E> (página 6).

E removeFirst( ) throws EmptyListException;

E removeLast( ) throws EmptyListException;

E remove( int position ) throws InvalidPositionException;

boolean remove( E element );

}

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;

    //DoublyLinkedList<E> continua na página 8.

```

```

//continuação da DoublyLinkedList<E> (página 7).

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 );
}

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

```

```
public interface Comparable<T>{  
    int compareTo( T object );  
}  
  
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 );  
}
```