Algoritmos e Estruturas de Dados
Exercícios das Aulas Práticas

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Ano Letivo 2023/2024

1. Uma instituição bancária necessita de uma nova aplicação de gestão de cartões de crédito. Essa aplicação deve permitir registar electronicamente os movimentos (de débito ou de crédito) de um cartão, fornecer a lista dos movimentos (não saldados) de um cartão, e obter e liquidar o saldo de um cartão.

   Os dados correspondentes a cada *cartão* de crédito são o *número* do cartão e o *nome* do detentor do cartão. É possível registar *movimentos* de cartões. Um *movimento* de um cartão é identificado pela *data do movimento* (até aos segundos) e possui um *tipo* (débito ou crédito) e um *montante* (o valor do débito ou do crédito). Considera-se que o *saldo* de um cartão é a soma dos montantes dos movimentos do cartão presentes no sistema. *Liquidar o saldo* de um cartão é pagar ou receber esse valor, removendo os respectivos movimentos do sistema e anulando o saldo do cartão.

   A aplicação deve permitir efectuar as seguintes operações.

   (a) Obter o número de um novo cartão, com o nome do detentor. Assuma que o número (único) é gerado internamente pelo sistema, por um processo que iremos ignorar neste contexto.

   (b) Obter o nome do detentor de um cartão, dado o seu número.

   (c) Inserir um novo movimento, dados o número do cartão, e a data, o tipo e o montante do movimento.

   (d) Listar a data, o tipo e o montante dos movimentos de um cartão, dado o seu número. A listagem deve estar ordenada cronologicamente por data.

   (e) Obter o saldo de um cartão, dado o seu número.

   (f) Liquidar o saldo de um cartão, dado o seu número.

   Elabore o **Modelo dos Tipos Abstractos de Dados (TAD)** deste problema. Esse modelo deve conter a seguinte informação.

   - Uma definição dos tipos abstractos de dados necessários para resolver o problema, incluindo a lista e descrição de todas as operações públicas a disponibilizar pelos TAD.
   - Uma enumeração dos atributos (e dos respectivos tipos de dados) das entidades cujo tipo foi definido no ponto anterior.

2. Elabore o **Modelo das Estruturas de Dados** do problema introduzido na pergunta 1. Esse modelo deve conter a seguinte informação:

   (a) A definição completa das estruturas de dados, indicando o que são e que informação guardam.

   (b) Para cada uma das operações do enunciado, uma descrição do comportamento do programa em termos das operações efectuadas sobre as estruturas de dados.

   (c) Uma justificação para terem escolhido aquelas estruturas de dados e não outras.

   (d) O estudo das complexidades temporais das operações do enunciado, no melhor caso, no pior caso e no caso esperado.

   (e) O estudo da complexidade espacial da solução proposta.

   Neste exercício, assuma que a única estrutura de dados base disponível (para além dos tipos básicos) é o vector.

3. Pedimos-lhe que considere a implementação de um Projeto de Revista Online, pensando no Sistema de Gestão de conteúdos. Este sistema irá gerir os autores dos textos publicados na revista, assim como a informação associada aos próprios textos. Os textos estarão disponíveis num repositório pelo que, para se ler um determinado texto, utilizamos o seu URL. Assim, o sistema irá fazer a gestão dos Autores dos textos, cuja informação inclui Código de autor, Nome de autor, Email e Número de telefone. Relativamente aos *Textos*, estes têm um Código de Texto, Título, um Autor e um URL. Assuma que não existem nomes de Autores repetidos e que os títulos dos textos de um autor são únicos. O Sistema deverá implementar as seguintes operações:

   (a) *Inserir um Autor*, recebendo a seguinte informação: Código de Autor, Nome de Autor, Email e Número de telefone. A inserção só tem sucesso se o código do autor ainda não existir no sistema;

   (b) *Inserir um Texto*, que é composto de: Código de Texto, Título, Código de Autor do Texto e URL. A Inserção só tem sucesso se o código de autor já existir no sistema e se o código do texto não existir no sistema;

   (c) *Aceder ao URL de um texto*, dando o código do texto. Esta operação só tem sucesso se o código do texto existir no Sistema;

   (d) *Remover Texto*, dando o código do texto. Esta operação só tem sucesso se o código do texto existir no sistema;

   (e) *Listar os Títulos de todos os textos de um determinado autor*, dando o **nome do autor**. A listagem deve ser apresentada por ordem alfabética do título do texto. Esta operação só tem sucesso se o autor em causa fizer parte do sistema.

   Elabore o **Modelo dos Tipos Abstractos de Dados (TAD)** deste problema. Esse modelo deve conter a seguinte informação:

   - Uma definição dos tipos abstractos de dados necessários para resolver o problema, incluindo a lista e descrição de todas as operações públicas a disponibilizar pelos TAD.
   - Uma enumeração dos atributos (e dos respectivos tipos de dados) das entidades cujo tipo foi definido no ponto anterior.

4. Elabore o **Modelo das Estruturas de Dados** do problema introduzido na pergunta 3. Esse modelo deve conter a seguinte informação.

   (a) A definição completa das estruturas de dados, indicando o que são e que informação guardam.

   (b) Para cada uma das operações do enunciado, uma descrição do comportamento do programa em termos das operações efectuadas sobre as estruturas de dados.

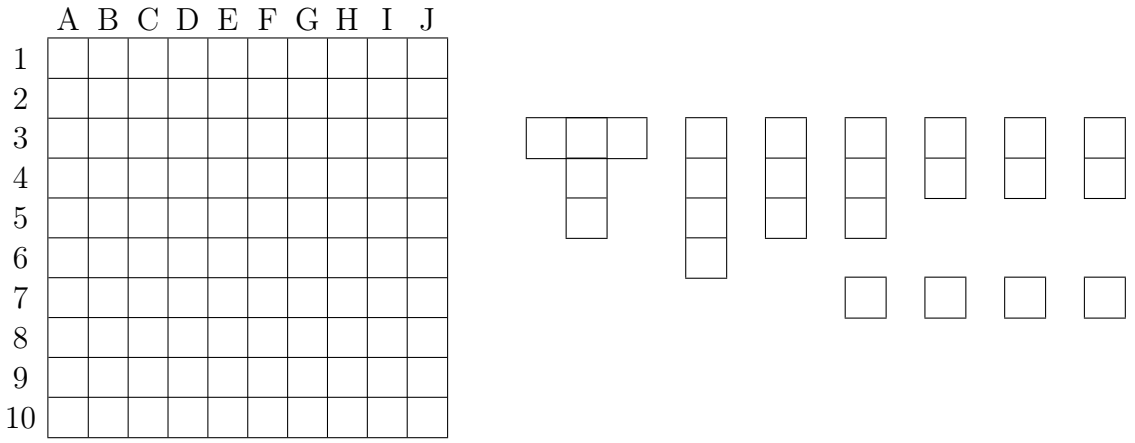
   (c) Uma justificação para terem escolhido aquelas estruturas de dados e não outras.

   (d) O estudo das complexidades temporais das operações do enunciado, no melhor caso, no pior caso e no caso esperado.

   (e) O estudo da complexidade espacial da solução proposta.

   Neste exercício, assuma que a única estrutura de dados base disponível (para além dos tipos básicos) é o vector.

5. O jogo da Batalha Naval simula um ataque a uma frota de barcos. O jogador deverá afundar toda a frota inimiga, que é composta por um porta-aviões, um barco de 4 canos, dois barcos de 3 canos, três barcos de 2 canos e quatro submarinos (barcos de 1 cano). O porta-aviões ocupa "o espaço de um T" e os barcos de n canos ocupam o espaço de n quadrados, em fila ou em coluna (como se ilustra na figura). Os barcos estão posicionados numa grelha quadrada (que, no jogo habitual é de dez por dez) e não se tocam, nem sequer na diagonal.



   O objectivo do *jogo singular* da batalha naval é afundar a frota inimiga, através de uma sequência de rajadas. Cada *rajada* é constituída por três tiros. Após cada rajada, sabe-se que tipos de barcos foram atingidos e (se for o caso) afundados. Um barco diz-se *afundado* quando todas as posições que ocupa foram atingidas por um tiro. Um *tiro* é composto por um par (fila,coluna), que identifica uma posição na grelha. O jogo termina quando todos os barcos forem afundados.

   Pretende-se implementar o jogo singular da batalha naval, quando a configuração da frota é obtida por um processo que iremos ignorar neste contexto.

   O programa interpreta os seguintes comandos.

   - **r x1 y1 x2 y2 x3 y3**: envio de uma rajada constituída por três tiros (o primeiro tiro é dado pelo par (x1,y1) e assim por diante).

     O programa fornece a seguinte informação, respeitante aos **tiros da rajada**.

1. Número de tiros *inválidos*.
   Um tiro é *válido* se corresponder a uma posição da grelha.
2. Número de tiros (válidos e) repetidos.
   Por exemplo, esse número é 1 na rajada (4,5), (2,3), (4,5).
3. Número de tiros (válidos, distintos e) *não originais*.
   Um tiro é *original* se atinge uma posição que ainda não tinha sido atingida.
4. Número de tiros (originais) que atingiram submarinos.
5. Número de tiros (originais) que atingiram barcos de 2 canos.
6. Número de tiros (originais) que atingiram barcos de 3 canos.
7. Número de tiros (originais) que atingiram o barco de 4 canos.
8. Número de tiros (originais) que atingiram o porta-aviões.
9. Número de submarinos afundados (nesta rajada).
10. Número de barcos de 2 canos afundados (nesta rajada).
11. Número de barcos de 3 canos afundados (nesta rajada).
12. Número de barcos de 4 canos afundados (nesta rajada).
13. Número de porta-aviões afundados (nesta rajada).
14. Indicação de que o jogo continua (porque ainda há barcos por afundar) ou terminou.
15. Número total de rajadas enviadas durante o jogo (que permite calcular a pontuação obtida pelo jogador), apenas no caso do jogo ter terminado.

- **d**: o jogo termina por desistência do jogador.
- **g**: apresenta-se a grelha, assinalando as posições atingidas pelos tiros dados até ao momento.
- **?**: afixa-se um texto explicando os comandos relacionados com o jogo.

Elabore o **Modelo dos Tipos Abstractos de Dados** deste problema.

6. Considere o tipo abstracto de dados *Fila Concatenável* de elementos do tipo E, caracterizado pela interface *ConcatenableQueue*.

```
public interface ConcatenableQueue<E> extends Queue<E>
{
        // Removes all of the elements from the specified queue and
        // inserts them at the end of the queue (in proper order).
        void append( ConcatenableQueue<E> queue );
}
```

Por exemplo, se q_1 e q_2 forem filas concatenáveis com os elementos

$$5,\ 44,\ 17,\ 10 \quad \text{e} \quad 11,\ 17,\ 50,$$

após $q_1.append(q_2)$, q_1 fica com 5, 44, 17, 10, 11, 17, 50 e q_2 fica vazia.

(a) Implemente este TAD.

(b) Calcule as complexidades temporais das operações, no melhor caso, no pior caso e no caso esperado.

7. Considere o tipo abstracto de dados *Fila Invertível* de elementos do tipo E, caracterizado pela interface *InvertibleQueue*.

```
public interface InvertibleQueue<E> extends Queue<E>
{
    // Puts all elements in the queue in the opposite order.
    void invert( );
}

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

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

    // Inserts the specified element at the rear of the queue.
    void enqueue( E element );

    // Removes and returns the element at the front of the queue.
    E dequeue( ) throws EmptyQueueException;
}
```

Por exemplo, se q for uma fila invertível com os elementos

$$5,\ 44,\ 17,\ 10,\ 11,\ 17,\ 50,$$

após $q.invert()$ e $q.enqueue(3)$, q fica com 50, 17, 11, 10, 17, 44, 5, 3.

(a) Implemente este TAD.

(b) Calcule as complexidades temporais das operações, no melhor caso, no pior caso e no caso esperado.

8. O objectivo deste exercício é o desenvolvimento de uma aplicação para testar a implementação da classe *InvertibleQueue*. Neste exercício a persistência dos dados deve ser assegurada entre execuções consecutivas. Isto significa que, antes de terminar a execução, o sistema deve guardar o estado da base de dados em disco, utilizando as funcionalidades de serialização do Java. Na próxima execução do programa, os dados armazenados deverão ser carregados do disco e o estado do sistema reconstituído. Dada uma Fila invertível, A aplicação deve permitir:

    1. Inserir um elemento na fila (comando `ENQ`). É fornecido o elemento a inserir. A operação sucede sempre (Enqueue efectuado com sucesso.)
    2. Remover um elemento na fila (comando `DEQ`). Em caso de sucesso, é apresentado o elemento removido seguido da a mensagem (Dequeue efectuado com sucesso.) A operação falha se a fila estiver vazia (Fila vazia.)
    3. Inverter a fila (comando `INVERT`). A operação sucede sempre retornando a mensagem (Inversao da fila efectuada com sucesso.)

4. Remoção de todos os elementos da fila (comando `DEQ-ALL`). Em caso de sucesso, a operação apresenta o cabeçalho (Dequeue de todos os elementos da fila:), seguido dos elementos removidos. A operação falha se a fila estiver vazia (Fila vazia.)
5. Terminar a execução do programa (comando `XS`). A operação tem sempre sucesso e não produz uma mensagem de saída.

(a) Complete os métodos load() e save() contidos no ficheiro Main.java disponibilizado no Moodle.

(b) Teste o programa com os testes disponibilizados no Moodle, utilizando a linha de comando. Pode apoiar-se nas instruções disponíveis no Moodle e nos vídeos também fornecidos.

(c) Depois de verificar o código localmente, submeta o seu programa no Mooshak.

9. Relembre a definição recursiva dos números de Fibonacci:

$$\text{Fibonacci}(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2), & \text{se } n \geq 2. \end{cases}$$

(a) Implemente uma versão recursiva desta função. (O tipo do resultado deve ser *long*.)

(b) Meça os tempos de execução do seu programa, quando a entrada é da forma $10\,k$, com $k = 0, 1, 2, \ldots, 6$.

(c) Implemente uma versão iterativa da mesma função.

(d) Meça os tempos de execução do seu novo programa, quando a entrada é da forma $10\,k$, com $k = 0, 1, 2, \ldots, 10$.

(e) Compare os tempos de execução dos dois programas e estude a complexidade temporal dos algoritmos.

10. (a) Relembre a definição recursiva habitual da potência (de base real e expoente natural) e implemente uma versão iterativa desta função.

$$x^n = \begin{cases} 1, & \text{se } n = 0; \\ x^{n-1} * x, & \text{se } n > 0. \end{cases}$$

(b) Considere a seguinte definição da potência (de base real e expoente natural) e implemente uma versão recursiva desta função.

$$x^n = \begin{cases} 1, & \text{se } n = 0; \\ (x * x)^{\frac{n}{2}}, & \text{se } n \text{ é par positivo}; \\ (x * x)^{\lfloor \frac{n}{2} \rfloor} * x, & \text{se } n \text{ é ímpar}. \end{cases}$$

(c) Meça e compare os tempos de execução dos dois algoritmos, quando o expoente é da forma 10^k, com $k = 1, 2, 3, \ldots, 18$.

(d) Estude a complexidade temporal dos dois algoritmos.

11. (a) Implemente uma versão iterativa da função definida na alínea (b) da pergunta 10.

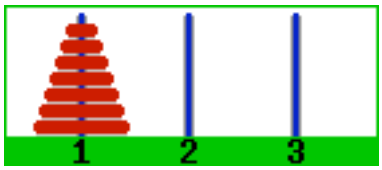
    (b) Estude a complexidade temporal do seu algoritmo.

12. Implemente uma função recursiva que devolva, como resultado, o valor máximo contido num vector de números inteiros. Assuma que o vector não é vazio. Estude a complexidade temporal do seu algoritmo.

13. Implemente uma função recursiva que devolva o resultado da multiplicação de dois números inteiros. A operação deverá ser realizada utilizando apenas operações de soma e subtracção. Estude a complexidade temporal do seu algoritmo.

14. No Problema das Torres de Hanoi, existem $n \geq 1$ discos, todos de diâmetros diferentes, e 3 estacas (denominadas 1, 2 e 3). Pretende-se deslocar os n discos, que se encontram *em pirâmide* na estaca 1, para a estaca 3,

    - movendo um disco de cada vez e
    - não podendo nunca colocar um disco maior sobre um disco menor.



Uma sequência de movimentos que resolve o problema pode ser obtida pelo seguinte algoritmo.

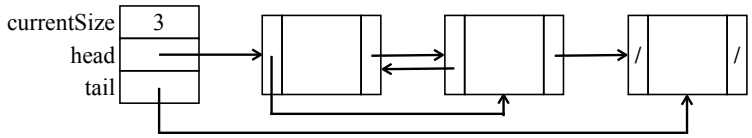
```
static void hanoi( int numberOfDisks )
{
        if ( numberOfDisks >= 1 )
                hanoi(numberOfDisks, 1, 3, 2);
}

static void hanoi( int nDisks, int source, int destination, int theOther )
{
        if ( nDisks == 1 )
                // Mover o disco da origem para o destino.
                moveDisk(source, destination);
        else
        {
                // Mover os nDisks−1 discos menores da origem para a auxiliar.
                hanoi(nDisks − 1, source, theOther, destination);
                // Mover o disco maior da origem para o destino.
                moveDisk(source, destination);
                // Mover os nDisks−1 discos menores da auxiliar para o destino.
                hanoi(nDisks − 1, theOther, destination, source);
        }
}
```

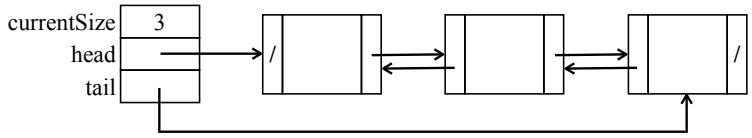
```
static void moveDisk( int source, int destination )
{
        System.out.print("Mova o disco (do topo) da estaca " + source);
        System.out.println(" para a estaca " + destination);
}
```

Determine a complexidade temporal do método *hanoi*(n), com $n \geq 1$, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

15. Considere que se pretende acrescentar uma funcionalidade à classe *DoubleList* que permita reconstruir a estrutura da lista apenas a partir dos apontadores para o nó seguinte, para além da cabeça e cauda. A classe *RestorableDoubleList* acrescenta esta funcionalidade, que é útil quando, por alguma razão, os apontadores para o nó anterior não foram correctamente afectados. Por exemplo, considere a lista:



Na lista ilustrada o apontador para o nó anterior da cabeça está incorrecto, devia ser *null*. Também o apontador para o nó anterior da cauda está incorrecto. Após restaurar os apontadores para o nó anterior, a lista ilustrada em cima fica então com a seguinte estrutura de ligações:



Implemente iterativamente e recursivamente a operação que restaura os apontadores para o nó anterior como descrito em cima, de acordo com a seguinte especificação:

```
public class RestorableDoubleList<E> extends DoubleList<E>
{
// Restores all previous pointers in the object list,
// based on next, head and tail pointers.
// Iterative implementation
protected void restoreAllPreviousPointersIt() {
...
}

// Restores all previous pointers in the object list,
// based on next, head and tail pointers.
// Recursive implementation
protected void restoreAllPreviousPointersRec() {
...
}
...
}
```

Programe também todos os métodos auxiliares necessários. Indique a complexidade dos métodos implementados no melhor caso, no pior caso e no caso esperado.

16. Considere que se pretende acrescentar uma funcionalidade à classe *DoubleList* que permita, dado um elemento, encontrar o nó que contém a referência para o elemento, caso exista. Caso o elemento não exista na lista então o resultado da pesquisa deve ser *null*. A classe *SearchableDoubleList* acrescenta esta funcionalidade, que é útil quando a lista é usada para fins de pesquisa. Implemente recursivamente a operação que encontra o nó que contém um determinado elemento de acordo com a seguinte especificação:

```
public class SearchableDoubleList<E> extends DoubleList<E>
{
// Returns the reference to the node that contains the
// given element or null if no such node is found.
// Recursive implementation
protected DoubleListNode<E> findNodeRec(E element) {
...
}
...
}
```

Programe também todos os métodos auxiliares necessários. Indique a complexidade dos métodos implementados no melhor caso, no pior caso e no caso esperado.

17. Programe a classe *OrderedDoubleList*, que implementa a interface *OrderedDictionary* através de uma lista duplamente ligada com cabeça e cauda. Esta classe servirá para suportar as listas de colisões da Tabela de Dispersão, que é um dos requisitos da 2ª fase do trabalho.

    ATENÇÃO: A *OrderedDoubleList* não é um dicionário.

    Como este exercício faz parte da 2ª fase do trabalho prático, deve ser desenvolvido, a partir do código fornecido pelo docente, e de forma autónoma, sem partilha de soluções entre grupos.

    Lembramos que, dado que responde à interface dicionário, esta classe deve apenas contemplar, como métodos públicos, o construtor e os métodos de inserção, remoção e pesquisa por chave.

    Para apoiar a implementação, poderá utilizar alguns dos métodos da *DoubleList* copiando o seu código para a nova classe e fazendo as alterações devidas.

    A classe irá apoiar-se na classe *DoubleListNode*, que deverá tornar-se independente (crie um ficheiro .java para a mesma), embora não pública. O iterador da *OrderedDoubleList* será o mesmo já fornecido para a *DoubleList*.