Algoritmos e Estruturas de Dados
Lista de Exercícios

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

Ano Lectivo 2019/20

Tipos Abstractos de Dados

1. O Tipo Abstrato de Dados (TAD) *CalorieConsumer* define as operações associadas a um indivíduo que pretende manter um registo das calorias que vai consumindo em refeições, ao longo de um período de tempo não especificado. Este TAD deve manter informação pessoal sobre o indivíduo (como o nome e a idade) e deve permitir a atualização dos dados relativos ao seu consumo de calorias, como a inserção de refeições e o acesso a algumas estatísticas (como o total de calorias consumido e o valor da refeição mais calórica consumida até ao momento). A partir deste TAD deve ainda ser possível listar as calorias de todas as refeições consumidas pelo consumidor até ao momento, por ordem cronológica, iniciando na mais recente e terminando na mais antiga. Apresenta-se abaixo o interface *CalorieConsumer*:

```
import dataStructures.Iterator;

interface CalorieConsumer {
    //Devolve o nome do consumidor.
    String getName();

    //Devolve a idade do consumidor.
    int getAge();

    //Incrementa a idade do consumidor, somando 1.
    void incrementAge();

    //Adiciona as calorias de uma refeição ao consumidor.
    //Requires: mealCalories > 0
    void addMeal(int mealCalories) throws NotValidException;

    //Devolve iterador (de calorias) das refeições do consumidor.
    // A iteração executada a partir deste iterador deve apresentar
    // as (calorias associadas às) refeições por ordem cronológica, iniciando
    // na refeição mais recente e terminando na mais antiga.
    //Requires: O consumidor tem de ter consumido (pelo menos) uma refeição
    Iterator<Integer> listMeals() throws EmptyMealsException;
```

```
    //Devolve total de calorias consumido pelo consumidor.
    int getCalories();
    //Devolve o valor de calorias da refeição mais calórica consumida
    //pelo consumidor.
    //Requires: O consumidor tem de ter consumido (pelo menos) uma refeição
    int getHighestMeal() throws EmptyMealsException;
}
```

Indique as variaveis de instância que usaria para implementar de forma adequada a interface *CalorieConsumer*. Justifique. Explique como implementaria cada método da interface.

2. Um *multiconjunto* é uma colecção de elementos que admite repetições. Por exemplo, $\{a,b,a,d,a,c,c\}$ é um multiconjunto com sete elementos (o "*size*" é 7). Se se inserisse 'a', o multiconjunto passaria a ser $\{a,b,a,d,a,c,c,a\}$. Se depois, se removesse 'c', o multiconjunto resultante seria $\{a,b,a,d,a,c,a\}$. Note que a ordem de apresentação dos elementos é irrelevante. É fácil concluir que um multiconjunto é um conjunto quando não tem elementos repetidos.
Considere o tipo abstracto de dados *Multiconjunto*, de elementos do tipo E, caracterizado pela interface *Multiset*.

```
public interface Multiset<E> {
    // Returns the number of elements in the multiset.
    int size( );

    // Returns true iff the multiset contains the specified element.
    boolean contains( E element );

    // Inserts the specified element in the multiset.
    void insert( E  element );

    // Removes one occurrence of the specified element from the multiset
    // and returns true, if the multiset contains the element.
    // Otherwise, returns false.
    boolean remove( E element );

    // Returns true iff the multiset is a set.
    boolean isASet( );
}
```

Indique as variaveis de instância que usaria para implementar de forma adequada a interface *Multiset*. Justifique. Explique como implementaria cada método da interface.

3. O Tipo Abstrato de Dados *Evaluation* define as operações associadas à gestão da avaliação anual dos colaboradores de uma empresa de vendas de automóveis. Um colaborador é identificado pelo seu workerId e é avaliado, várias vezes por ano, com um valor entre 0 e 10 valores. O sistema a desenvolver deve guardar apenas as avaliações (únicas) de cada colaborador para o ano que está a decorrer e deve também poder listar os identificadores dos colaboradores que obtiveram a avaliação máxima, até ao momento, por ordem alfabética do identificador. Apresenta-se abaixo o interface *Evaluation*:

```
public interface Evaluation {
```

```
    //Insere a avaliação anual do colaborador identificado por workerId.
    void insertEvaluation(String workerId, int value)
                        throws ExistingWorker, NonValidValue;

    //Devolve o valor médio das avaliações do trabalhador identificado
    // por workerId.
    float currentEvaluation(String workerId) throws NonExistingWorker;

    //Devolve o valor da avaliação máxima efetuada
    int maxEvaluationValue() throws NoEvaluations;

    //Devolve um iterador com os identificadores de todos os colaboradores
    //com avaliação máxima, por qualquer alfabética do identificador.
    Iterator<String> listBestWorkers();
}
```

Indique as variaveis de instância que usaria para implementar de forma adequada a interface *Evaluation*. Justifique. Explique como implementaria cada método da interface.

4. Pretendemos implementar um sistema de gestão de textos de grande dimensão. Um texto é constituído por várias linhas e cada linha por várias palavras. Não estão definidos máximos para o número de linhas de cada texto, nem para o número de palavras existentes em cada linha. Cada linha tem um número de ordem atribuído e pode ocorrer repetição de palavras ao longo do texto. Abaixo pode consultar a interface Java definida para o tipo abstrato de dados (TAD) LineText . Este contém operações que permitem adicionar linhas ao texto (sempre no final do mesmo), saber o número de ocorrências de cada palavra existente no texto, consultar os números das linhas onde uma determinada palavra ocorre e consultar uma determinada linha do texto, inserindo o seu número de ordem.

```
public interface LineText {
    //Adiciona linha de texto no final do mesmo
    void addLine( String line );

    //Devolve o número de ocorrências atual de uma palavra no texto
    int numberOfOccurrences( String word );

    //Devolve um iterador dos números de ordem das linhas onde a palavra
    //word é mencionada no texto. Este iterador deve iterar os números de
    //linha crescentemente.

    Iterator<Integer> whereIsWord( String word ) throws NonExistingWord;

    //Devolve a linha com o número de ordem lineNumber
    String showLine( int lineNumber ) throws NonExistingLines;
}
```

Indique as variaveis de instância que usaria para implementar de forma adequada a interface *LineText*. Justifique. Explique como implementaria cada método da interface.

Estruturas de Dados fundamentais

5. Exercício 3.17 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

6. Exercícios 3.6 e 3.12 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

7. Exercícios 3.11 e 3.16 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

8. Considere que se pretende acrescentar uma funcionalidade à classe *DoublyLinkedList* 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 *SearchableDoublyLL* 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 SearchableDoublyLL<E> extends DoublyLinkedList<E>
{
// Returns the reference to the node that contains the
// given element or null if no such node is found.
// Recursive implementation
protected DListNode<E> findNodeRec(E element) {
...
}
...
}
```

   Programe também todos os métodos auxiliares necessários.

Análise de algoritmos

9. Exercícios 4.9 a 4.13 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

10. Exercícios 5.1 e 5.8 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

11. Na classe *SearchableDoublyLL* implementada, indique a complexidade dos métodos implementados no melhor caso, no pior caso e no caso esperado.

12. Relembre a definição recursiva dos números de Fibonacci:

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

    (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.

13. (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.

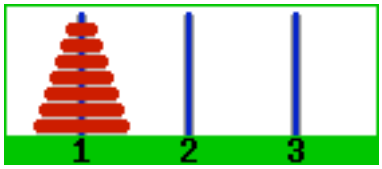
(e) Implemente uma versão iterativa da função definida na alínea (b) e estude a complexidade temporal do seu algoritmo.

14. 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.

15. 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.

16. No Problema das Torres de Hanoi, existem $n \geqslant 1$ 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 $n \geqslant 1$, no melhor caso, no pior caso e no caso esperado, justificando todos os cálculos com muita clareza.

Escolha das Estruturas de Dados adequadas para implementar algumas interfaces

17. Explicite detalhadamente as estruturas de dados mais adequadas para implementar a interface *CalorieConsumer*, descreva brevemente como implementaria as operações e calcule as suas complexidades temporais no caso esperado, justificando.

18. Explicite detalhadamente as estruturas de dados mais adequadas para implementar a interface *Multiset*, descreva brevemente como implementaria as operações e calcule as suas complexidades temporais no caso esperado, justificando. Se quiser, pode assumir que os elementos são comparáveis entre si, com complexidade constante.

19. Explicite detalhadamente as estruturas de dados mais adequadas para implementar a interface *Evaluation*, descreva brevemente como implementaria as operações e calcule as suas complexidades temporais no caso esperado, justificando. Se quiser, pode assumir que os elementos são comparáveis entre si, com complexidade constante.

20. Explicite detalhadamente as estruturas de dados mais adequadas para implementar a interface *LineText*, descreva brevemente como implementaria as operações e calcule as suas complexidades temporais no caso esperado, justificando. Se quiser, pode assumir que os elementos são comparáveis entre si, com complexidade constante.

TAD - Queue

21. Exercícios 6.7 e 6.12 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

22. 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.

23. 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.

TAD - Stack

24. Exercícios 6.3 e 6.5 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

25. Implemente o método *ExpressionEvaluation* que, dada uma expressão matemática de valores inteiros com os operadores binários (+, - * e /) em notação posfixa, retorne o valor da expressão. Por exemplo, ExpressionEvaluation("12 5 + 4 *") retorna 68 ((12+5)*4 = 68); e ExpressionEvaluation("12 5 4 + *") retorna 108 (12*(5+4) = 108). Calcule as complexidades temporais, no melhor caso, no pior caso e no caso esperado.

TAD - Map

26. Exercícios 10.6 e 10.7 do livro - "Data Structures and Algorithms in Java", Michael T. Goodrich and Roberto Tamassia (6th edition), 2015.

27. Programe a classe *MapWithSinglyLinkedList*, que implementa a interface *Map* através de uma lista simplesmente ligada com cabeça e cauda.

TAD - SortedMap

28. Programe a classe *SortedMapWithOrderedDoubleList*, que implementa a interface *SortedMap* através de uma lista ordenada duplamente ligada com cabeça e cauda.

29. Implemente o iterador infixo da árvore binária de pesquisa.

    Calcule as complexidades temporais dos métodos do iterador, no melhor caso, no pior caso e no caso esperado, justificando. Quais são as complexidades temporal e espacial de uma iteração completa? Justifique.

30. Implemente um iterador que percorre as entradas de uma árvore binária de pesquisa por níveis e, dentro de cada nível, da esquerda para a direita. Ou seja, o percurso começa pela raiz, prossegue pelos filhos da raiz (filho esquerdo e filho direito), continua pelos netos da raiz, só depois passa aos bisnetos da raiz, etc.

Calcule as complexidades temporais dos métodos do iterador, no melhor caso, no pior caso e no caso esperado, justificando. Quais são as complexidades temporal e espacial de uma iteração completa? Justifique.

31. Apresente um algoritmo eficiente (em tempo e espaço) que teste se dois dicionários ordenados possuem as mesmas chaves. Assuma que a probabilidade de haver chaves distintas é elevada.

```
public class BinarySearchTree<K extends Comparable<K>, V>
        implements SortedMap<K,V>
{
        // Returns true iff the set of keys in the dictionary is equal
        // to the set of keys in the specified dictionary.
        public boolean equalKeys( SortedMap<K,V> dictionary );
}
```

Assumindo que os dois dicionários estão implementados em árvore binária de pesquisa (ambas razoavelmente equilibradas), calcule as complexidades temporal e espacial do seu algoritmo, no melhor caso e no pior caso, justificando.

32. Considere uma árvore binária de pesquisa, onde cada nó possui um campo que fornece o número de nós da subárvore esquerda (através do método inteiro *leftSubtreeSize*()). Apresente um algoritmo para encontrar a n-ésima entrada do percurso infixo da árvore.

```
public class BinarySearchTree<K extends Comparable<K>, V>
        implements SortedMap<K,V>
{
        // Returns the nth entry of the tree inorder traversal.
        public Entry<K,V> nthEntry( int n ) throws NoSuchEntryException;
}
```

Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

33. Implemente um método que, dados os percursos prefixo e infixo de uma árvore binária, denotados por p e i, respectivamente, devolve a árvore binária cujo percurso prefixo é p e cujo percurso infixo é i. Assuma que a árvore não tem elementos repetidos e que os percursos estão guardados em vector.

```
public class BinaryTree<E>
{
        // Creates a binary tree with the specified
        // preorder and inorder traversals.
        public BinaryTree( E[] preorder, E[] inorder );
}
```

Calcule a complexidade temporal do seu algoritmo, no melhor caso, no pior caso e no caso esperado, justificando.

TAD - PriorityQueue

34. Implemente um iterador que percorre as entradas de um *heap* por níveis e, dentro de cada nível, da esquerda para a direita. Ou seja, o percurso começa pela raiz, prossegue pelos filhos da raiz (filho esquerdo e filho direito), continua pelos netos da raiz, só depois passa aos bisnetos da raiz, etc.

```
public class MinHeap<K extends Comparable<K>, V>
      implements MinPriorityQueue<K,V>
{
      // Returns a breadth first iterator of the tree.
      public Iterator<Entry<K,V>> breadthFirstIterator( )
      {
            return new HeapBreadthFirstIterator<K,V>( ··· );
      }
}
```

Calcule as complexidades temporais dos métodos do iterador, no melhor caso, no pior caso e no caso esperado, justificando. Quais são as complexidades temporal e espacial de uma iteração completa? Justifique.