Exercícios Preparação APF

Exercício E1

Crie um módulo `LeakyQueue`, baseado na estrutura de dados fila, de forma a que o programa `ProgX` funcione devidamente[1].

Uma fila "rota" (*leaky queue*) é uma estrutura de dados baseada numa fila, mas em que só ficam armazenados, no máximo, os últimos N números inseridos. Quando a fila está preenchida (N elementos) a inserção de um novo número implica a saída do primeiro (que deixa de existir).

Exemplos de utilização ($N = 3$) e resultados esperados:

```
java -ea ProgX 1 2 3 4 5 6
i = 0    1.0              (Min = 1.0)
i = 1    1.0  2.0         (Min = 1.0)
i = 2    1.0  2.0  3.0    (Min = 1.0)
i = 3    2.0  3.0  4.0    (Min = 2.0)
i = 4    3.0  4.0  5.0    (Min = 3.0)
i = 5    4.0  5.0  6.0    (Min = 4.0)
```

```
java -ea ProgX 9 8 7 6 5 4 3 2 1
i = 0    9.0              (Min = 9.0)
i = 1    9.0  8.0         (Min = 8.0)
i = 2    9.0  8.0  7.0    (Min = 7.0)
i = 3    8.0  7.0  6.0    (Min = 6.0)
i = 4    7.0  6.0  5.0    (Min = 5.0)
i = 5    6.0  5.0  4.0    (Min = 4.0)
i = 6    5.0  4.0  3.0    (Min = 3.0)
i = 7    4.0  3.0  2.0    (Min = 2.0)
i = 8    3.0  2.0  1.0    (Min = 1.0)
```

```
java -ea ProgX 1 3 - 5 7 - 9 11 -
i = 0    1.0              (Min = 1.0)
i = 1    1.0  3.0         (Min = 1.0)
i = 2    3.0              (Min = 3.0)
i = 3    3.0  5.0         (Min = 3.0)
i = 4    3.0  5.0  7.0    (Min = 3.0)
i = 5    5.0  7.0         (Min = 5.0)
i = 6    5.0  7.0  9.0    (Min = 5.0)
i = 7    7.0  9.0 11.0    (Min = 7.0)
i = 8    9.0 11.0         (Min = 9.0)
```

```
java -ea ProgX 2 - - 4 - 6 8
i = 0    2.0              (Min = 2.0)
i = 1
i = 2
i = 3    4.0              (Min = 4.0)
i = 4
i = 5    6.0              (Min = 6.0)
i = 6    6.0  8.0         (Min = 6.0)
```

[1] Não pode usar os módulos do pacote `exameP2` neste problema.

Exercício E2

O programa `ProgX` serve para verificar se uma expressão aritmética (formada por dígitos, operações elementares e parêntesis) é sintacticamente válida. Construa o módulo `PilhaX`, baseado na estrutura de dados pilha, de forma a que este programa funcione devidamente[2].

Exemplos de utilização e resultados esperados:

`java -ea ProgX "2+2"`

`  PUSH: D`
`REDUCE: e`
`  PUSH: e+`
`  PUSH: e+D`
`REDUCE: e+e`
`REDUCE: e`
`Correct expression!`

`java -ea ProgX "2+(2-3)"`

`  PUSH: D`
`REDUCE: e`
`  PUSH: e+`
`  PUSH: e+(`
`  PUSH: e+(D`
`REDUCE: e+(e`
`  PUSH: e+(e-`
`  PUSH: e+(e-D`
`REDUCE: e+(e-e`
`REDUCE: e+(e`
`  PUSH: e+(e)`
`REDUCE: e+e`
`REDUCE: e`
`Correct expression!`

`java -ea ProgX "3*(4/(3))"`

`  PUSH: D`
`REDUCE: e`
`  PUSH: e*`
`  PUSH: e*(`
`  PUSH: e*(D`
`REDUCE: e*(e`
`  PUSH: e*(e/`
`  PUSH: e*(e/(`
`  PUSH: e*(e/(D`
`REDUCE: e*(e/(e`
`  PUSH: e*(e/(e)`
`REDUCE: e*(e/e`
`REDUCE: e*(e`
`  PUSH: e*(e)`
`REDUCE: e*e`
`REDUCE: e`
`Correct expression!`

`java -ea ProgX "2+"`

`  PUSH: D`
`REDUCE: e`
`  PUSH: e+`
`Bad expression!`

`java -ea ProgX "(3*(2+4)+5))"`

`  PUSH: (`
`  PUSH: (D`
`REDUCE: (e`
`  PUSH: (e*`
`  PUSH: (e*(`
`  PUSH: (e*(D`
`REDUCE: (e*(e`
`  PUSH: (e*(e+`
`  PUSH: (e*(e+D`
`REDUCE: (e*(e+e`
`REDUCE: (e*(e`
`  PUSH: (e*(e)`
`REDUCE: (e*e`
`REDUCE: (e`
`  PUSH: (e+`
`  PUSH: (e+D`
`REDUCE: (e+e`
`REDUCE: (e`
`  PUSH: (e)`
`REDUCE: e`
`  PUSH: e)`
`Bad expression!`

`java -ea ProgX "2+4*(4++5)"`

`  PUSH: D`
`REDUCE: e`
`  PUSH: e+`
`  PUSH: e+D`
`REDUCE: e+e`
`REDUCE: e`
`  PUSH: e*`
`  PUSH: e*(`
`  PUSH: e*(D`
`REDUCE: e*(e`
`  PUSH: e*(e+`
`  PUSH: e*(e++`
`  PUSH: e*(e++D`
`REDUCE: e*(e++e`
`  PUSH: e*(e++e)`
`Bad expression!`

[2]Não pode usar os módulos do pacote `exameP2` neste problema.

Exercício E3

O programa `MainTrain` demonstra a utilização de uma estrutura de dados para gerir a carga e descarga de vagões num comboio de mercadorias. Crie o módulo `Train` de forma que este programa compile e funcione devidamente[3].

Um objecto da classe Train representa um comboio composto de vários vagões de mercadorias a granel. Quando se cria um comboio, é necessário especificar a capacidade de cada vagão e a capacidade total que o comboio suporta, ambas em toneladas. Pode acrescentar-se um vagão com certa carga à cauda de um comboio (addWagon) ou pode retirar-se um vagão da cauda (removeWagon), segundo uma política LIFO (último a entrar é o primeiro a sair). Naturalmente, a carga de um vagão não pode superar a sua capacidade e só se pode acrescentar um vagão que não faça ultrapassar a carga total máxima do comboio. Também é possível pedir para descarregar (unload) uma dada quantidade, o que será feito pela descarga completa e retirada de zero ou mais vagões da cauda e pela descarga parcial de outro vagão para completar a quantidade pedida. Em qualquer altura é possível obter uma lista da carga nos vagões do comboio (list); saber o número de vagões (size) ou a carga total transportada (totalCargo).

Exemplos de utilização e resultados esperados:

```
java -ea MainTrain 10 100 1 2 3 R R 4.5 0.1

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 100.0 ton.)
args[2]="1": Junta vagão com 1.0 ton
  (1 vagões, 1.0 ton): Loc0_[1.0]
args[3]="2": Junta vagão com 2.0 ton
  (2 vagões, 3.0 ton): Loc0_[1.0]_[2.0]
args[4]="3": Junta vagão com 3.0 ton
  (3 vagões, 6.0 ton): Loc0_[1.0]_[2.0]_[3.0]
args[5]="R": Retira vagão com 3.0 ton
  (2 vagões, 3.0 ton): Loc0_[1.0]_[2.0]
args[6]="R": Retira vagão com 2.0 ton
  (1 vagões, 1.0 ton): Loc0_[1.0]
args[7]="4.5": Junta vagão com 4.5 ton
  (2 vagões, 5.5 ton): Loc0_[1.0]_[4.5]
args[8]="0.1": Junta vagão com 0.1 ton
  (3 vagões, 5.6 ton): Loc0_[1.0]_[4.5]_[0.1]
```

```
java -ea MainTrain 10 100 4 2 5 7 -2 -11 -1

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 100.0 ton.)
args[2]="4": Junta vagão com 4.0 ton
  (1 vagões, 4.0 ton): Loc0_[4.0]
args[3]="2": Junta vagão com 2.0 ton
  (2 vagões, 6.0 ton): Loc0_[4.0]_[2.0]
args[4]="5": Junta vagão com 5.0 ton
  (3 vagões, 11.0 ton): Loc0_[4.0]_[2.0]_[5.0]
args[5]="7": Junta vagão com 7.0 ton
  (4 vagões, 18.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[7.0]
args[6]="-2": Descarrega 2.0 ton e retira 0 vagões vazios.
  (4 vagões, 16.0 ton): Loc0_[4.0]_[2.0]_[5.0]_[5.0]
args[7]="-11": Descarrega 11.0 ton e retira 2 vagões vazios.
  (2 vagões, 5.0 ton): Loc0_[4.0]_[1.0]
args[8]="-1": Descarrega 1.0 ton e retira 1 vagões vazios.
  (1 vagões, 4.0 ton): Loc0_[4.0]
```

```
java -ea MainTrain 10 20 2 10 11

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 20.0 ton.)
args[2]="2": Junta vagão com 2.0 ton
  (1 vagões, 2.0 ton): Loc0_[2.0]
args[3]="10": Junta vagão com 10.0 ton
  (2 vagões, 12.0 ton): Loc0_[2.0]_[10.0]
args[4]="11": ERRO: Sobrecarga de vagão!
```

```
java -ea MainTrain 10 20 5 7 9

(Capacidade dos vagões: 10.0 ton.)
(Capacidade do comboio: 20.0 ton.)
args[2]="5": Junta vagão com 5.0 ton
  (1 vagões, 5.0 ton): Loc0_[5.0]
args[3]="7": Junta vagão com 7.0 ton
  (2 vagões, 12.0 ton): Loc0_[5.0]_[7.0]
args[4]="9": ERRO: Sobrecarga do comboio!
```

[3]Não pode usar os módulos do pacote `exameP2` neste problema.

Exercício E4

Pretende-se construir um programa (`PhoneCalls.java`) que processe uma lista de chamadas telefónicas que estão descritas em ficheiros do tipo `*.cls` (por exemplo: `calls.cls`, com a organização seguinte: *número de origem*, *número de destino* e *duração* em segundos). Tem ainda disponível ficheiros do tipo `*.nms` (por exemplo: `names.nms`) com a informação seguinte: *número* e *nome*[4].

calls.cls

```
009047362 269633507 287
269633507 545065453 723
269633507 021693118 680
513512774 269633507 265
564359070 564359070 751
503512774 396659735 475
071356756 181964754 719
```

names.nms

```
396659735 Sergio Tavares
269633507 Paula Nunes
208974207 Mario Nunes
462589991 Maria Nunes
564359070 Joao Nunes
181964754 Ana Nunes
503512774 Paula Melo
009047362 Miguel Silva
482318937 Pedro Oliveira
071356756 Tomas Alberto
```

a. Utilizando da melhor forma possível o pacote `exameP2`, faça um programa que leia a informação dos ficheiros do tipo `*.nms` passados como argumento para uma (ou mais) estrutura de dados apropriada. Por outro lado, para os argumentos terminados em `.cls`, o programa deve listar o seu conteúdo, mas trocando o número de telemóvel pelo nome do dono, se já for conhecido. Os ficheiros devem ser processados pela ordem dos argumentos. Por exemplo:

java -ea PhoneCalls names.nms calls.cls

```
Miguel Silva to Paula Nunes (287 seconds)
Paula Nunes to 545065453 (723 seconds)
Paula Nunes to 021693118 (680 seconds)
513512774 to Paula Nunes (265 seconds)
Joao Nunes to Joao Nunes (751 seconds)
Paula Melo to Sergio Tavares (475 seconds)
Tomas Alberto to Ana Nunes (719 seconds)
```

b. Altere o programa de forma que qualquer argumento que não termine com as extensões definidas (`.nms` ou `.cls`) seja considerado um número de telemóvel. Para cada um desses argumentos, o programa deve escrever imediatamente a lista de chamadas feitas por esse telemóvel, e a lista de chamadas recebidas. Tal como na alínea anterior trocando o número de telemóvel pelo nome do dono, sempre que possível.

java -ea PhoneCalls names.nms calls.cls 269633507

```
...
Calls made by Paula Nunes:
  - to phone 021693118 (680 seconds)
  - to phone 545065453 (723 seconds)
Calls received by Paula Nunes:
  - from phone 513512774 (265 seconds)
  - from Miguel Silva (287 seconds)
```

[4]O número é a primeira palavra da linha, sendo as restantes palavras consideradas como sendo o nome.

Exercício E5

Pretende-se construir um programa (`CityTraveler.java`) que apresente as cidades visitadas por cada um dos funcionários de uma empresa. Para cada cidade existe um ficheiro com a lista dos funcionários que a visitaram.

Por exemplo, dados os seguintes ficheiros:

Aveiro
Maria
Marisa
Miguel
António
Luis
José

Porto
Luis
Miguel
António
Rui
Pedro
Francisco

Lisboa
Manuel
Miguel
Maria

Se executar:

```
java -ea CityTraveler Aveiro Porto Lisboa
```

a saída do programa seria (eventualmente com os funcionários numa outra ordem):

```
Luis      : Aveiro Porto
José      : Aveiro
Rui       : Porto
Maria     : Aveiro Lisboa
Miguel    : Aveiro Porto Lisboa
Francisco : Porto
Pedro     : Porto
António   : Aveiro Porto
Marisa    : Aveiro
Manuel    : Lisboa
```

a. Utilizando o pacote `exameP2`, comece por escolher uma estrutura de dados adequada para resolver este problema e crie uma função que preencha essa estrutura com a informação retirada de um único ficheiro. Detalhes:

   - Cada linha (não vazia) do ficheiro corresponde ao nome de um funcionário.
   - O nome do ficheiro é o nome da cidade.

b. Complete o programa para fazer o pretendido, tendo em consideração que:

   - O programa recebe como argumentos os ficheiros com as listas de funcionários.
   - A lista de todos os funcionários deve ser escrita no dispositivo de saída.

Exercício E6

Construa um programa (`JustifiedText.java`) que permita alinhar um texto simultaneamente às margens esquerda e direita ("justificar" o texto). O programa recebe como parâmetros o comprimento de cada linha e o nome de um ficheiro de entrada, e deve escrever o texto justificado na consola.

Para resolver este problema tem de utilizar pelo menos uma estrutura adequada do pacote `exameP2.jar`.

Por exemplo, dado o seguinte ficheiro:

```
If one cannot enjoy reading a book over and over again, there is no use
in reading it at all.
Perfect day for scrubbing the floor and other exciting things.

You
are      standing   on my
toes.  You have taken yourself too seriously.
```

Dois exemplos de utilização do programa serão:

`java -ea JustifiedText 40 texto.txt`	`java -ea JustifiedText 30 texto.txt`

`If one cannot enjoy reading a book over and over again, there is no use in reading it at all. Perfect day for scrubbing the floor and other exciting things.`
 	
`You are standing on my toes. You have taken yourself too seriously.`
	`If one cannot enjoy reading a book over and over again, there is no use in reading it at all. Perfect day for scrubbing the floor and other exciting things.`
	
	`You are standing on my toes. You have taken yourself too seriously.`

Detalhes a ter em consideração:

- Cada linha escrita deve conter o maior número possível de palavras sem ultrapassar o comprimento definido. Considera-se "palavra" qualquer sequência de caracteres delimitada por espaços em branco.
- Não se podem juntar palavras (espaçamento nulo) nem dividir nenhuma palavra entre linhas.
- Os espaçamentos entre palavras de uma linha devem ter comprimentos iguais ou diferir no máximo de um espaço.
- A última linha de cada parágrafo deve ficar alinhada à esquerda (com um único espaço entre palavras). Considere que um parágrafo termina com uma linha vazia ou com o fim do ficheiro.

Exercício E7

Pretende-se construir um programa (`Restaurante.java`) que faz a gestão da entrada de alimentos e saída de refeições de um restaurante. A saída de uma refeição só pode ocorrer assim que o restaurante tiver as quantidades de ingredientes para ela requeridos e depois de todos os anteriores pedidos de refeições tiverem sido servidos. O programa recebe os alimentos e os pedidos de refeições através de um ou mais ficheiros de entrada (passados como argumentos do programa). Estes ficheiros têm o seguinte formato: Uma entrada de um ingrediente é uma linha com o prefixo: "`entrada: `" seguido de uma palavra com o nome do ingrediente. Os pedidos para refeições são linhas com o prefixo: "`saida: `", seguido de uma lista de palavras compostas pelo nome do ingrediente e a quantidade requerida (separadas pelo símbolo `:`). Considere que nestes pedidos não pode haver a repetição do mesmo ingrediente. Um desses ficheiros é exemplificado em baixo.

O programa a implementar deve processar todos os seus argumentos interpretando-os como indicado em cada alínea (na dúvida, verifique o comportamento do ficheiro `jar`).

a) Utilizando da melhor forma possível o pacote `exameP2`, faça um programa que leia e registe a informação das entradas de ingredientes dos ficheiros de entrada e a escreva com o formato exemplificado à frente (nesta alínea, pode ignorar completamente as saídas). Deve ser criada uma função para a leitura da informação de cada ficheiro. A seguir exemplifica-se o comportamento que o programa deve ter.

food-data01.txt

```
entrada:  cerveja
entrada:  sopa
entrada:  carne
entrada:  carne
entrada:  feijao
saida:  sopa:1 carne:1 cerveja:1
saida:  sopa:1 peixe:1 agua:1 torta:1
entrada:  sumo
entrada:  sopa
saida:  carne:1 sumo:1
entrada:  torta
saida:  carne:1 feijao:1
entrada:  peixe
entrada:  agua
```

java -ea Restaurante food-data01.txt

```
Comida em stock:
  feijao: 1
  torta: 1
  cerveja: 1
  sumo: 1
  peixe: 1
  carne: 2
  sopa: 2
  agua: 1
```

b) Altere o programa por forma a servir também as refeições. As refeições têm de ser servidas exactamente pela ordem com que aparecem nos ficheiros de entrada e logo que possível (portanto terá de modificar a função anterior). No exemplo dado, a primeira refeição pode de imediato ser servida, já que existem em stock todos os ingredientes, mas o mesmo já não acontece com a refeição seguinte. Por outro lado, a terceira refeição fica "suspensa" pelo facto da segunda ainda não ter sido servida. No final, o programa deve indicar, para além da comida em stock (alínea a), as refeições que ficaram pendentes ou por falta de ingredientes ou por estarem atrás de uma refeição pendente[5].

java -ea Restaurante food-data02.txt

```
Refeicao servida:  sopa:1 carne:1 cerveja:1
Refeicao servida:  sopa:1 peixe:1 agua:1 torta:1
Refeicao servida:  carne:1 sumo:1
Comida em stock:
  feijao: 1
Refeicao pendente:  carne:1 feijao:1
```

[5]Para facilitar a depuração do programa existem vários comandos de teste (`test*.sh`) que pode utilizar (quer para o seu programa quer para o programa `jar` fornecido).

Exercício E8

Crie um programa que, dado um número inteiro positivo como argumento, escreva todos os divisores do número que não o próprio e o número um, e, recursivamente, faça o mesmo para todos esses divisores.

Seguem alguns exemplos da execução pretendida do programa:

`java -ea AllDivisors 12`	`java -ea AllDivisors 23`	`java -ea AllDivisors 81`	`java -ea AllDivisors 32`

12
  6
  3
  2
  4
  2
  3
  2
	23	81
		  27
		  9
		  3
		  3
		  9
		  3
		  3
			32
			  16
			  8
			  4
			  2
			  2
			  4
			  2
			  2
			  8
			  4
			  2
			  2
			  4
			  2
			  2

Exercício E9

Crie um programa que dado um número racional pertencente ao intervalo aberto entre zero e um, e expresso como uma fracção (n/d), escreva essa fracção como sendo uma soma de fracções unitárias com denominadores diferentes[6]. Uma fracção unitária é uma fracção em que o numerador é igual a um. O programa a desenvolver deve fazer uso de um algoritmo recursivo.

Seguem alguns exemplos da execução pretendida do programa:

`java -ea UnitaryFractionSum 3 4`	`3/4 = 1/2 + 1/4`
`java -ea UnitaryFractionSum 3 7`	`3/7 = 1/3 + 1/11 + 1/231`
`java -ea UnitaryFractionSum 1 8`	`1/8 = 1/8`
`java -ea UnitaryFractionSum 2 20`	`2/20 = 1/10`

Para resolver o problema considere a seguinte estratégia (designada por "gananciosa" e proposta no Séc. XIII por Fibonacci):

a. Tentar subtrair da fracção a maior fracção unitária possível. Para descobrir essa fracção unitária $(1/d)$ considere a seguinte fórmula:

$$\frac{\text{num}}{\text{den}} - \frac{1}{d} \geq 0 \quad \Leftrightarrow \quad d \geq \frac{\text{den}}{\text{num}} \quad \Rightarrow \quad d = \left\lceil \frac{\text{den}}{\text{num}} \right\rceil$$

b. A fracção será a soma da fracção unitária $1/d$ somada à fracção unitária obtida da fracção resultante da diferença (para a qual se deverá aplicar o mesmo algoritmo);

c. O procedimento termina quando o numerador for divisor do denominador (indicando que já é uma fracção unitária).

[6] Fibonacci demonstrou que qualquer número racional pode ser expresso por uma soma finita de fracções unitárias com denominadores diferentes.