Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Algoritmos e Estruturas de Dados
Semestre de Inverno 2014/15
Segunda série de exercícios

Observações:

- Data de entrega: **28 de Novembro de 2014**.
- Não é permitida a utilização de algoritmos e estruturas de dados já existentes na biblioteca base da plataforma Java.

1 Exercícios

1. Realize o método estático

   `public static int evaluateRPN(String expression)`

   que recebe uma string que corresponde a uma expressão aritmética em notação posfixa e retorna o valor da expressão. Considere apenas os operadores $+$, $-$, $*$, e $/$. Caso a expressão aritmética não tenha o formato descrito anteriormente, deverá ser lançada uma exceção do tipo `IllegalArgumentException`. Além disso, deverá ocorrer uma exceção do tipo `ArithmeticException` caso seja realizada uma divisão por zero.

   A Figura 1 mostra alguns exemplos de expressões usando a notação infixa (convencional) e a notação posfixa.

Infixa	Posfixa	Resultado
1 + 1	1 1 +	2
(7 + 5) * 2	7 5 + 2 *	24
10 / (8 - 3) * 6	10 8 3 - / 6 *	12

Figure 1: Exemplos de expressões algébricas usando notação infixa e posfixa

2. Realize a classe `ListUtils`, contendo os seguintes métodos estáticos:

   2.1. Realize o método estático

   `public static <E> void internalReverse(Node<Node<E>> list)`

   que dada as listas duplamente ligadas, não circulares e sem sentinela, presentes em `list`, inverte a ordem dos seus elementos respectivos. Note que `list` é também uma lista duplamente ligada não circular e sem sentinela. Por exemplo, no caso da lista `list` ser definida por $\{\{1,2,3\},\{4,7,6\},\{3,1,2,4\}\}$, o método deverá transformar a lista `list` em $\{\{3,2,1\},\{6,7,4\},\{4,2,1,3\}\}$.

   2.2. Realize o método estático

   `public static <E> Node<E> intersection(Node<E> list1, Node<E> list2, Comparator<E> cmp)`

   que, dadas as listas duplamente ligadas, circulares e com sentinela, referenciadas por `list1` e `list2`, e ordenadas de modo crescente segundo o comparador `cmp`, retorna uma lista com os elementos que estejam simultaneamente presentes em `list1` e `list2`, removendo-os em ambas as listas. A lista retornada deverá ser duplamente ligada,não circular e sem sentinela, ordenada de modo crescente. Deve ainda reutilizar os nós de uma das listas (`list1` ou `list2`) e não pode conter elementos repetidos.

   Para as implementações destes métodos, assuma que cada objecto do tipo `Node<E>` tem 3 campos: um `value` e duas referências, `previous` e `next`.

3. Realize o método estático

   `public static <E> Iterable<E> distinct(Iterable<E> source, Predicate<E> criterion)`

   que retorna uma sequência com os elementos da sequência ordenada `source` que obedecem ao predicado `criterion`, sem repetições e com a ordem relativa preservada. Considere que, quaisquer que sejam os objectos `o1` e `o2`, se `o1.equals(o2) == true`, então o resultado da aplicação de `criterion` a `o1` é o mesmo da aplicação de `criterion` a `o2`.

4. Considere a seguinte classe:

```
public class Pair<E1,E2>{
  public E1 first;
  public E2 second;
  public Pair(E1 first, E2 second){
    this.first = first;
    this.second = second;
  }
}
```

Realize o método estático,

```
public static <E> Iterable<Pair<E, Integer>> histogram(E[] array)
```

que dada a sequência representada pelo array não ordenado de elementos, retorna um iterável de pares com o histograma da sequência. O algoritmo implementado deve usar uma tabela de dispersão para o cálculo do histograma.

2 Problema: Escalonamento de Processos

No início da era da computação os computadores estavam divididos fisicamente entre a componente de computação e a componente de introdução e recolha de resultados, como mostra a Figura 2.

Pretende-se neste problema modelar um sistema semelhante ao descrito anteriormente, no contexto de uma instituição de ensino superior. Considere que o sistema central de computação apenas consegue executar um processo de cada vez. Em cada momento, existe 1 processo em execução e N numa fila de espera (isto é, submetidos mas ainda não em execução). Após a conclusão do processo em execução, o próximo é escolhido por um de três critérios: ordem de chegada, número de instruções, ou prioridade por classe de utilizador.

Cada processo W_i é caracterizado pelo seu identificador, *pid*; número de instruções (na unidade milhares de instruções), n_i; classe do utilizador que o colocou na fila, c_i; e pelo instante em que foi submetido à fila de espera, t_i. Existem quatro classes de utilizador: *low*, *regular*, *high* e *super*.

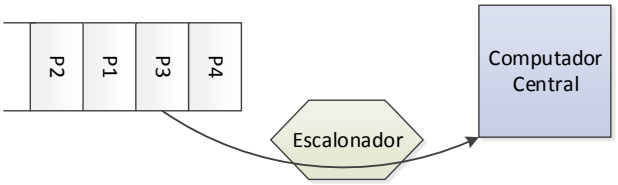

Figure 2: Arquitectura do escalonador

Funcionalidades a implementar

A aplicação a desenvolver deve:

1. Acrescentar novos processos à fila de espera.
2. Determinar e retirar 1 processo da fila de espera, usando uma das três políticas de escalonamento possíveis:
   2.1. Ordem de chegada à fila;
   2.2. Menor número de instruções;
   2.3. Prioridade da classe do utilizador, seguindo a relação de ordem *low* < *regular* < *high* < *super*. Para os processos cujos utilizadores são da mesma classe, o processo com maior prioridade é o que foi submetido primeiro na fila de espera.
3. Mudar a política de escalonamento, para uma das descritas no ponto anterior.
4. Dado um processo, W_i, aumentar a classe do utilizador que lhe está associado.

Parâmetros de execução

Para iniciar a execução terá de ser especificado a política de escalonamento: tempo de submissão (`time`), dimensão do processo (`smallest`) ou prioridade (`prio`).

```
java JobsScheduler time|smallest|prio
```

Durante a sua execução, a aplicação processa os seguintes comandos:

- `add <pid> <n> <c>`, que adiciona à fila de espera o programa com identificador `pid`, com `n` instruções, do utilizador da classe `c`. Este comando corresponde à funcionalidade 1.
- `next`, que corresponde à funcionalidade 2.
- `newpolicy time|smallest|prio`, que corresponde à funcionalidade 3.
- `more <pid> <newc>`, que corresponde à funcionalidade 4.

Tipo de Dados

O escalonador deve usar uma fila de prioridade `PriorityQueue`. A implementação do tipo de dados genérico fila prioritária `PriorityQueue<E>` deverá ser o mais genérica possível de forma a ser utilizada não só no contexto do problema acima descrito, mas também no contexto de outras aplicações. A fila prioritária deverá **pelo menos** suportar as seguintes operações:

1. `public void add(E elem, int prio, KeyExtractor<E> keyExtractor)`, que adiciona o elemento `elem` com prioridade `prio` e chave `keyExtractor.getKey()` à fila;
2. `public E pick()`, que retorna o elemento com maior prioridade presente na fila;
3. `public E poll()`, retorna e remove o elemento com maior prioridade presente na fila;
4. `public void update(int key, int prio)`, que atualiza a prioridade do elemento identificado pela chave `key`;
5. `public void remove(int key)`, que remove o elemento identificado pela chave key.

Considere que a interface `KeyExtractor<E>` é definida da seguinte forma:

```
public interface KeyExtractor<E>{
   public int getKey(E e)
  }
```

O custo de todos este métodos descritos deve pertencer a *O(lg n)*, onde `n` representa o número de elementos na fila.

No contexto do escalonador, note que os identificadores dos processos a executar poderão não ser sequenciais.

Avaliação Experimental

Realize uma avaliação experimental do(s) algoritmo(s) desenvolvido(s) para a resolução deste problema. Apresente os resultados graficamente, utilizando uma escala adequada.

ISEL, 4 de Novembro de 2014.