

**Algoritmos e Estruturas de Dados   
2ª Série**

**(Problema)**

|  |  |
| --- | --- |
|  |  |
|  |  |

**Exercício 2**

|  |  |
| --- | --- |
| 46973 | Jorge Alexandre Luzio Simões |
| 47199 | Tiago Luís Lima da Silva |

Licenciatura em Engenharia Informática e de Computadores  
Semestre de Inverno 2020/2021

16/11/2020

# Introdução

Pretende-se desenvolver uma aplicação que permita realizar um conjunto de operações entre coleções de pontos no plano, nomeadamente as operações de união, interseção e diferença. Cada coleção de pontos encontra-se descrita num ficheiro de texto. Cada ponto é descrito no ficheiro por um identificador e por duas coordenadas, a do X e a do Y, respetivamente. A aplicação *ProcessPointsColections* deverá ter as seguintes características:

* recebe como parâmetro dois ficheiros de texto (com extensão .co);
* permite produzir um novo ficheiro (com extensão .co) com os pontos, sem repetições, que ocorram em pelo
* menos um dos ficheiros de input - operação *union*;
* permite produzir um novo ficheiro (com extensão .co) com os pontos que estejam presentes em ambos os
* ficheiros de input, sem repetições - operação *intersection*;
* permite produzir um novo ficheiro (com extensão .co) com os pontos que estejam presentes apenas em um
* dos ficheiros de input, sem repetições - operação *difference*;

# Maior número de ocorrências

A resolução deste exercício foi divida em três partes analise do problema, estruturas de dados utilizadas e análise de complexidade.

## Análise do problema

Independentemente da operação a ser feita o ficheiro de output terá todos os pontos que respeitam a operação sem repetições, por esta razão é preciso ter uma verificação rápida se já o valor foi lido anteriormente. Para cumprir este objetivo o uso de um *HashMap/HashTable* é necessário visto que a operação de procura através de uma chave é de O (1), tornando assim uma das operações que vamos executar o mais rápido possível. Com esta escolha tambem a inserção será rápida porque a inserção numa *HashMap/HashTable* tambem é de O (1).

Para a chave optamos por criar uma classe auxiliar que tem como atributos x e y e usar então esta classe como chave.

A nível de tamanho do *array* usado no *HashMap/HashTable* existe nos ficheiros de input, na parte do cabeçalho, o número de pontos que existe nesse ficheiro, visto que pode haver pontos repetidos consideramos apenas como tamanho inicial metade deste valor e quando houver necessidade aumentar o tamanho do *array*, devido a o (número de chaves) / (dimensão do *array*) > 1.33, este é aumentado apenas por metade do tamanho inicial.

Finalmente como a complexidade a nível de tempo é igual para o *HashMap* e *HashTable* decidimos pelo *HashMap* para guardar o número de vezes que um ponto se repetiu nos dois ficheiros de input como valor para escrita posterior no ficheiro de output.

## Estruturas de Dados

Como referido anteriormente implementamos um *HashMap* onde a chave de cada entrada é uma classe auxiliar *Point* e o valor é o número de ocorrências desse ponto nos ficheiros de input.

A classe *Point* tem apenas dois atributos, x e y, visto que não fazemos operações com estes atributos estes foram mantidos como *String* e reimplementa os métodos *equals* e *hashcode* para uso da HashMap. A nível da *HashMap* foi usado um array onde cada posição é *HashNode*, classe auxiliar que representa uma lista simplesmente ligada usada para resolução de conflitos de chave para o mesmo índice no *array*.



Figura 1 - Imagem de lista simplesmente ligada.

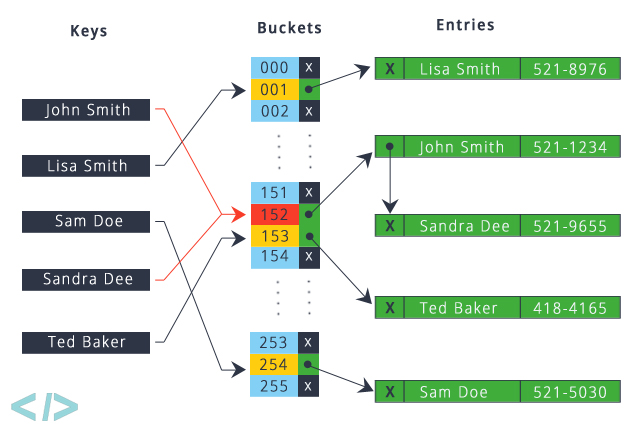


Figura 2 - Imagem de *HashMap* com uso de lista simplesmente ligada para resolução de conflitos.

## Algoritmos e análise da complexidade

Tanto na primeira versão (*HashMap* do java) como na segunda versão (a nossa implementação de *HashMap*) as complexidades são iguais, a diferença de tempo de execução existente deve-se ao número de vezes que cada implementação aumenta o tamanho do *array* interno, visto que é preciso reindexar todos os nós existentes por causa da mudança no tamanho.

Com isto a nível de complexidades de tempo temos:

* Inserção de um novo nó:
* Remoção de um nó:
* Procura de um nó:
* Aumento do *array*: , sendo N o tamanho do *array* antes do aumento.

A nível de complexidade de tempo visto que lemos dois ficheiros, se não houver repetições de pontos em nenhum ficheiro, a complexidade é de:

Sendo N o maior ficheiro dos dois inputs.

Em termos de complexidade de espaço no caso no ficheiro N não haver repetições, no ficheiro M não haver repetições e a interseção de N e M ser nula:

* União: , sendo N e M o número pontos existentes nos ficheiros de input.
* Interseção: , sendo N o menor número de pontos entre os ficheiros de input.
* Diferença: , sendo N o número de pontos entre do primeiro ficheiro de input.

Já no caso no ficheiro N não haver repetições, no ficheiro M não haver repetições e a interseção de N e M ser diferente de nulo:

* União:
* Interseção:
* Diferença:

Sendo N e M o número pontos existentes nos ficheiros de input.

# Avaliação Experimental

A avaliação experimental foi feita com os seguintes parâmetros:

* Mudanças de ficheiros de input.
* Cada algoritmo (u*nion, intersection e difference*) foi corrido cinco vezes para cada conjunto de ficheiros de input.

Comparação da implementação com as classes do Java e a nossa implementação da *HashMap*.

Tabela 1 - Tempos em milissegundos do algoritmo *Union* da implementação em Java.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Union | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 3149.66 | 10514.21 | 7474.48 | 135024.2 | 139988.4 |
| 2 | 3071.93 | 10374.32 | 7291.72 | 160970.1 | 146444.3 |
| 3 | 3245.42 | 9882.6 | 7690.65 | 155442.2 | 142859.2 |
| 4 | 3150.66 | 10392.75 | 7283.58 | 148629.2 | 148960.5 |
| 5 | 3198.48 | 10149.15 | 7961.31 | 141160.1 | 148035.5 |
| Average | 3163.23 | 10262.61 | 7540.348 | 148245.2 | 145257.6 |

Tabela 2 - Tempos em milissegundos do algoritmo *Difference* da implementação em Java.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Difference | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 2029.06 | 8388.69 | 4179.8 | 54995.52 | 131762.2 |
| 2 | 2016.57 | 8844.64 | 4522.26 | 53150.46 | 131234.1 |
| 3 | 2050.61 | 8676.4 | 4263.64 | 53222.98 | 132165.7 |
| 4 | 2050.14 | 8517.25 | 4463.9 | 53873.15 | 135026.4 |
| 5 | 2112.64 | 8692.37 | 4224.31 | 54042.32 | 133990.8 |
| Average | 2051.804 | 8623.87 | 4330.782 | 53856.89 | 132835.8 |

Tabela 3 - Tempos em milissegundos do algoritmo *Intersection* da implementação em Java.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Intersection | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 1118.09 | 7851.46 | 2480.49 | 65647.78 | 62567.09 |
| 2 | 1122.17 | 7930.98 | 2429.91 | 66682.55 | 65680.62 |
| 3 | 1130.31 | 7935.21 | 2524.49 | 66558 | 61824.87 |
| 4 | 1118.36 | 7958.32 | 2527.4 | 68065.84 | 66117.09 |
| 5 | 1106.09 | 7940.62 | 2465.7 | 65655.11 | 69850.3 |
| Average | 1119.004 | 7923.318 | 2485.598 | 66521.86 | 65207.99 |

Tabela 4 - Tempos em milissegundos do algoritmo *Union* da nossa implementação.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Union | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 3328.77 | 10874.19 | 8396.36 | 151596.9 | 154369.4 |
| 2 | 3355.06 | 9822.13 | 8156.89 | 147882.3 | 153871.4 |
| 3 | 3244.81 | 10038.89 | 8922.5 | 149931.4 | 155593.1 |
| 4 | 3172.41 | 10581.61 | 8562.27 | 142609.6 | 166369 |
| 5 | 3192.8 | 10038.73 | 7832.88 | 151291.6 | 161281.3 |
| Average | 3258.77 | 10271.11 | 8374.18 | 148662.3 | 158296.8 |

Tabela 5 - Tempos em milissegundos do algoritmo *Difference* da nossa implementação.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Difference | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 2064.97 | 9110.27 | 4314.1 | 58413.85 | 136996.8 |
| 2 | 2058.9 | 9208.12 | 4572.4 | 56190.18 | 140005 |
| 3 | 2033.04 | 9011.21 | 5267.37 | 55540.09 | 140043.1 |
| 4 | 2239.3 | 9099.12 | 5023.77 | 55528.14 | 143161.1 |
| 5 | 2011.61 | 9155.05 | 4376.89 | 58368.24 | 148457.7 |
| Average | 2081.564 | 9116.754 | 4710.906 | 56808.1 | 141732.8 |

Tabela 6 - Tempos em milissegundos do algoritmo *Intersection* da nossa implementação.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Intersection | F1 + F2 | F1r + F2r | F3 + F4 | F4r + F6r | F6r + F4r |
| 1 | 1117.68 | 8243.48 | 2543.32 | 76548.65 | 73897.83 |
| 2 | 1125.45 | 8138.17 | 2517.53 | 74631.15 | 73178.47 |
| 3 | 1120.44 | 8252.02 | 2518.7 | 75374.46 | 70972.34 |
| 4 | 1121.47 | 8211.3 | 2660.26 | 76151.28 | 70800.54 |
| 5 | 1121.94 | 8195.56 | 2533.78 | 74903.41 | 71118.54 |
| Average | 1121.396 | 8208.106 | 2554.718 | 75521.79 | 71993.54 |

# Conclusões

Podemos que concluir que a nossa implementação por estar só a considerar apenas metade do número de pontos existentes no ficheiro, torna-se mais lenta quando não existe metade dos pontos repetidos. Em comparação a implementação do *HashMap* do java a quantidade de vezes que nos aumentamos o tamanho do *array* quando necessário é maior visto que a nosso cálculo de quanto de dimensão aumentar é muito mais simples que a versão do java correndo o risco de ser necessário aumentar mais vezes do que o necessário.