UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

# Relatório do Projeto 4 – Tabelas de Dispersão

Trabalho realizado no âmbito da cadeira Algoritmos e Estruturas de Dados por:
Diogo Almeida, nº: 81477
Helena Vassal, nº: 75525
Grupo: PL2 – G16
Docente: Dr. João Dias

**Neste relatório, documentamos os resultados da investigação realizada sobre o Projeto 4.**

## Projeto 4: ForgettingCuckooHashTable

O projeto 4 tem como objetivo a implementação de um dos componentes usado pelo sistema *Monolith* (sistema de recomendações usado pelo TikTok), uma tabela de dispersão implementada pelos engenheiros da *ByteDance*, baseada no conceito de *Cuckoo Hashing* (dispersão de Cuco). As principais diferenças entre a nossa tabela de dispersão de Cuckoo face a uma tabela de dispersão tradicional é:

1) A existência de duas tabelas[1] e duas funções de hash, que são usadas para dispersar as chaves nas respetivas tabelas.
2) O tratamento de colisões. Numa tabela tradicional guardaríamos várias chaves numa posição da tabela no acontecimento de uma colisão, ou calcularíamos um *offset* com uma segunda função de hash para guardar a chave na mesma tabela. No caso da *ForgettingCuckooHashTable*, numa colisão, introduzimos a chave na posição inicialmente calculada da T0 ou T1 e a chave que ocupava a posição é enviada para a outra tabela, repetindo o processo, até encontrar uma posição livre.
3) A funcionalidade de esquecimento, na eventualidade de uma chave não ser interessante, tratamos esta chave como se não existisse, no caso de uma colisão com a mesma, guardamos a chave que estamos a inserir na posição da chave esquecida e não nos preocupamos com a inserção da mesma na outra tabela.

## Testes realizados:

O primeiro teste realizado foi sobre a qualidade das diferentes funções de hash utilizadas, testámos também o impacto da funcionalidade de esquecimento de chaves no número médio de trocas e variância e fizemos uma análise da eficiência temporal da tabela de dispersão implementada, mais especificamente dos métodos de busca e inserção.

1) Para a análise das funções de hash, gerámos exemplos de tabelas com 100, 1000 e 100 000 chaves aleatórias, calculámos os valores do nº de trocas médias e de variância das últimas 100 inserções de chaves com as diferentes funções de hash;
2) Para o impacto da funcionalidade de esquecimento, construímos um teste em que 20% das chaves são interessantes e 80% não, fizemos os cálculos mencionados no ponto 1, e ainda, o tempo médio de execução do método de inserção;
3) Para a análise da eficiência temporal, fizemos testes de razão dobrada para determinar a complexidade temporal dos métodos de pesquisa e inserção e comparámos o tempo médio de execução com uma tabela de endereçamento aberto com exploração linear.

[1] As tabelas 0 e 1 serão referenciadas em todo o documento como T0 e T1 e as funções de hash como h0 e h1.

## 1) Análise da qualidade das funções hash

As funções de hash são responsáveis por determinar a posição de uma entrada nas tabelas de dispersão, por isso, são extremamente importantes para o bom funcionamento e eficiência da mesma. É essencial que as funções de hash que implementemos sejam consistentes, eficientes, e que dispersem de forma uniforme as chaves por **todas** as posições disponíveis nas tabelas, de forma a minimizar a quantidade de colisões.

Inicialmente, criámos uma versão básica de h0 e h1. Estas funções utilizam o método *hashCode()* para retornar um inteiro resultante do cálculo de um algoritmo de hashing aplicado ao objeto (neste caso, uma chave) e aplicámos o módulo desse valor hash com um valor p, que é um número primo menor que a capacidade da tabela atual.

```
private int h0(Key key) {
    int p = primesTable0[capacityIndex-1];
    return key.hashCode() % p;
}

private int h1(Key key) {
    int p = primesTable1[capacityIndex-1];
    return key.hashCode() % p;
}
```

*Figura 1 - Versão Inicial das funções de hash*

Rápidamente percebemos que existiam alguns problemas com esta versão. Primeiro, no caso do valor do *hashCode()* ser negativo, teríamos uma exceção *IndexOutOfBounds*. Segundo, o valor de 'p' ser menor que a capacidade da tabela limitava os índices disponíveis ao valor de 'p', devido ao uso do módulo. Para contornar isto, fizemos o seguinte:

Primeiro, usámos a operação bit a bit *"& 0x7fffffff"*, que, muito resumidamente, remove o bit mais significativo (bit de sinal), de forma a assegurar que o resultado seja sempre positivo. Depois, trocámos o 'p' pela capacidade da respetiva tabela, garantindo que todas as posições das tabelas são válidas para a inserção de chaves.

```
// Hashing functions
private int h0(Key key) {
    return (key.hashCode() & 0x7fffffff) % capacityTable0;
}

private int h1(Key key) {
    return (key.hashCode() & 0x7fffffff) % capacityTable1;
}
```

*Figura 2 - Segunda versão das funções de hash*

A segunda versão de h0 e h1 é válida e poderia ser usada por todo o projeto, no entanto, decidimos fazer uma melhoria para a versão final de forma a dispersar melhor as chaves ao diferenciar mais h0 e h1.

Para isso, usámos o operador de negação bit a bit '~' (til), este operador nega todos os bits do hashcode, diferenciando h0 e h1 e resultando numa melhor dispersão:

```java
// Hashing functions
private int h0(Key key) {
    return (key.hashCode() & 0x7fffffff) % capacityTable0;
}

private int h1(Key key) {
    return (~key.hashCode() & 0x7fffffff) % capacityTable1;
}
```

*Figura 3 - Versão final das funções de hash*

Esta é a versão final de h0 e h1. Usámos o operador na função h1 pois é menos usada e a eficiência é menos afetada face à utilização do operador em h0.

## Testes realizados:

Para analisar a qualidade das funções de *hash*, criámos exemplos de tabelas com diferentes tamanhos. Utilizámos as diferentes funções de *hash* e calculámos a média do número de trocas entre tabelas e a variância para as últimas 100 inserções de chaves.

## O que esperávamos:

Tendo em conta as falhas e melhorias de cada função de *hash*, esperávamos que a versão inicial falhasse na maioria dos testes, não só na média e variância mas também na funcionalidade. Entre a segunda e última versão das funções, esperávamos uma pequena melhoria nos resultados, mantendo a funcionalidade.

## Resultados obtidos:

Os resultados confirmaram a nossa análise, os melhores resultados foram obtidos pela versão final das funções de *hash*. A segunda versão passou nos testes, no entanto, com valores maiores. E a primeira versão, com valores altíssimos para o primeiro teste e a sua falha na funcionalidade não permitiu que fosse realizado o segundo e terceiro teste.

| ---------- | Initial Version | | Second Version | | Final Version | |
|---|---|---|---|---|---|---|
| Complexity: | Average | Variation | Average | Variation | Average | Variation |
| 100 | 1,95 | 20,65 | 1,04 | 4,2 | 0,994 | 2,68 |
| 1000 | ---------- | ---------- | 0,84 | 0,23 | 0,79 | 0,22 |
| 100000 | ---------- | ---------- | 0,19 | 0,011 | 0,16 | 0,0016 |

*Figura 4 - Resultados obtidos dos testes de média do nº de trocas e variância para diferentes funções de hash*

A falha nos testes da versão inicial provém maioritariamente do número de trocas ser alto, este acontecimento desencadeia uma série de operações, tais como o redimensionamento da tabela, inúmeras vezes, alcançando os limites de tamanho predefinidos e lançando uma exceção *IndexOutOfBounds*.

## 1) Conclusão:

A análise das funções de *hash* revelou a importância crítica destas funções para o desempenho e eficiência de tabelas de dispersão. A versão inicial apresentou problemas, como a limitação dos índices válidos, esta falha resultava em grandes aglomerados de chaves, que por sua vez levou a mais trocas e maiores valores de variância.

A versão final das funções de *hash* demonstrou resultados mais eficazes nos testes, com uma dispersão melhorada das chaves. A comparação com as versões anteriores confirmou a eficácia das melhorias, evidenciando uma redução significativa nos valores de trocas e uma maior estabilidade funcional face à versão inicial.

Os testes reforçaram a importância de considerar a qualidade das funções de *hash* na implementação de estruturas de dados como tabelas de dispersão.

## 2) Impacto do esquecimento do número de trocas

A funcionalidade do esquecimento é uma ideia chave das tabelas de dispersão utilizadas no *Monolith*. É também de certa forma um conceito simples, fácil de implementar.

Para isto, guardámos um selo temporal junto com cada chave, que dita o tempo de vida da mesma. Se a chave for acedida de alguma forma, seja por uma pesquisa ou atualização do seu valor, o seu tempo de vida (selo temporal) é reiniciado. Caso não seja acedida durante 24 horas, torna-se desinteressante e na eventualidade de necessitarmos do espaço que esta chave ocupa (numa colisão), podemos simplesmente ignorá-la e guardar a nova chave no lugar da chave esquecida.

Esta funcionalidade visa reduzir o número de trocas entre tabelas e manter nas mesmas, só a informação importante (chaves interessantes), mitigando assim problemas de memória a longo prazo e reduzindo o tempo de inserção.

### Testes realizados:

Para analisar o impacto da funcionalidade de esquecimento, criámos um teste com 50000 chaves, em que 20% das chaves são interessantes (acedidas) e 80% das chaves são pouco ou nada acedidas e o tempo vai avançando, calculámos o número médio de trocas, a variância e o tempo médio de execução do put a cada 1000 puts, por fim, comparámos estes valores com o mesmo teste, no entanto, com uma versão sem a funcionalidade de esquecimento.

### O que esperávamos:

A funcionalidade de esquecimento é sem margem de dúvidas uma melhoria significativa face à não existencia da funcionalidade. Esperamos que para os testes, a versão com esquecimento seja superior em todos os testes, menos trocas, o que implicita menos colisões, uma menor variância (melhor dispersão) e um tempo de execução menor.

### Resultados obtidos:

Tal como previsto, a versão sem esquecimento teve os piores resultados, o tempo de execução médio do método put foi quase o dobro do tempo das versões com esquecimento. O número médio de trocas subiu 55% em relação à versão com esquecimento de 2 horas por 1000 puts e a média de variância quase quadriplicou face às duas versões com esquecimento.

| H * 1000 PUTS | Average MS/ 1000 PUTS | Average Number of Swaps | Average Variation |
|---|---|---|---|
| 0 | 0,441 | 0,88 | 9,70E-01 |
| 1 | 0,294 | 0,661 | 2,10E-01 |
| 2 | 0,277 | 0,573 | 2,00E-01 |

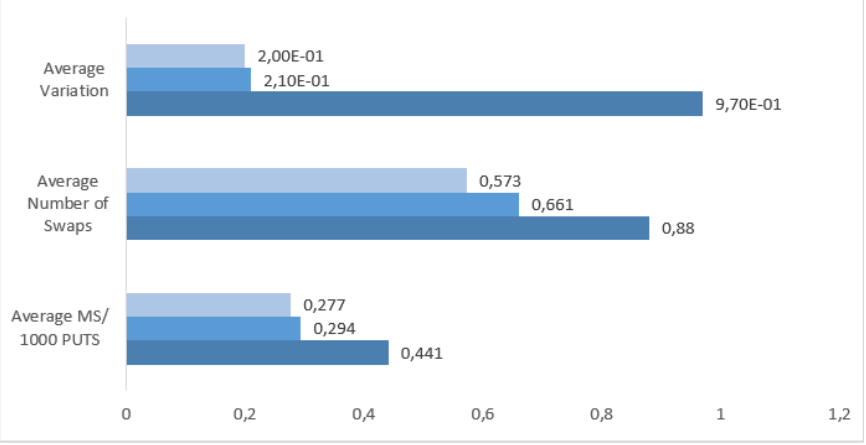


*Figura 5 - Testes realizados sobre a qualidade das funções de hash*

## 2) Conclusão:

Os testes feitos sobre o impacto do esquecimento do número de trocas revelaram resultados significativos e esclarecedores. A funcionalidade de esquecimento implementada nas tabelas de dispersão do *Monolith* apresentou melhorias substanciais em comparação à versão sem a mesma funcionalidade.

Os resultados indicaram que a operação de inserção nas tabelas com esquecimento é significantemente mais eficiente do que a versão sem o esquecimento, foram necessárias realizar menos trocas e a variância foi menor, indicando uma melhor dispersão das chaves.

Estes resultados confirmam a eficácia da funcionalidade. Assim, a conclusão é que a implementação da mesma é uma melhoria significativa, resultando num desempenho mais eficiente do sistema como um todo.

## 3) Análise da Eficiência Temporal

Nesta última alínea pretende-se analisar a eficiência temporal da tabela de dispersão implementada. A complexidade temporal de uma tabela de dispersão, quando as funções de dispersão dispersam de forma uniforme as chaves pelas posições da tabela, é da ordem de O(1). O objetivo é confirmar este resultado empiricamente, fazendo testes de razão dobrada para os métodos *get* e *put*.

Vamos também determinar o tempo médio de execução dos mesmos métodos, e comparar com o tempo médio dos mesmos métodos de uma implementação alternativa de uma tabela de endereçamento aberto com exploração linear.

### 3.1) Testes de razão dobrada do método Put:

#### O que esperávamos:

O método put é eficiente, se e só se, as funções de dispersão dispersam de forma uniforme as chaves por todas as posições da tabela, pois isto indica que independentemente de onde vou inserir a chave, é esperado que terei de fazer o mesmo número de trocas que faria em qualquer outra posição da tabela. Ou seja, num caso em que insiro 100 chaves e a média

do número de trocas é 1, então, o *put* vai custar, aproximadamente, 1 inserção direta numa tabela e uma inserção na outra tabela, o que indica, que a complexidade temporal do método é O(1), sendo constante, independentemente do tamanho da tabela.

### Resultados obtidos:

Ao realizar os testes de razão dobrada confirmámos a nossa análise. O tempo levado a realizar um *put* é constante independentemente do tamanho N da tabela. E a sua complexidade temporal é de facto, da ordem de O(1).

| i | complexity | time(ms) | estimated r |
|---|---|---|---|
| 0 | 125 | 0 | --- |
| 1 | 250 | 0 | 0 |
| 2 | 500 | 0 | 0 |
| 3 | 1000 | 0 | 0 |
| 4 | 2000 | 0 | 0 |
| 5 | 4000 | 0 | 0 |
| 6 | 8000 | 0 | 0 |
| 7 | 16000 | 0 | 0 |
| 8 | 32000 | 0 | 0 |
| 9 | 64000 | 0 | 0 |
| 10 | 128000 | 0 | 0 |
| 11 | 256000 | 0 | 0 |
| 12 | 512000 | 0 | 0 |
| 13 | 1024000 | 0 | 0 |

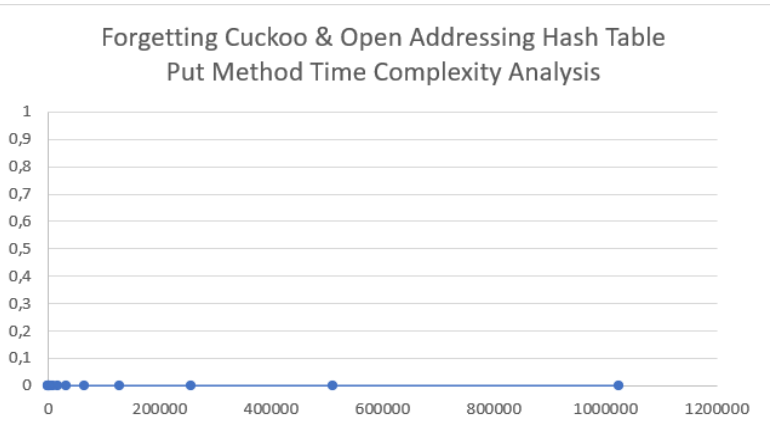


*Figura 6 - Resultados dos testes de razão dobrada do método put para ambas as tabelas de endereçamento*

## 3.2) Testes de razão dobrada do método Get:

### O que esperávamos:

O método Get é extremamente eficiente independentemente do tamanho da tabela, pois é um método que **só realiza operações de tempo constante**, por isso, sabemos que a sua complexidade temporal é da ordem de O(1), de qualquer forma, realizámos testes empíricos para confirmar esta análise.

### Resultados obtidos:

Os resultados obtidos asseguram a veracidade da análise feita anteriormente. O tempo levado a realizar um *get* é constante independentemente do tamanho N da tabela. E a sua complexidade temporal é da ordem de O(1).

| i | complexity | time(ms) | estimated r |
|---|---|---|---|
| 0 | 125 | 0 | --- |
| 1 | 250 | 0 | 0 |
| 2 | 500 | 0 | 0 |
| 3 | 1000 | 0 | 0 |
| 4 | 2000 | 0 | 0 |
| 5 | 4000 | 0 | 0 |
| 6 | 8000 | 0 | 0 |
| 7 | 16000 | 0 | 0 |
| 8 | 32000 | 0 | 0 |
| 9 | 64000 | 0 | 0 |
| 10 | 128000 | 0 | 0 |
| 11 | 256000 | 0 | 0 |
| 12 | 512000 | 0 | 0 |
| 13 | 1024000 | 0 | 0 |

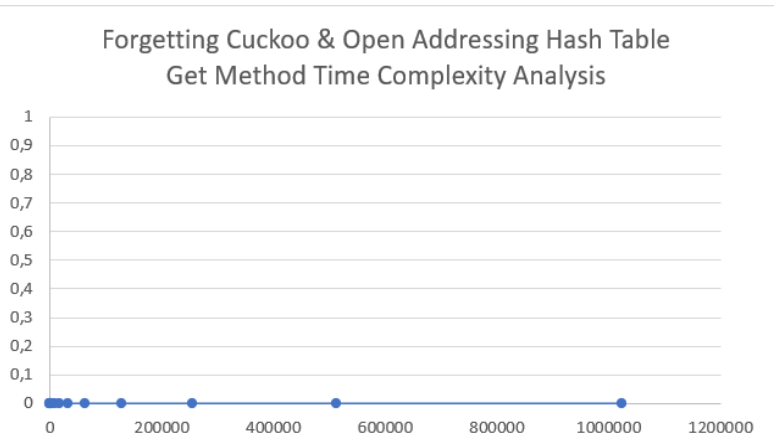


*Figura 7 - Resultados dos testes de razão dobrada do método get para ambas as tabelas de endereçamento*

## 3.3) Testes do tempo de execução médio:

### O que esperávamos:

Para ambos os métodos testados, esperávamos que a nossa implementação tivesse tempos de execução menores contra a implementação com exploração linear, o *Cuckoo Hashing* tem o potencial de ter uma constante de tempo baixa para estas operações, pois minimiza as colisões.

Esperamos que o aumento do *trigger* de redimensionamento da tabela aumente de forma significativa os tempos de execução de ambas as tabelas pois isto implica que o fator de carga chegará a níveis maiores, resultando em mais colisões.

### Resultados obtidos:

Os resultados indicam que de facto, a implementação da tabela de Cuckoo é mais eficiente em comparação com a tabela de endereçamento aberto. Surpreendentemente para nós, a maior diferença ocorreu no método de inserção, a tabela de endereçamento aberto teve um tempo de execução médio de aproximadamente o dobro, contra a tabela de Cuckoo.

| Resize Load Factor Trigger: | 50% of Table Size | | 75% of Table Size | |
|---|---|---|---|---|
| Methods: | GET | PUT | GET | PUT |
| Forgetting Cuckoo Hash Table | 0,0097 ms | 0,0148 ms | 0,0099 ms | 0,0152 ms |
| Open Addressing Hash Table | 0,0134 ms | 0,0279 ms | 0,0132 ms | 0,0287 ms |

*Figura 8 - Resultados sobre o tempo médio de execução obtidos para as duas tabelas de dispersão*

Também podemos dizer com certeza que o aumento do *trigger* de redimensionamento da tabela afetou negativamente os tempos de execução média de ambas as tabelas, com uma mudança mais significativa na tabela de endereçamento aberto.

## 3) Conclusão:

Com base na análise da eficiência temporal realizada, podemos concluir seguramente que os métodos *put* e *get*, métodos de inserção e pesquisa de uma tabela de dispersão, a nossa inclusive, tem uma complexidade temporal de ordem O(1). No entanto, após realizar os testes de tempo de execução médio de ambos os métodos com as duas implementações de tabela de dispersão diferentes, concluimos também que a nossa implementação, é significantemente mais eficiente nestas operações.

## Projeto 4 – Conclusão Final

A conclusão final do Projeto 4 é que a tabela de dispersão ForgettingCuckooHashTable, com características específicas, tais como a existência de duas tabelas, duas funções de dispersão, tratamento de colisões baseado no conceito de *Cuckoo Hashing* e a funcionalidade de esquecimento, demonstrou ser uma estrutura de dados eficaz e eficiente.

Esta implementação provou não só ser eficaz e eficiente mas também, **mais** eficaz e eficiente que uma tabela de dispersão de endereçamento aberto com exploração linear, que por si só, já é uma implementação bem estabelecida e muito usada para resolver colisões em tabelas de dispersão.

Em suma, a implementação da ForgettingCuckooHashTable demonstrou ser uma solução sólida e *state of the art*, o que é de esperar, pois é uma das componentes chave do sistema de recomendações do *TikTok*, o *Monolith*.