UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2023-2024

Projeto 4 – Tabelas de Dispersão – versão (1.1)

TualgTok

Introdução

"É parvo estamos a aprender a implementar estas estruturas de dados complicadas quando a linguagem Java já implementa isto tudo!". Citação atribuída a um aluno imaginário que provavelmente reprovou a algoritmos e estruturas de dados.

A afirmação acima está errada, pois mais tarde ou mais cedo, um bom engenheiro informático vai ter de implementar uma estrutura de dados específica para um determinado problema que está a tentar resolver. Um bom exemplo disto, e que serviu como inspiração para este projeto, é o sistema Monolith[1], o sistema de recomendações usado pelo *TikTok*.

Neste projeto iremos implementar um dos componentes usados pelo sistema, uma tabela de dispersão customizada implementada pelos engenheiros da *ByteDance*, baseada no conceito de *Cuckoo Hashing (dispersão de Cuco)* e que é usada para guardar uma tabela de representações vetoriais (*embeddings*) esparsa.

Antes de entrarmos em pormenores, é importante perceberem que uma tabela de dispersão é uma estrutura de dados muito apropriada para guardar uma tabela esparsa. Uma tabela esparsa é uma tabela onde a maior parte dos seus elementos não existe, ou tem um valor não relevante como zero. Se por exemplo, no caso do *TikTok* quisermos guardar o quanto cada utilizador gosta de cada vídeo, não seria possível criar uma tabela bidimensional tradicional para guardar esta informação, pois teríamos qualquer coisa como 1000 milhões de utilizadores e milhões de vídeos. Se multiplicarem estas duas dimensões vão ver que não seria possível guardar esta tabela completa em memória.

[1] Liu Z. et al. (2022). Monolith: Real Time Recommendation System with Collisionless Embedding Table.

Nestes casos, quando as tabelas são esparsas (no caso do *TikTok* cada utilizador classifica um número limitado de vídeos) utilizam-se tabelas de dispersão, usando uma combinação do id do utilizador e do id do vídeo como chave.

Problema A: *ForgettingCuckooHashTable*

Neste problema pretende-se implementar em Java uma tabela de dispersão equivalente à usada no sistema *Monolith*. A primeira das ideias chave da tabela de dispersão utilizada é a técnica de cuco usada para lidar com colisões de chaves.

O nome de técnica de cuco advém da ave cuco, que coloca o seu ovo num ninho alheio, e que para melhor enganar as outras aves, chega inclusivamente a roubar, ou jogar fora um dos ovos já existentes. Usando esta ideia como inspiração, uma tabela de dispersão com a técnica de cuco funciona da seguinte forma: em vez de uma única tabela de endereçamento aberto, temos duas tabelas de endereçamento aberto (T0, e T1), com aproximadamente metade do tamanho cada uma. Usamos também duas funções de dispersão (*hash functions*) diferentes. A função *h0* é usada para determinar a posição de uma chave na tabela *T0*, e a função *h1* é usada para determinar a posição de uma chave na tabela *T1*.

A operação de inserção é efetivamente a operação mais complicada desta tabela de dispersão, e pode ser mais pesada que as operações de inserção em tabelas de dispersão convencionais, mas o lado positivo é que a operação de pesquisa se torna extremamente eficiente. Quando inserimos uma chave, começamos por aplicar a função h0 e determinar a posição para a chave na tabela T0, se a posição determinada estiver livre então inserimos e retornamos (neste caso isto é igual a uma inserção numa tabela de dispersão convencional). No entanto, se a posição em T0 já estiver ocupada, iremos então aplicar uma abordagem semelhante à da ave de Cuco. Colocamos a nova chave no lugar indicado por h0, e tiramos a chave que lá estava anteriormente, para ser colocada na tabela *T1*, usando a função de dispersão *h1*. Pode também acontecer, que ao tentar colocar a chave antiga em *T1*, também esteja outra chave na posição desejada. Não há problema, aplicamos novamente a técnica do cuco, mas desta vez a chave que lá estava é colocada na tabela T0, usando a função de dispersão *h1*. A figura 1 ilustra um exemplo de inserção de chaves que dá origem a várias operações de Cuco. Ao tentar colocar a chave A em T0, temos de mover a chave B para T1. Ao colocar a chave B em T1, temos de mover a chave C para T0. Ao mover a chave C para T0, temos de mover a chave D para T1. Finalmente, quando tentamos colocar D em T1, apercebemo-nos de que o espaço para D em T1 está livre, e por isso podemos parar.

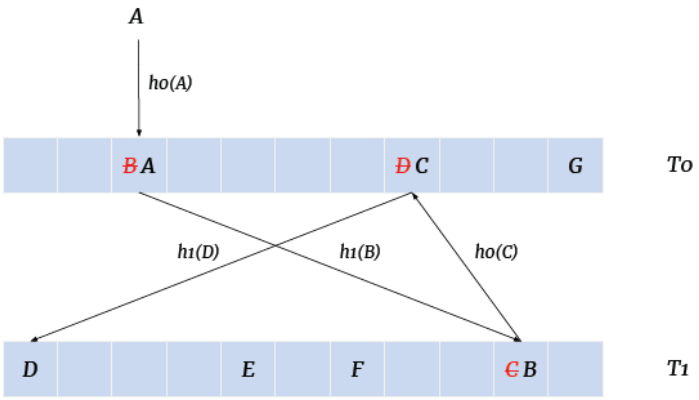

Figura 1 – Inserção de chaves baseada na técnica de Cuco (imagem retirada de Liu et al. 2022)

Infelizmente, pode acontecer ainda um problema adicional, que é este processo de andar a trocar as chaves de tabela entrar num ciclo muito grande, ou até mesmo infinito. Portanto é definido um limite, i.e. número máximo de trocas, e quando o número máximo de trocas é atingido, quer dizer que provavelmente o tamanho atual das tabelas usadas não é grande o suficiente para evitar colisões, e não temos hipótese que não seja aumentar o tamanho das tabelas (teremos de fazer reinserção de todas as chaves) e tentar novamente inserir a chave depois de aumentar de tamanho.

Se a inserção de uma nova chave é um processo um pouco complexo, a sua pesquisa é trivial. Uma chave *k* existe numa tabela de dispersão Cuco se estiver na tabela T0, na posição dada pela função h0, ou se estiver na tabela T1, na posição dada pela função h1. Caso contrário, a chave não existe.

A remoção é igualmente trivial, procuramos pela chave, e se encontrarmos a chave basta apagá-la de onde a encontrámos.

1) Construtores

Implemente os construtores:

`ForgettingCuckooHashTable(int primeIndex)`
Construtor que recebe qual o tamanho inicial a usar para as tabelas, através de um índice da tabela de primos.
`ForgettingCuckooHashTable()`
Construtor que constroi uma tabela de dispersão com o tamanho mínimo possível.

A tabela de dispersão deverá usar como tamanho os valores primos especificados no ficheiro fornecido.

2) Métodos básicos

Implemente os seguintes métodos:

`int`	`size()`
Retorna o tamanho da tabela, ou seja, o número de chaves guardadas.	
`boolean`	`isEmpty()`
Retorna verdade se a tabela estiver vazia, e falso caso contrário.	
`int`	*getCapacity()*
Retorna a capacidade máxima da tabela de dispersão.	
`Float`	`getLoadFactor()`
Retorna o fator de carga, ou seja a razão entre o número de chaves e a capacidade máxima da tabela.	

3) Inserção

Implemente o método de inserção, como indicado acima. No entanto, não se esqueça de que o método de *put* pode funcionar simplesmente como atualização de uma chave já existente. Caso a chave recebida já exista na tabela, este método não é considerado uma inserção, mas sim uma atualização.

`void`	`put(Key k, Value v)`
Insere o valor v na tabela de dispersão associando-o à chave k. A chave K não pode ser nula. Caso seja nula é lançada uma IllegalArgumentException. Caso a chave já exista na tabela é feita uma atualização ao seu valor. Inserir o valor null numa tabela de dispersão é equivalente a fazer delete(k). Esta tabela não suporte a adição de 3 chaves com o mesmo hashcode. Se ao inserir uma nova chave, a tabela detetar que o hashcode é igual a outras duas chaves já inseridas, é lançada uma excepção do tipo IllegalArgumentException.	

Adicionalmente, a capacidade da tabela de dispersão deverá ser ajustada automaticamente em função do fator de carga. Antes de inserir um novo valor (no caso de uma inserção), caso o fator de carga seja >= 0.5, devemos redimensionar a capacidade da tabela para o próximo valor (aproximadamente o dobro).

Existe também uma limitação desta tabela de dispersão, que é o facto de a tabela não suportar mais três ou mais chaves com o mesmo código de dispersão (hashcode). Quando é inserida uma nova chave que tem o mesmo código de dispersão que outras duas já existentes, esta situação deve ser detetada e ser lançada uma exceção do tipo IllegalArgumentException com uma mensagem de erro ilustrativa. Se não o fizéssemos iriamos obter um ciclo infinito de trocas, que não é possível ser resolvido nem mesmo com um redimensionamento para cima da tabela.

Felizmente, é fácil detetar esta situação. Quando inserimos uma nova chave, e vamos buscar as chaves que já estão na tabela T0, e na tabela T1 nas posições correspondentes à nova chave recebida, e se detetarmos que embora não iguais as 3 têm o mesmo código de dispersão, podemos parar imediatamente e lançar a exceção.

4) Pesquisa e remoção

Implemente os seguintes métodos de pesquisa e remoção:

`boolean containsKey(Key k)`
Dada uma chave k, retorna verdade se a chave estiver guardada na tabela, e falso caso contrário.
`Value get(Key k)`
Dada uma chave, retorna o valor guardado juntamente com a chave, caso a chave esteja na tabela. Caso a chave não exista na tabela, é retornado *null*.
`void delete(Key k)`
Apagar a chave (e respetivo valor) da tabela de dispersão caso ela exista. Caso não exista nada acontece.

Após a remoção de um valor, caso o fator de carga seja <0.125 (1/8), devemos redimensionar a capacidade da tabela para o valor anterior na tabela de primos. No entanto, a capacidade nunca poderá ser inferior à capacidade mínima.

5) Iterador

Implemente um iterador iterável para percorrer todas as chaves da tabela de dispersão. A ordem das chaves. Não é especificada a ordem pela qual o iterador deve percorrer os elementos.

`Iterable<Key> keys()`
Devolve um objecto iterável que pode ser usado para percorrer (através de um foreach) todas as chaves da tabela de dispersão.

6) análise da qualidade das funções de dispersão (hash)

A utilização de funções de hash h0 e h1 apropriadas é muito importante para o correto funcionamento da tabela de dispersão com técnica de cuco. Em particular é muito importante que as duas funções de hash dispersem as chaves de forma pelas duas tabelas T0 e T1, mas também que o façam de forma muito diferente. Por exemplo, se para uma dada chave k, h0(k) = h1(k), isto será péssimo para o funcionamento da tabela, e dará origem a muitas operações de trocas.

Para conseguir analisar a qualidade das duas funções de hash a trabalhar em conjunto, iremos calcular a média e a variância do número de operações de troca efetuadas por inserções de novas chaves. Para isso,

devemos alterar o método *put*, de forma a guardar o número de trocas (n.º de vezes em que uma chave mudou de tabela) das últimas 100 inserções de novas chaves. No entanto isto só deverá ser feito caso seja chamado o método que ative o processo de registo (logging). Por omissão o processo de registo de guardar as trocas das últimas 100 inserções está desligado.

Depois implemente os seguintes métodos:

`void setSwapLogging(boolean state)`
Recebe um valor booleano (verdadeiro ou falso). Se for verdadeiro ativa os registos do número de trocas quando são feitas inserções de novas chaves. Se for falso, desativa o registo.
`float getSwapAverage()`
Devolve o número médio de trocas de chave efetuadas pelas últimas 100 inserções de novas chaves. Caso não existam 100 chaves inseridas, retorna o número médio de trocas de chave das inserções efetuadas. Caso o registo de trocas esteja desativado, retorna 0.
`float getSwapVariation()`
Devolve a variância de trocas de chave efetuadas pelas últimas 100 inserções de novas chaves. Caso não existam 100 chaves inseridas, retorna variância de trocas de chave das inserções efetuadas. Caso o registo de trocas esteja desativado, retorna 0.

Atenção que caso a inserção de uma chave corresponda a uma atualização, essa atualização não deve ser guardada como uma das últimas 100 inserções de novas chaves.

Depois de implementados os métodos, analise a qualidade das duas funções de *hash* escolhidas pelo seu grupo, gerando exemplos com chaves aleatórias, e calculando os valores acima. Em particular a análise da variância é muito importante. Valores baixos de variância são desejáveis, e valores altos muito indesejáveis.

Se as funções de hash escolhidas não forem boas, escolha novas funções de hash de forma a obter bons resultados. Descreva a análise efetuada, e os resultados no relatório. Indique no relatório se houve alguma alteração nas funções de hash, e os resultados para cada uma.

7) Esquecimento

A segunda funcionalidade chave implementada pela tabela de dispersão usada no sistema *Monolith* é a funcionalidade de esquecimento. Quando guardada na tabela de dispersão, iremos guardar um selo temporal (*timestamp*), que indica o tempo de inserção, ou a última vez que foi feita alguma coisa de interessante com aquela chave, como por exemplo, uma pesquisa, ou uma atualização.

A funcionalidade de esquecimento integra-se muito bem com a técnica de cuco, pois podemos utilizar uma técnica de esquecimento preguiçoso, que é eficiente, e favorece uma menor quantidade de trocas em novas inserções.

A ideia é simples, quando estamos a fazer uma inserção de uma nova chave, e a inserção de uma nova chave deu origem a uma troca de chaves (tanto faz que seja de T0 para T1 ou de T1 para T0), se a chave a mover tiver expirado (quer dizer que não houve interesse nesta chave), então podemos "esquecer esta chave" simplesmente não a movendo para a tabela seguinte. Nesta situação teremos de atualizar corretamente o número de chaves (adicionámos uma chave, mas a chave esquecida é para todos os efeitos apagada), e a grande vantagem é que não temos de prosseguir com o processo de trocas de cuco.

Iremos usar como tempo de expiração 24 horas. Ou seja, se uma chave não for utilizada durante 24 horas, irá ser eventualmente esquecida. As redes sociais não perdoam *memes* sem piada.

Atualize os métodos necessários para que a sua tabela de dispersão tenha a funcionalidade de esquecimento. Adicionalmente, para conseguirmos testar esta funcionalidade implemente o seguinte método:

`void advanceTime(int hours)`
Avança o tempo interno da tabela de dispersão o número de horas recebido.

A ideia disto não é mudar o tempo do sistema, mas guardar uma variável interna que guarde um desvio (*offset*) relativamente ao tempo correto. Por exemplo, se forem 15:30, e tiver um desvio de 6 horas, devo considerar que são 21:30. Este tempo com o desvio deve ser o tempo usado para guardar (ou atualizar) os selos temporais, e para verificar se uma chave deve ou não ser esquecida.

Importante: operações de redimensionamento da tabela de dispersão vão dar origem a uma reinserção de todas as chaves, mas neste caso, a reinserção não deve atualizar os selos temporais guardados junto de cada chave, uma vez que esta reinserção é apenas devido a um aspeto técnico de funcionamento da tabela, e não devido a uma atualização ou inserção de uma nova chave.

8) Impacto do esquecimento no número de trocas

Analise o impacto da funcionalidade de esquecimento no número de trocas (número médio de trocas, e variância).

Construa um teste em que cerca de 20% das chaves são acedidas com frequência, e que 80% das chaves são pouco acedidas (ou não acedidas), e veja o que acontece quando o tempo vai passando, e se vão inserindo novas chaves.

Compare uma versão com esquecimento, com uma versão sem esquecimento. Apresente os resultados e a sua análise sobre o impacto no número médio de trocas, a variância, e também o tempo médio de execução do método *put*.

Submeta as funções de teste implementadas, bem como alguns dos testes efetuados na sua função *main*, submetida no projeto. Descreva os testes, o resultado dos testes e a sua análise no relatório.

9) Análise de eficiência temporal

Nesta última alínea pretende-se analisar a eficiência temporal da tabela de dispersão implementada.

Nas aulas teóricas aprendemos que 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). Confirme este resultado empiricamente, fazendo testes de razão dobrada para os métodos *get* e *put.*

Determine também o tempo médio de execução dos método *get* e *put* da versão mais eficiente da tabela implementada, e compare com o tempo médio dos mesmos métodos de uma implementação alternativa de uma tabela de endereçamento aberto com exploração linear[2].

Adicionalmente, para além de testar a performance das tabelas com redimensionamento a partir de fator de carga de 50%, altere o redimensionamento para que só seja ativado a partir de um fator de carga de

[2] O Código desta tabela de dispersão é disponibilizada nos slides da cadeira.

75%, e faça a análise do tempo médio de execução dos métodos *get* e *put*, para a tabela de dispersão implementada, e para a tabela com exploração linear.

Submeta as funções de teste implementadas, bem como alguns dos testes efetuados na sua função *main*, submetida no projeto. Descreva os testes, o resultado dos testes e a sua análise no relatório.

Requisitos técnicos: Os métodos pedidos deverão ser implementados na classe *ForgettingCuckooHashTable* fornecida (ficheiro *ForgettingCuckooHashTable.java*). A classe deverá estar definida na package aed.tables[3]. Outras classes ou métodos auxiliares deverão ser definidas no mesmo ficheiro.

Os testes realizados devem ser submetidos juntamente com o vosso código, e devem ter um método *main* que execute alguns dos testes.

Submeta **apenas o ficheiro ForgettingCuckooHashTable.java** no Problema A.

Relatório

Parte da avaliação do trabalho efetuado neste problema será feita através do relatório. Portanto, para além da submissão no Mooshak, deverá ser feita a entrega de um relatório em formato pdf com o que foi pedido anteriormente.

Condições de realização

O projeto deve ser realizado em grupos de no máximo 2 alunos. Recomendamos que os grupos sejam formados por alunos do mesmo turno de laboratório. Projetos iguais, ou muito semelhantes, serão anulados e poderão dar origem a reprovação na disciplina. O mesmo se aplica ao relatório entregue. O corpo docente da disciplina será o único juiz do que se considera ou não copiar num projeto.

O código do projeto deverá ser entregue obrigatoriamente por via eletrónica, através do sistema Mooshak, **até às 23:59 do dia 15 de Novembro.**

O relatório deverá ser entregue também por via eletrónica na página da cadeira na tutoria um dia mais tarde, ou seja, **até às 23:59 do dia 16 de Novembro**.

As validações terão lugar na semana seguinte, de 18 a 22 de Dezembro. Os alunos terão de fazer a discussão juntamente com o docente **durante** o horário de laboratório correspondente ao turno em que estão inscritos. **A avaliação e correspondente nota do projeto só terá efeito após a discussão do projeto.** O corpo docente poderá atribuir **notas diferentes aos elementos do grupo, de acordo com a sua prestação individual na discussão.**

A avaliação da execução do código é feita automaticamente através do sistema Mooshak, usando vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória

[3] Poderá fazê-lo usando a instrução *"package aed.tables;"* (sem as aspas) no ínicio do ficheiro. O ficheiro deverá estar guardado na pasta *"src/aed/tables"*.

utilizada. Não é necessário o registo para quem já se registou no 1.º projeto, podendo usar o mesmo *username* e *password*.