UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2022-2023

Projeto 4 – Tabelas de Símbolos – versão (1.1)

Twualger

Imagem retirada de @MidnightMitch/Twitter

Introdução

Este projeto vai ser ligeiramente diferente dos anteriores, pois para além de saber implementar estruturas de dados de raiz, um Engenheiro Informático também deve saber usá-las para resolver problemas concretos de um sistema informático de forma eficiente. Portanto neste projeto iremos focar-nos na utilização de Tabelas de Símbolos (Tabelas de Dispersão e Árvores de Pesquisa) para implementação de funcionalidades de forma eficiente.

A ideia para este projeto surgiu com a recente aquisição do Twitter pelo Elon Musk, e a controvérsia gerada relativamente à eficiência do *Twitter*. Neste projeto iremos debruçar-nos numa das funcionalidades do *Twitter*, a construção de uma linha temporal – *timeline* (a partir deste momento usaremos o termo em inglês) – onde os *tweets* das contas que seguimos são apresentados por ordem cronológica descendente.

O sistema a desenvolver – Twualger – irá ler os tweets a partir de ficheiros[1]. No entanto, como existem contas com um número elevado de seguidores, estas contas irão muito frequentemente aparecer em *timelines*. Assim sendo, uma boa forma de tornar o sistema mais rápido é usar uma cache, guardando em memória *RAM* os tweets das celebridades com mais seguidores.

Para implementar as funções do projeto, recomendamos que o façam pela ordem aqui apresentada no enunciado, pois funções especificadas mais tarde podem depender de funções definidas anteriormente. Foi fornecido juntamente com o enunciado algum código fonte que deverá usar para o seu projeto, incluindo o esqueleto das classes principais a implementar para o problema A e B. Recomenda-se a leitura com atenção do código fonte fornecido.

Os testes no *Mooshak* estão desenhados de modo a ir obtendo pontos à medida que se implementam corretamente cada um dos métodos pedidos. Não é necessário ter um problema totalmente implementado para obter a cotação correspondente no projeto.

Problema A: TwualgerA

Neste problema começaremos por implementar uma versão menos eficiente do sistema, mas que já é capaz de ler tweets a partir de ficheiros, guardá-los numa cache, e construir *timelines*. Não precisa de se preocupar em tornar esta implementação muito eficiente. Isso será feito no problema seguinte.

Começe por preparar o projeto (em IntelliJ ou Eclipse), descarregando o código fonte fornecido, bem como os ficheiros com *tweets*[2] que iremos usar. Os ficheiros com *tweets* deverão ser descarregados e colocados dentro de uma pasta data dentro da vossa diretoria principal do projeto. Este problema deverá ser implementado usando a classe TwualgerA fornecida.

1) readTweetsFromFile

Este método é responsável por ler os tweets de um determinado utilizador a partir de um ficheiro. Recebe como argumentos um *path*, que indica a directoria onde os ficheiros .csv com os *tweets* estão localizados e um *username*.

Deverão construir o *path* final usando o *path* recebido e o nome do ficheiro. Cada utilizador tem um ficheiro com os seus tweets, e o nome do ficheiro correspondente é dado pelo *username*. Por exemplo, para o Elon Musk, cujo username é elonmusk, o nome do ficheiro correspondente é elonmusk.csv.

Para ler o ficheiro é recomendado um *BufferedReader* em vez de um *Scanner* devido aos problemas de ineficiência do Scanner.

Um ficheiro csv com *tweets* tem a seguinte organização: A 1.ª linha tem um cabeçalho com 3 campos (*twitter*_id,*date*,*tweet*) separados por vírgula. Esta linha pode ser ignorada. Cada uma das restantes linhas corresponde a um *tweet*, organizado da mesma forma que o cabeçalho. Podemos ver um exemplo na figura seguinte.

[1] Poderíamos ler de uma base de dados, mas no contexto deste projeto, não é relevante.
[2] Os tweets foram obtidos do site kaggle em Novembro de 2022:
https://www.kaggle.com/datasets/ahmedshahriarsakib/top-1000-twitter-celebrity-tweets-embeddings

```
1  twitter_id,date,tweet
2  1546602051022495746,2022-07-11 21:07:18+00:00,b'@BillyM2k Mimic + Dark Moon + Stars of Ruin'
3  1546598042844991489,2022-07-11 20:51:22+00:00,b'https://t.co/mvVUR4doqB'
4  1546561966331789318,2022-07-11 18:28:01+00:00,b'@greg16676935420 @MrBeast \xf0\x9f\x94\xa5\x
5  1546552413083111424,2022-07-11 17:50:03+00:00,b'@WholeMarsBlog Absolutely'
```

Figura 1 Extrato de ficheiro csv com os tweets do utilizador elonmusk

Este método deverá ler todos os *tweets*, e devolver uma *UserCache* de *tweets* (aqui deverão usar as classes Tweet e UserCacheA fornecida,).

Como data iremos usar o tipo *OffsetDateTime* do Java, que representa uma data e tempo com zona temporal. Para extrair a data a partir de uma *String*, devemos usar o método estático *OffsetDateTime*.parse, e passar a *String* juntamente com um *DateTimeFormatter* que pode ser construído através da expressão `DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssZZZZZ")`.

2) Construtor TwualgerA

Uma vez implementado o método anterior podemos implementar o construtor. A ideia é que para esta classe iremos usar uma cache para guardar os *tweets* de cada utilizador. Nesta versão mais simples iremos usar um *ArrayList* para guardar as caches de cada utilizador.

Existe uma lista bem conhecida de quais as contas com maior número de seguidores. Esta lista pode ser obtida usando o método *readTopCelebs* da classe abstrata *Twualger*. No construtor, para além de inicializar os campos da classe, deverá carregar imediatamente para a cache os *tweets* de todos as contas que pertencem ao top, pois a probabilidade de um destes ser usado em qualquer pesquisa é muito elevada.

3) getUserCache

Este método recebe um *username*, e vai procurar na cache em memória pelos *tweets* desse utilizador. Caso os tweets não existam em memória, irá ler os tweets a partir do ficheiro usando o método implementado na alínea 1, e irá colocá-los na sua cache. Finalmente é devolvida a *UserCache* com todos os *tweets* do utilizador.

Este método irá contabilizar também o número de acertos e falhas (hits e misses) da cache. Um hit ocorre quando ao procurar pelos *tweets* de um utilizador, a cache já se encontra em memória, e um miss ocorre quando a cache ainda não existe em memória e tem de ser lida a partir do ficheiro. Adicionalmente, sempre que uma *UserCache* é devolvida (independentemente de ter sido lida do disco, ou retirada de memória) é incrementado o número de vezes que essa cache foi usada (ou procurada).

4) totalSearches e cacheHitRatio

Aqui pretende-se implementar dois métodos simples úteis para perceber o funcionamento da cache. O *totalSearches* retorna o número total de pesquisas que foram feitas no sistema, contabilizando o total para todos os utilizadores. Por exemplo, para construirmos uma *timeline* com 10 contas que estão a ser seguidas, o número de pesquisas necessárias são 10 (independentemente das contas já estarem em cache ou em disco).

O segundo método, *cacheHitRatio* calcula e devolve um *float* entre 0 e 1 que representa o rácio entre o total de hits (pesquisas por elementos já na cache) e o número total de pesquisas efetuada.

5) buildTimeLine

Este é o método responsável por construir uma *timeline* que irá ser usada mais tarde para renderizar a homepage de um determinado utilizador. Recebe como argumento uma Lista de nomes de contas que são seguidas pelo utilizador, uma data de início (*from*) e uma data de fim (*to*). Este método irá buscar os tweets de cada conta que está a ser seguida, que estejam entre a data de início e a data de fim. No fim é devolvida uma lista com todos os *tweets* agrupados, ordenados de forma cronológica descendente, ou seja, do tweet mais recente para o *tweet* mais antigo.

Para ordenar a lista retornada pode por exemplo usar o método *sort* da classe *ArrayList*, pois este método usa um algoritmo de ordenação muito semelhante ao método de ordenação implementado no projeto 3, e que irá funcionar de forma bastante eficiente para *arrays* que já estão parcialmente ordenados[3]

6) downsizeCache

Este método tem como objetivo diminuir o tamanho da cache quando ela tem um tamanho demasiado grande. Quando for possível, pretende-se eliminar 50% das caches menos usadas. No entanto, existem utilizadores que queremos sempre manter em cache, são os utilizadores top que foram carregados no construtor. Por esta razão, pode acontecer que não seja possível eliminar 50% das caches. Portanto vamos eliminando as caches menos usadas (evitando apagar as pertencentes a contas top), e quando tivermos eliminado 50% das caches (ou quando não houver mais caches para analisar) paramos.

Adicionalmente este método também irá fazer *reset* a todos os contadores relacionados com a contabilização do funcionamento da cache, como por exemplo o *cacheHit*, o *cacheMiss*, e os contadores para a cache de cada utilizador (que não seja apagado).

Não se preocupe demasiado com a eficiência deste método, pois não irá ser invocado com tanta frequência como os outros.

Requisitos técnicos: Os métodos pedidos deverão ser implementados na classe *TwualgerA* fornecida (ficheiro Twalger.java). Outras classes ou métodos auxiliares deverão ser definidas no mesmo ficheiro.

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

Problema B: Twualger B

Imagine que O CTO[4] da empresa fícticia (Twualger) que nos contratou é tão exigente ou ainda pior que o Elon Musk – vamos chamar-lhe Melon Tusk. Durante uma das *daily meetings*[5], o Melon Tusk pediu para ver o nosso código (Problema A), e chamou-nos vários nomes desagradáveis ao ver que estavamos a guardar as caches num *ArrayList*. E avisou-nos de que se o código não estivesse melhor e mais eficiente até ao final do *Sprint*[6], seriamos despedidos ☹. O *Sprint* termina no dia 2 de Novembro.

---

[3] Se pensarem bem, vão ver que este é o nosso caso, pois cada cache de cada utilizador já está na realidade ordenada pela ordem pretendida. O resultado final é que não está.

[4] Chief Technical Officer

[5] Uma *daily* é nome dado a uma reunião rápida entre uma equipa de desenvolvimento numa metodologia de desenvolvimento ágil chamada *Scrum*, e que é muito usada atualmente em desenvolvimento de *software*.

[6] Na metodologia *Scrum*, o desenvolvimento é divido em blocos temporais curtos (normalmente duas semanas) onde um conjunto de funcionalidades é escolhido para implementação e implementado. Cada um destes blocos é designado de *Sprint*, e termina numa reunião de *Sprint Review*.

Felizmente vamos poder usar algumas das coisas que aprendemos em AED para não ser despedidos.

A primeira coisa que podemos fazer é usar uma Tabela de Dispersão (*HashTable*) para guardarmos as caches de cada utilizador (indexadas pelo username ou handle que é único). Dessa forma, cada vez que precisarmos de verificar (ou encontrar) a cache para um determinado utilizador, podemos fazê-lo em tempo constante (em vez de ter de percorrer a lista toda).

A segunda coisa que podemos fazer é tornar mais eficiente a seleção de tweets a partir das datas pretendidas (em vez de andarmos a comparar um a um). Uma forma de o conseguir é usar uma Tabela de Símbolos Ordenada. Aqui podemos usar a ordenação natural das datas (a classe *OffsetDateTime* do Java é *Comparable*), usando a data como chave para guardar e organizar os tweets de cada utilizador. Assim sendo, a nossa Tabela de Símbolos Ordenada vai conter todos os Tweets ordernados do menor (mais antigo) para o maior (mais novo).

Uma vez que usemos uma Tabela de Símbolos Ordenada pela data, passamos a ter uma série de funcionalidades eficientes ($O(\log_2 n)$), como por exemplo obter o máximo (*tweet* mais recente), o mínimo (*tweet* mais antigo), todos os valores a seguir (ou antes) de uma data, ou até mesmo todos os valores num intervalo de datas.

Para usar uma Tabela de Símbolos Ordenada em Java poderiamos usar a classe *TreeMap* do Java, que implementa uma Tabela de Símbolos Ordenada usando árvores *red-black* (tal como as demos nas aulas teóricas), mas em vez disso (para tentar impressionar o Melon Tusk) vamos fazer algo com mais estilo. Vamos usar uma outra implementação: *Treap*[7]*. Um Treap* é uma estrutura de dados que combina características de uma binary search **tre**e com um h**eap**, e usando números aleatórios como prioridades que vão ser usadas para reorganizar o Treap, de modo a que a árvore final obtida seja semelhante à arvore que se obteria colocando os elementos por ordem aleatória. O Treap foi um dos projetos do ano passado, implementado pelos vossos colegas. Não o vão ter de implementar (o código é dado), mas vão ter de o saber usar.

Implemente novamente os métodos implementados no problema A, mas desta vez tendo em conta as optimizações mencionadas.

Requisitos técnicos: Os métodos pedidos deverão ser implementados na classe *TwualgerB.* Deverá usar também a classe *UserCacheB, e a classe Treap para organização da cache.* Caso necessite de outras classes ou métodos auxiliares, estas deverão ser definidas no mesmo ficheiro.

Submeta **apenas o ficheiro *TwualgerB*.java** no Problema B.

Problema C: Twualger C

As alterações efetuadas no Problema B já deixaram o Melon Tusk um pouco mais satisfeito, mas ele acha que o sistema ainda está demasiado lento. Após analisar os pedidos de pesquisa mais comuns, chegámos à conclusão que 80% a 90% das vezes as pesquisas para construção de uma *timeline* correspondem às últimas 72 horas[8]

[7] Para confirmar a escolha, comparei a eficiência do Treap contra a TreeMap do Java, e o Treap é ligeiramente mais rápido.

[8] Como só temos dados de tweets até 11/07/2022, iremos usar as 72 horas anteriores ao fim deste dia.

Sendo esse o caso, não faz muito sentido estar a carregar todos os *tweets* existentes para a cache guardada em memória *RAM*. Vamos fazer uma nova otimização com baste nesta ideia. Iremos precisar e efetuar as seguintes alterações.

1) readUserTweetsFromFile

O método de leitura de *tweets* irá receber agora mais um argumento, que é a data mais antiga para a qual queremos ler os *tweets*. À medida que formos lendo os vários *tweets*, vamos verificar se a data do *tweet* é anterior à data mais antiga. Se o for, quer dizer que já não queremos guardar esse *tweet* na cache, e podemos parar a leitura. No entanto, antes de retornarmos é importante guardarmos uma *flag* na cache a dizer que existem mais *tweets* por ler no ficheiro.

2)updateUserCacheFromFile

Iremos também precisar de um método adicional de leitura, utilizado para atualizar uma cache para ir ler *tweets* mais antigos até uma data mais antiga. Quando estamos a atualizar uma cache, não nos interessa estar a colocar no *treap* os tweets que já lá estão, e por isso só adicionamos tweets para datas que estão entre a anterior data mais antiga do *Treap*, e a nova data mais antiga recebida como argumento. Por exemplo, se o *treap* tiver como data mais antiga 72 horas anteriores (-72H), e precisarmos de ler até uma semana atrás (-7D), quando estivermos a ler do ficheiro podemos ignorar todas as datas mais recentes, e só colocar no *treap* as datas entre -7D e -72H. Quando chegamos a uma data que é anterior à nova data mais antiga, esse *tweet* já não deve ser registado e podemos parar a leitura.

Adicionalmente, também temos de voltar a determinar qual o valor correto para a *flag* que indica se existem mais *tweets* por ler.

3)Construtor TwualgerC e getUserCache

Agora quando tentamos obter os *tweets* de um conta pela primeira vez, e essa conta não está em cache, devemos ler apenas os tweets até às 72 horas anteriores[9] (da data mais recente para a qual temos dados). Isto aplica-se também à leitura das contas top no construtor.

4) buildTimeLine

Quando construirmos uma *timeline*, temos de verificar se pretendemos incluir *tweets* de uma data mais antiga que o tweet mais antigo guardado no *treap*. Se isto acontecer, e se a cache estiver marcada como tendo mais *tweets* por ler, então vamos ter de atualizar a cache para que vá buscar todos os tweets até a nova data pretendida. Mas se a cache estiver marcada como não tendo mais tweets por ler, não é preciso fazer nada, pois mesmo que tentássemos não conseguiríamos ler mais nada.

Uma vez atualizada a cache, podemos prosseguir como fazíamos no problema B.

5) downsizeCache

O método de *downsize*, para além de remover as contas menos pesquisadas da cache, irá também remover os *tweets* mais velhos que as 72 horas anteriores para as contas que ficarem em cache.

6) Comparação entre implementações efetuadas

Uma vez implementadas as otimizações, tem de conseguir convencer o Melon Tusk de que o seu código está bastante mais eficiente. Compare o tempo médio de execução de muitas pesquisas (i.e. construção

[9] Pode caso entenda usar outro valor que faça sentido.

de *timelines*) para as 3 variantes implementadas, usando as seguintes condições: 80% das pesquisas são feitas a contas top (restantes 20% às outras contas), 80% das pesquisas começam na data mais recente (restantes 20% podem começar até 10 meses atrás) e 80% das pesquisas têm uma amplitude de 72 horas (restantes 20% podem ter uma amplitude até 90 dias).

Escreva um breve sumário dos testes e resultados, e uma breve análise dos resultados como comentários no ficheiro Java a submeter. Indique qual o ganho de eficiência temporal obtido quando comparando a solução final com a inicial.

O método *main* deverá invocar os testes utilizados. Embora este método não seja validado de forma automática pelo *Mooshak* será tido em consideração na validação do projeto, e contará para a nota final do mesmo.

Condições de realização

O projeto deve ser realizado individualmente. Projetos iguais, ou muito semelhantes, originarão a reprovação na disciplina. 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 02de Dezembro.** Os alunos terão de validar o código juntamente com o docente **durante** o horário de laboratório correspondente ao turno em que estão inscritos. **A avaliação e correspondente nota do projeco só terá efeito após a validação do código pelo docente.**