UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2025-2026

Projeto 3 – Tabelas de Símbolos

Introdução

No 3.º Projeto da cadeira de AED pretende-se explorar a implementação de tabelas de símbolos, com particular foco nas árvores de pesquisa binária e em *tabelas de dispersão*. Este projeto tem 2 problemas e um relatório. O problema A incide sobre árvores de pesquisa binária e o problema B sobre tabelas de dispersão. Como ainda não foram lecionadas tabelas de dispersão, o problema B só será publicado mais tarde.

Problema A: UAlgTree (8 valores)

No problema A iremos implementar uma variante de uma árvore binária de pesquisa, auto-balanceadora, e que possui um mecanismo que favorece a pesquisa de elementos mais frequentes. A esta árvore designaremos de *UAlgTree*.

Auto-balanceamento

O mecanismo de auto-balanceamento da árvore é baseado numa função w(n), que dado um nó n, calcula o peso do nó. **A especificação concreta da função w(n) será dada nas aulas de laboratório de AED**.

A função *w(n)* representa de certa forma o peso da sub-árvore iniciada a partir do nó *n*. Para um nó pai *n*, o nó diz-se perfeitamente balanceada se o peso da sub-árvore esquerda de n é igual ao peso da sub-árvore direita de n. Uma árvore *A* é perfeitamente balanceada se todos os seus nós forem perfeitamente balanceados.

$$\forall n \in A, w\left(filho_{esq}(n)\right) = w\left(filho_{dir}(n)\right)$$

Mas no nosso caso não estamos interessados num balanceamento perfeito, mas sim num balanceamento razoável. Dizemos que uma árvore A é razoavelmente balanceada se e só se:

$$\forall n \in A, w\left(filho_{esq}(n)\right) \geq 0.4\, w\left(filho_{dir}(n)\right) \text{ e } w\left(filho_{dir}(n)\right) \geq 0.4\, w\left(filho_{esq}(n)\right)$$

Ou seja, para qualquer nó da árvore, nenhum dos filhos tem menos de 40% do peso do outro filho, ou alternativamente, nenhum dos filhos tem um peso mais do que 2,5 vezes superior ao outro.

Inserção

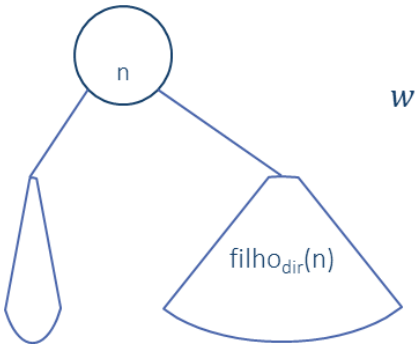
Quando é feita a inserção de uma nova chave (e correspondente valor) numa *UAlgTree* temos de garantir que a árvore continua razoavelmente balanceada depois da inserção. Isto é feito de forma semelhante a uma árvore red-black, mas desta vez usaremos os pesos para decidir quando fazer uma rotação.

Começamos por percorrer a árvore, e com a chave recebida, determinamos o sítio de inserção. Inserimos o novo nó, atualizando os pesos correspondentes. Dado o nó pai atual, caso os dois

filhos respeitem a propriedade acima, este nó já está razoavelmente balanceado, e não precisamos de fazer nenhuma rotação. Avançamos para cima na árvore (ou retornamos caso a nossa implementação seja recursiva).

Caso um dos filhos seja significativamente mais gordo que o outro, o nó não está razoavelmente balanceado, e temos de fazer algumas rotações para que algum peso do filho gordo passe para o outro lado.

Consideremos o caso em que o filho da direita é bastante mais gordo que o filho da esquerda, como ilustrado na figura seguinte:



$$w\left(filho_{esq}(n)\right) < 0.4w\left(filho_{dir}(n)\right)$$

Figura 1. Exemplo de filho demasiado gordo à direita.

Nesta situação, teremos de passar peso da direita para a esquerda. Para determinar o tipo exato de rotação a fazer, temos de analisar os filhos do filho direito de n, ou seja os seus netos direitos.

Caso o neto esquerdo do filho direito seja mais que 1.5 vezes mais gordo que o seu irmão (estamos na situação ilustrada na Figura 2), não podemos fazer uma rotação para a esquerda entre *n* e *x*, pois isto iria transladar o peso deste neto para a esquerda de n. O nó deixaria de estar desbalanceado à direita, mas passaria a estar desbalanceado à esquerda.

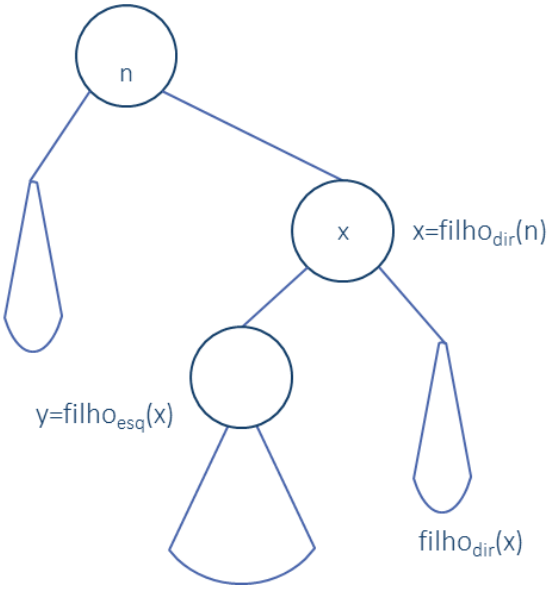


Figura 2. Situação em que o filho gordo da direita tem um filho gordo à esquerda, temos de fazer duas rotações neste caso.

Para resolver este problema, temos de fazer duas rotações (ver Figura 3). Em primeiro lugar fazemos uma rotação para a direita, fazendo com que *y* troque de lugar com *x*. Isto irá mover parte do peso extra do lado esquerdo de x para o seu lado direito.

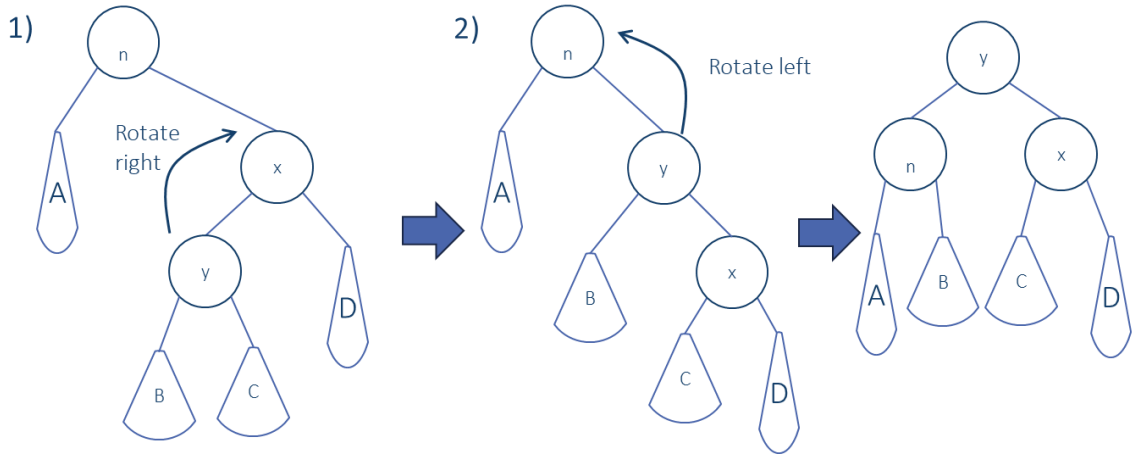


Figura 3 – Rotação à direita seguida de rotação à esquerda para resolver a situação ilustrada na figura 2.

De seguida, rodamos o nó y para a esquerda, trocando-o de lugar com n. Esta rotação irá equilibrar os pesos do lado esquerdo e direito da sub-árvore resultante.

Caso o neto esquerdo do filho direito de n não seja mais que 1,5 vezes mais gordo que o seu irmão, isto implica que ou têm ambos um peso parecido, ou que o neto mais gordo é o da direita. Em qualquer um destes casos, basta uma única rotação para tornar a sub-árvore razoavelmente balanceada. A rotação a efetuar é uma rotação para a esquerda entre o filho direito de *n*, e o nó *n*.

As rotações a efetuar para o caso em que o filho mais gordo de *n* seja o filho da esquerda são simétricas relativamente às situações aqui ilustradas. Ou fazemos duas rotações, rotate left seguido de rotate right, quando o filho esquerdo gordo de n tem um filho gordo à sua direita (com mais que 1,5 vezes o peso do seu irmão), ou fazemos uma única rotação rotate right caso contrário.

Remoção

A remoção de uma chave da árvore implica a remoção de um nó, que no pior caso poderá ter dois filhos, o que torna a remoção um pouco mais complicada. A remoção de um nó na nossa árvore começa por ser igual à remoção numa árvore binária de pesquisa tradicional.

Numa fase inicial, a remoção corresponde a encontrar a chave que se pretende remover. Se não encontrarmos a chave, não existe nada a remover. Quando encontramos a chave, o nó onde a chave se encontra tem de ser removido da árvore. Aqui temos 3 casos possíveis.

1) Se o nó a remover é um nó sem filhos, a sua remoção é trivial. Basta colocar a *null* o ponteiro que o seu pai tem para ele, e atualizar os campos como *size* e *weight*.
2) Se o nó a remover tem apenas um único filho (ou esquerdo, ou direito) podemos substituir diretamente o nó pelo seu único filho (atualizando o ponteiro do pai).
3) Se o nó *n* a remover tem 2 filhos não o podemos remover diretamente. Temos de encontrar a partir do filho direito de *n* o nó com a menor chave possível (chamemos-lhe n_2). Uma vez encontrado n_2, trocamos a chave e valor de *n* com n_2. De seguida removemos o nó n_2 da árvore, aplicando o caso 1) ou 2). A figura 4 ilustra a remoção de um nó nesta situação.

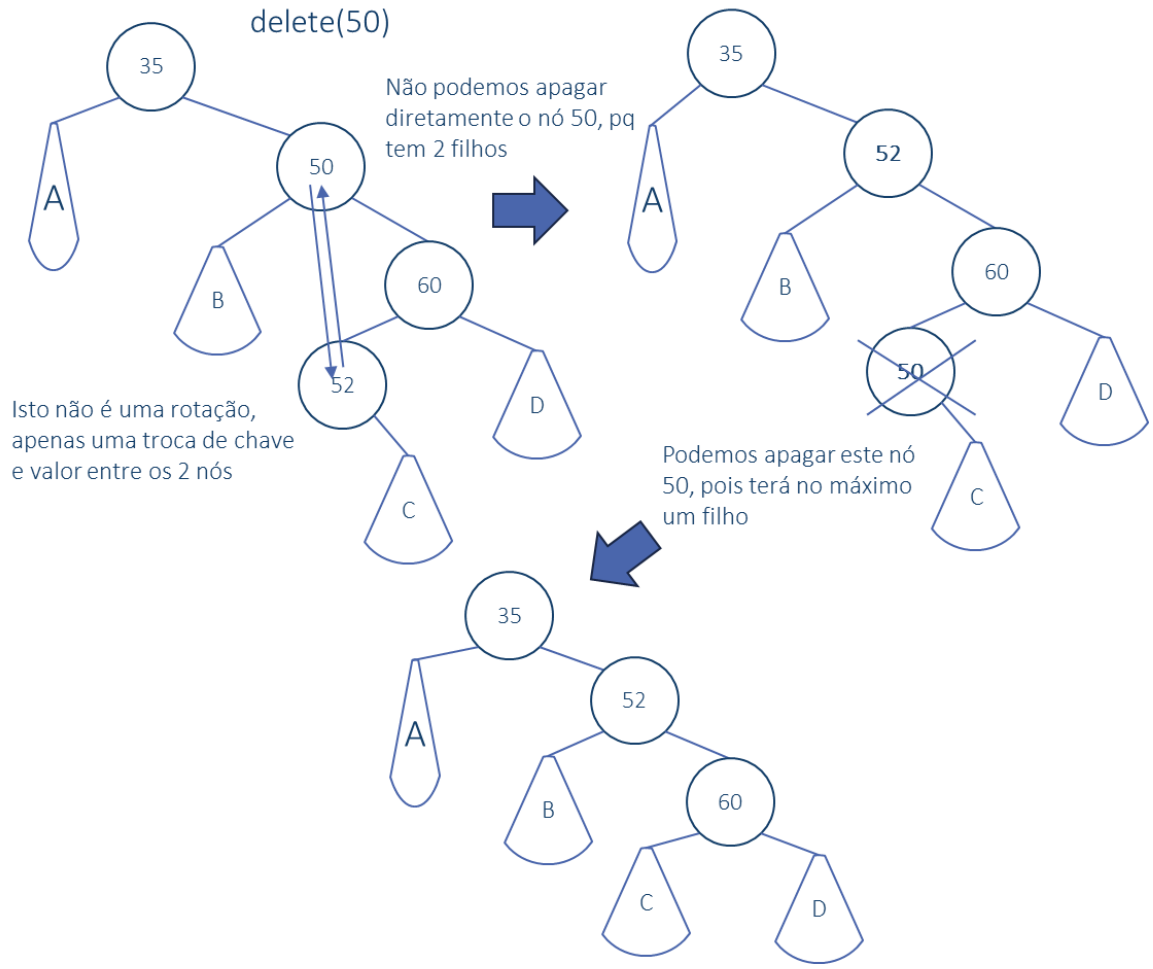


Figura 4 – Exemplo de remoção de um nó com dois filhos.

Uma vez removido o nó, é necessário atualizar o tamanho e peso de cada um dos nós no caminho até ao nó removido, o que poderá implicar um debalanceamento da árvore. Tal como é feito na inserção, devemos voltar a testar o balanceamento razoável para todos os nós no caminho, e aplicar as rotações indicadas (quando necessário), da mesma forma que foram feitas para o caso da inserção.

Pesquisa

A pesquisa na nossa árvore tem uma característica importante, que é tentar tornar a pesquisa mais eficiente para as chaves que são pesquisadas com maior frequência. Para conseguir este objetivo, iremos ter de implementar a seguinte propriedade:

Quando é feita uma pesquisa por uma chave k que exista na árvore, o nó n que contém a chave k deverá obrigatoriamente subir um nível na árvore, a não ser que o nó n já seja a raiz da árvore.

Isto fará com que as chaves pesquisadas mais frequentemente acabem por ficar no topo, ou perto do topo da árvore, portanto, a pesquisa por essas chaves torna-se mais eficiente.

Uma solução simples para implementar isto seria fazer uma rotação (rotate left para o caso em que o nó pesquisado está à direita do seu pai, e rotate right caso contrário) mas isto poderá

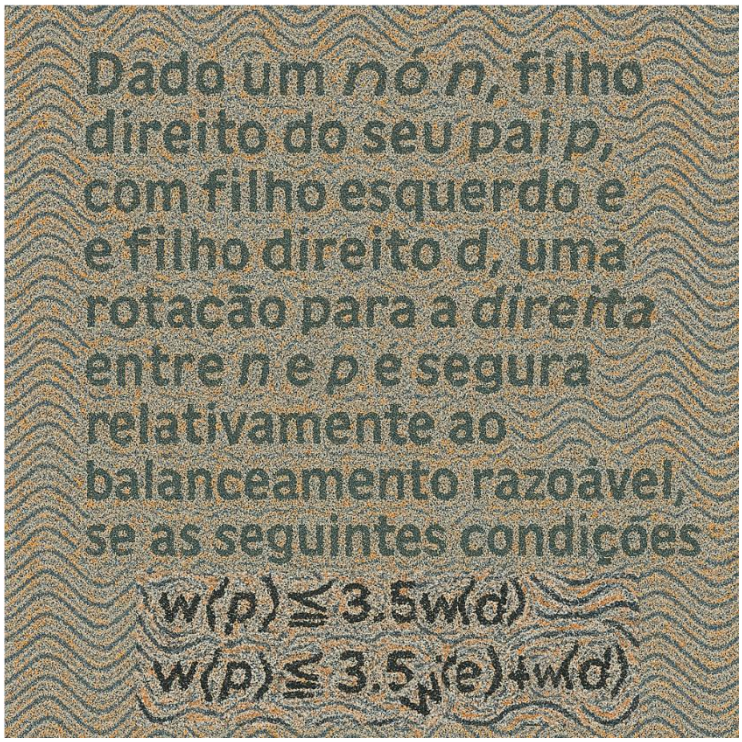
provocar um desbalanceamento, e queremos tentar evitar que isto aconteça, por isso não aplicaremos esta solução tão simples[1].

A nossa implementação parte da observação de que para subir o nó *n* com a chave *k* um nível podemos aplicar uma rotação a qualquer par de nós no caminho entre o nó *n* e a raiz.

Portanto, começamos por fazer uma pesquisa como faríamos numa árvore binária de pesquisa tradicional.

1) Se não encontrarmos a chave k, não há nada a fazer, tal como na pesquisa numa árvore normal, retornamos null.
2) Quando encontrarmos a chave k, seja *n* o nó que a contém. Várias situações podem ocorrer:
    a. Se o nó *n* é um nó raiz, não é preciso fazer nada, pois a chave já está no topo da árvore.
    b. Se o pai de *n* é um nó raiz, e ainda não foi feita nenhuma rotação, somos obrigados a fazer esta rotação (não vale a pena testar o balanceamento). Fazemos a rotação (rotate left caso *n* seja um filho direito do seu pai, ou rotate right caso contrário).
    c. Caso contrário, aplicaremos uma rotação entre o nó n e o seu pai apenas se a rotação for determinada como segura para o balanceamento razoável. Se a rotação for efetuada, já não é necessário fazer mais nenhuma rotação. Se a rotação não for segura, desistimos de tentar rodar este nó, e avançamos para o seu pai, voltando a repetir os testes b e c, até que uma rotação seja efetuada.

A definição do que é considerado uma rotação segura é dada nas figuras seguintes:



[1] Se estiverem aflitos com a implementação da técnica mais completa, podem implementar esta variante mais simples, que vos dará 6 valores em vez de 8 valores.

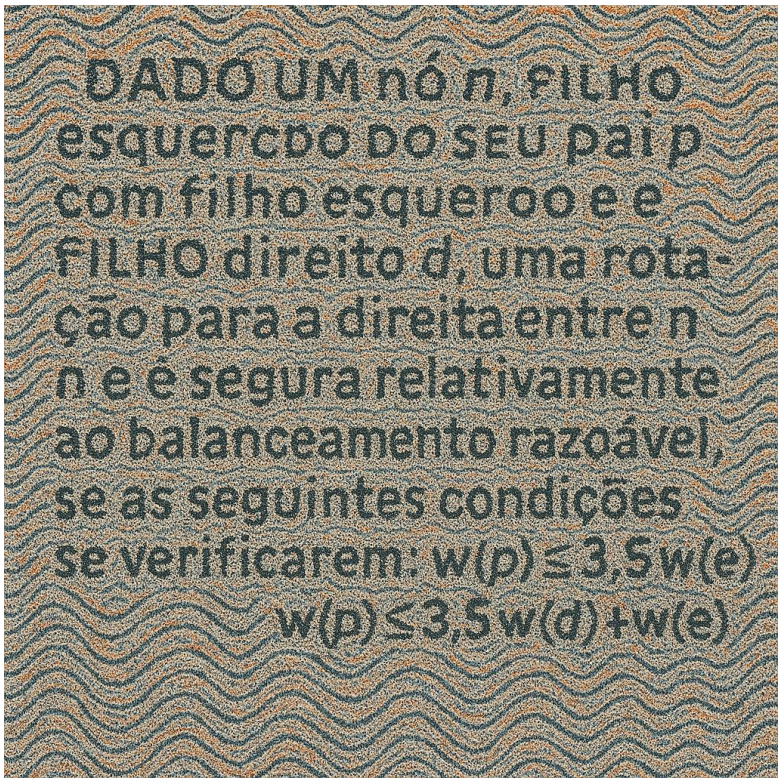


Figura 5 – definição de uma rotação segura para o balanceamento razoável

Não nos podemos esquecer que para além da rotação, no final, o método de pesquisa deverá retornar o valor associado à chave encontrada.

Implemente a estrutura de dados *UAlgTree* de acordo com a especificação. Poderá implementar outros métodos privados auxiliares caso assim o entenda, mas deverá implementar obrigatoriamente os seguintes métodos:

UAlgTree<Key extends Comparable<Key>,Value> – Implementa uma tabela de símbolos ordenada de acordo com a especificação indicada.		
	`UAlgTree()`	Cria uma árvore vazia.
`IUAlgTreeNode`	`getRoot()`	Retorna uma interface para o nó da árvore que corresponde à raiz.
`int`	`size()`	Retorna o número de pares chave-valor guardados na *árvore*
`boolean`	`containsKey(Key k)`	Verifica se uma chave existe na *árvore*. Retorna *true* caso a chave exista e *false* caso contrário.
`Value`	`get(Key k)`	Retorna o valor guardado na árvore para a chave recebida como argumento. Caso a chave não exista na árvore é retornado o valor *null*.
`void`	`put(Key k, Value v)`	Insere o valor v na árvore associando-o à chave k. Caso a chave já exista na árvore é feita uma atualização ao seu valor. Inserir o valor *null* numa árvore deve ser equivalente a fazer delete(k).
`void`	`delete(Key k)`	Remove a chave (e o seu valor) da árvore. Tentar remover uma chave que não existe não produz qualquer efeito.

`Iterable<Key>`	`keys()`	Devolve um objecto iterável que pode ser usado para percorrer (através de um foreach) todas as chaves da árvore. As chaves devem ser percorridas obrigatoriamente da menor para a maior chave.
`Iterable<Value>`	`values()`	Devolve um objecto iterável que pode ser usado para percorrer (através de um foreach) todos os valores da *árvore*. Os valores devem ser iterados pela mesma ordem que as chaves.
`UAlgTree`	`shallowCopy()`	Retorna uma cópia superficial da *Árvore*. Ou seja, retorna uma nova árvore com uma estrutura semânticamente equivalente à árvore recebida, mas sem fazer uma cópia dos objectos (chaves e valores) que estão na árvore original.
`void`	`main(string[] args)`	Método main, que deverá ser usado para testar os métodos acima.

Método main

Este método deverá implementar os testes usados para testar e comparar os métodos acima. Embora este método não seja validado de forma automática pelo Mooshak será tido em consideração na validação do projecto, e contará para a nota final do mesmo.

Implemente a classe *UAlgTree* no ficheiro *UAlgTree.java*. Submeta **apenas o ficheiro *UAlgTree.java*** no Problema A.

Problema B: *UAlshTable (9 valores)*

No problema B iremos implementar uma tabela de dispersão muito parti**c**ular, que designarem**o**s de *UAlshTable*. As tabelas de dispersão estudadas nas teóricas já são muito eficie**n**tes, no entanto não no**s** dão garant**i**as verdadeiras do que acont**e**ce no pior caso, e são vulner**á**veis a comportamento malicioso[2]. Adicionalmente, as implementações estudadas assumem que o custo de comparar 2 chaves não é demasiado elevado. No entanto, existem casos práticos em a comparação de 2 chaves pode ser pesada. Por exemplo, a comparação das strings “aaaaaaaaaaaa” e ““aaaaaaaaaaab” envolvem a comparação de 13 caracteres.

A **t**abela de dispersão que iremos implementar, tem como objetivo, com uma probabilidade elevada, de fazer apenas uma única comparação ao inserir ou pesquisar por uma c**h**ave. Poderão **e**xistir casos onde seja feita mais do que uma comparação, mas estes terão garantidamente uma menor probabilidade. Adicionalmente, esta tabela será robusta face a ataques de colisão de **h**ash.

Múltiplas funções de dispersão

A pr**i**meira característica importante da *UAlshTable* é que utiliza várias funções de hash, calculadas a partir de **d**uas funções hashco**d**e independentes.

[2] Por exemplo, um “hash attack” no contexto de negação de serviço (DoS) refere-se normalmente a um ataque por colisões de hash, no qual um atacante explora uma função *hash* fraca para fazer com que uma tabela hash tenha muitas colisões. Isto força a estrutura de dados ao seu pior desempenho, $O(n)$. Isto é relativamente fácil de fazer, se por exemplo usarmos Strings como chaves.

Dada uma chave x, $h_{code1}(x)$ $h_{code2}(x)$ são duas funções de hash independentes que retornam um inteiro. A partir destas duas funções de hashcode podemos obter qualquer número i=1…L de funções de hash, combinando-as da seguinte forma:

$$h_i(x) = (h_{code1}(x) + ih_{code2}(x))\%m_i$$

Sendo m_i o tamanho da tabela *i* onde a função h_i será aplicada.

Múltiplas subtabelas de dispersão

A nossa tabela de dispersão é na realidade composta por L subtabelas de dispersão, em que **a** 1ª subtabela T_1 tem como tamanho m_1, sendo m_1 da ordem de **g**randeza *n*, onde *n* é o número de pares chave-valor que queremos guardar na tabela, **e** as subtabelas seguintes têm aproximadamente metade do tamanho da subtabela anterior. O valor de m_1 não é igual a n apenas porque queremos usar números primos para os tamanhos das subtabelas, e por isso escolhemos para m_1 o número primo superior ou igual a *n*, mais perto de *n*. O mesmo se aplica para as restantes subtabelas. Por exemplo, para n=1024, escolhemos para m_1 1031 e para m_2 521**!**

A cada uma das posições das subtabelas, designaremos de baldes. Um balde pode estar vazio (ainda não foi colocado nada lá dentro), ou ter informação acerca de um par chave/valor. A ideia principal, é a de que, quando queremos inserir uma nova chave X, iremos aplicar as L funções de hash $h_i(x)$, para determinar a posição possível em cada uma das L subtabelas de dispersão. Das L posições possíveis, escolhemos para inserção a subtabela com menor *i* que encontre o balde respetivo a $h_i(x)$ vazio.

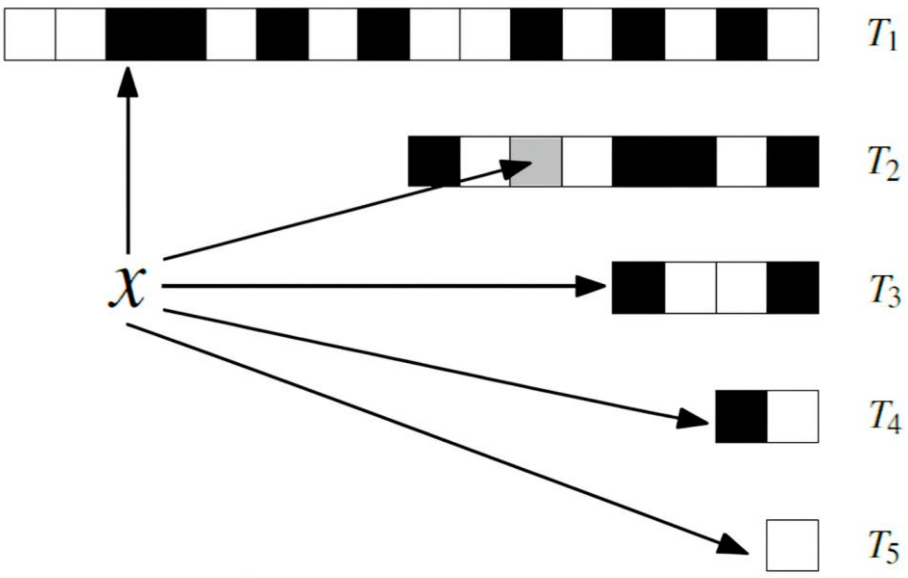


Figura 6 – Ilustração de uma inserção numa tabela com 5 subtabelas. Os baldes pretos encontram-se preenchidos, e os baldes a branco estão vazios. O balde cinzento foi o balde escolhido para inserção da nova chave.

Pesquisa eficiente

Uma implementação menos eficiente da função de pesquisa corresponde, para uma tabela com L subtabelas, simplesmente a calcular L valores de hash $h_i(x)$, …, $h_L(x)$, ir buscar os baldes correspondentes, e no pior caso, comparar a chave procurada com cada uma das L chaves guardadas. No entanto, como queremos tentar fazer 1 comparação apenas, não faremos isso.

Seja B_i o balde correspondente à posição $h_i(x)$ da subtabela i para a chave x.

$$B_i = T_i[h_i(x)]$$

O método de pesquisa começa por ir buscar todos os baldes Bi. Diz-se que estes baldes estão ligados entre si pela chave x, ou que partilham a chave x, se a chave for inserida em qualquer um dos baldes.

Entre outras coisas, cada balde irá guardar uma variável *maxSharedTable (*abreviaremos para *maxST*). Para cada balde, esta variável guarda qual a subtabela de maior índice ligada a este balde por uma chave qualquer. Por exemplo, se B_1.*maxST* = 3, isto quer dizer que a posição do balde B_1 é partilhada com uma chave y (que desconhecemos qual é), que entretanto foi colocada na subtabela T_3 (porque na altura T_1 devia já estar ocupada).

O valor de z será especificado nas aulas de laboratório.

Então, o valor de z na realidade diz-nos a subtabela onde podemos começar a procurar pela chave x pretendida. Se z=0, na realidade isto indica-nos que a chave x não se encontra na tabela, e que podemos retornar imediatamente (reparem que neste caso não foi necessária nenhuma comparação). Se z=1, isto indica-nos, que caso exista, a chave x que estamos à procura terá que estar obrigatoriamente na subtabela T_1 (e por consequência B1), e por isso teremos que fazer uma única comparação com o conteúdo de B1, para ter a certeza. Se z=2, isto implica que poderá estar na tabela T_2 ou na T_1, começamos por testar B_2, pois será a localização mais provável, mas teremos também de testar B_1, caso B_2 não contenha a chave que estamos à procura.

Inserção em detalhe

Iremos agora explicar como implementar a inserção com mais detalhe. A primeira coisa a perceber é que uma inserção pode ser na realidade uma atualização, caso a chave passada como argumento seja uma chave que já tenha sido introduzida anteriormente.

Por esta razão, depois de ir buscar todos os baldes, a primeira coisa a fazer é determinar o valor de z, o mínimo dos máximos partilhados. Se z=0, então esta é com certeza a adição de uma nova chave e passamos para a fase seguinte. Se z>0, temos de tentar encontrar a chave usando o mesmo método da função de pesquisa. Se a encontrarmos, então quer dizer que a inserção na realidade é uma atualização. Simplesmente atualizamos o valor associado ao balde onde encontrámos a chave, e saímos.

Caso estejamos perante uma inserção, antes de inserir, convém verificarmos o tamanho da tabela. Caso o número de elementos guardados na tabela seja igual ao tamanho da subtabela T_1, isto implica que a tabela já está a ficar um pouco cheia, e que por isso é boa ideia aumentar o tamanho da mesma. Tal como temos feito noutras estruturas de dados, iremos dobrar o tamanho (número primo com aproximadamente o dobro do tamanho) de todas as subtabelas, e infelizmente, ter de reintroduzir todas as chaves/valores, pois as chaves vão todas mudar de posição (mesmo que não mudem de subtabela), e não podemos manter as posições.

Uma vez que estejamos confortáveis com o tamanho da tabela, determinamos a subtabela/balde para inserção, escolhendo o balde vazio com menor i.

$$j = \min\{i | B_i = \emptyset\}$$

Adicionamos a chave e o valor ao balde B_j selecionado, atualizando tudo o que for necessário. No entanto, antes de sair temos de fazer algo crucial, que é atualizar a *maxSharedTable* de todos os baldes ligados pela chave (incluindo o próprio B_j) de acordo com a expressão.

$$para\ todo\ i = 1 \dots L, B_i.maxST = \max(B_i.maxST, j)$$

Isto é o que nos irá permitir fazer as pesquisas sem termos de comparar várias chaves.

Remoção

Para remover uma chave, teremos de usar uma técnica de remoção preguiçosa para não termos de recalcular os *maxST* todos. Existe uma forma de apagar chaves sem recalcular, mas teríamos de mudar a forma como determinamos a 1ª tabela a começar a pesquisa (teríamos de usar algo ligeiramente mais complicado do que o *maxSharedTable*), e achei melhor não complicar.

Começamos por procurar a chave. Caso a chave não exista, não existe nada a remover. Se a chave existir, depois de encontrado o balde, não podemos apagar a chave do balde, mas simplesmente marcar o balde como sendo um balde apagado. Uma forma de marcar o balde como apagado é colocar o seu valor a *null*. O campo *size*, que guarda o número de elementos guardados, deve diminuir, mas devemos ter um segundo campo que guarda o número de elementos marcados que não foram apagados. Esta marcação é importante.

Quando fazemos uma pesquisa de uma chave (através do método get), e encontramos uma chave guardada num balde que foi marcado como apagado, temos de fazer de conta que essa chave não foi encontrada retornando *null*.

Se fizermos uma reinserção da mesma chave, que, entretanto, está num balde apagado, vamos tirar proveito disso, fazendo apenas uma atualização ao valor, e fazendo com que o balde deixe de ser considerado como apagado.

Se no final do processo de remoção, o número de elementos guardados for inferior a 1/4 do tamanho da subtabela T_1, temos de redimensionar o tamanho das subtabelas para aproximadamente metade, reinserindo as chaves de todos os baldes não apagados.

É importante realçar que num redimensionamento, quer seja para cima, quer seja para baixo, baldes marcados como apagados são ignorados, não sendo colocados nas novas subtabelas. A reinserção pode ser também otimizada, uma vez que sabemos que a reinserção usada no redimensionamento corresponde sempre a uma inserção de uma nova chave que ainda não existe na tabela, e que, portanto, não precisa de fazer nenhuma comparação.

UAlshTable<Key,Value> – Implementa uma tabela de dispersão		
	`UAlshTable( Function<Key,Integer> hc2)`	Cria uma tabela vazia. A UAlshTable trabalha com duas funções de hashcode distintas, e aqui recebemos como argumento a segunda função de hashcode a usar. A primeira será a função hashCode da própria linguagem java.
`int`	`size()`	Retorna o número de pares chave-valor guardados na tabela de dispersão
`int`	`getMainCapacity()`	Retorna a capacidade principal atual da tabela de dispersão, i.e. o tamanho da subtabela T_1.

`int`	`getTotalCapacity()`	Retorna a capacidade total da tabela de dispersão, que corresponde à soma da dimensão de todas as subtabelas.
`float`	`getLoadFactor()`	Retorna o fator de carga da tabela (valor entre 0 e 1), i.e. o rácio entre o número de elementos guardados e a capacidade total da tabela.
`int`	`getDeletedNotRemoved()`	Retorna o número de chaves que são consideradas como apagadas, mas que ainda não foram removidas devido à utilização de remoção preguiçosa.[3]
`IUAlshBucket<Key,Value>[]`	`getSubTable(i)`	Retorna a subtabela Ti, que corresponde a um array de IUAlshBuckets. Caso não exista a tabela i, retorna null. [3]
`boolean`	`containsKey(Key k)`	Verifica se uma chave existe na tabela de dispersão. Retorna *true* caso a chave exista (e não tenha sido apagada) e *false* caso contrário.
`Value`	`get(Key k)`	Retorna o valor guardado na tabela de dispersão para a chave recebida como argumento. Caso a chave não exista na tabela (ou tenha sido apagada) é retornado o valor *null*.
`void`	`put(Key k, Value v)`	Insere o valor v na tabela de dispersão associando-o à chave k. 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).
`void`	`fastPut(Key k, Value v)`	Insere o valor v na tabela de dispersão associando-o à chave k. Este método assume que a chave passada como argumento é sempre uma chave que ainda não existe na tabela, para otimizar a operação de inserção.[3]
`void`	`delete(Key k)`	Remove a chave (e o seu valor) da tabela de dispersão. Tentar remover uma chave que não existe (ou que tenha sido apagada) não produz qualquer efeito.

[3] Este tipo de método não é normalmente tornado público, mas precisamos dele para efeitos de testes automáticos.

`Iterable<Key>`	`keys()`	Devolve um objeto iterável que pode ser usado para percorrer (através de um *foreach*) todas as chaves válidas (não apagadas) da tabela de dispersão.
`void`	`main(string[] args)`	Método *main*, que deverá ser usado para testar os métodos acima.

Requisitos Técnicos

Número de subtabelas e respetivo tamanho

Na implementação deverá usar 5 subtabelas, usando como tamanho os valores primos especificados no ficheiro fornecido. Quando inicializada, a subtabela principal T_1 deverá ter tamanho 37, tendo as restantes tamanho 17, 11, 7 e 5. O tamanho da subtabela principal nunca pode ser inferior a 37.

Evitar comparações desnecessárias

A UAlshTable não gosta mesmo de fazer comparações entre chaves. Existe uma otimização adicional a fazer. Quando inserimos uma chave/valor num balde, guardamos também o hashcode_1 e o hashcode_2 para a chave no balde. Antes de fazer qualquer comparação de chaves k_1 e k_2 com o método *equals*, comparamos os seus *hashcodes*. Por definição, se duas chaves forem iguais, os seus *hashcodes* são exatamente os mesmos. Por isso, se algum dos *hashcodes* não for igual, sabemos imediatamente que as chaves não são iguais, e já não precisamos de chamar o método *equals*.

Método *keys*

No método *keys* pretende-se devolver um objeto iterável, i.e. pode-se fazer um *foreach* sobre ele, para percorrer todas as chaves guardadas na tabela. A ordem pela qual se percorrem as chaves não é normalmente especificada numa tabela de dispersão. Mas neste caso em particular, com o objetivo de mais facilmente testar a implementação, pede-se que iterem as chaves pela ordem das subtabelas, começando pela subtabela T_1 até à T_5, e para cada subtabela pelos índices dos baldes correspondentes, começando pelo índice 0 até ao último índice. Lembramos que apenas devem ser iteradas as chaves válidas, i.e. as que estão guardadas e que não foram, entretanto, marcadas como apagadas.

Método main

Este método deverá implementar os testes usados para testar e comparar os métodos acima. Embora este método não seja validado de forma automática pelo Mooshak será tido em consideração na validação do projecto, e contará para a nota final do mesmo.

Implemente a classe *UAlshTable* no ficheiro *UAlshTable.java* fornecido. Submeta **apenas o ficheiro *UAlshTable.java*** no Problema A.

Parte C – Análise e relatório (3 valores).

Na parte C do projeto, irão analisar alguns aspetos da eficiência das estruturas de dados implementadas no problema A e B.

Relativamente ao problema A, existem dois aspetos interessantes a testar de forma empírica. O primeiro é o balanceamento da árvore. Para testar o balanceamento, determine um método

para calcular a profundidade mínima para qualquer caminho a partir de um nó, e outro para calcular a profundidade máxima de qualquer caminho a partir de um nó. Estes métodos não precisam de ser eficientes.

Uma vez implementado, crie uma UAlgTree e insira um número muito elevados de chaves aleatórias (por exemplo, 100 000). Depois de construída a árvore, determine o rácio entre a profundidade máxima e a mínima da árvore. Convém fazer isto várias vezes.

Faça o mesmo teste, mas fazendo agora com que as chaves sejam introduzidas de forma crescente (ou decrescente). O que pode concluir destes testes?

O segundo aspeto é a eficiência da árvore em situações onde tiramos partido de trazer as chaves para cima. Comece por determinar, usando testes de razão dobrada, a complexidade temporal assimptótica de pesquisas numa árvore já muito grande, para chaves que são escolhidas para pesquisa de forma aleatória seguindo uma distribuição uniforme.

De seguida faça o mesmo estudo, mas desta vez deverá usar um teste em que 20% das chaves são pesquisadas em 80% das vezes, e as restantes 80% das chaves são pesquisadas nas restantes vezes (esta técnica é inspirada na regra de Pareto). Compare a performance (em termos de velocidade média de pesquisa) e a complexidade temporal assimptótica com a situação anterior.

Relativamente ao problema B, é pouco interessante avaliar a complexidade temporal assimptótica, uma vez que é fácil perceber que será sempre constante (no pior dos casos, mas com uma probabilidade muito baixa, temos 5 comparações).

Para percebermos a eficácia da tabela em evitar fazer comparações, vamos contabilizar o número de vezes em que a tabela teve de comparar 2 chaves. Sempre que é feita uma comparação esse contador aumenta. Temos de ter também um contador para contabilizar o número de chamadas a métodos de pesquisa, e a chamadas a métodos de put (o fastPut não deve ser contabilizado, porque é um método interno usado no redimensionamento). Implementem também um método para fazer reset aos contadores (dá jeito).

Faça testes, inserindo um número muito elevado de elementos (ex: 100 000) numa *UAlshTable*, e determine o número de comparações médio por inserção efetuado.

Na *UAlshTable* já cheia, faça agora muitas pesquisas, e determine o número de comparações médio por pesquisa de uma chave existente na tabela, e o número de comparações médio por pesquisa de uma chave inexistente na tabela. Analise os resultados, e indique o que pode concluir acerca da *UAlshTable*.

Os testes efetuados, bem como a sua análise, e as conclusões devem ser escritas num relatório (com no máximo 10 páginas), e deverão submeter o relatório juntamente com o código do projeto final (incluindo todas as classes, todos os ficheiros *main*, e todos os testes efetuados) na tutoria.

Condições de realização e avaliação

O projeto deve ser realizado em grupos de 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. Projetos obtidos a partir de ferramentas de geração automática de código, como o ChatGPT, Copilot, ou Claude, serão também anulados. 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 5 de dezembro.** 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 6 de dezembro**. As validações/discussões do projeto terão lugar na semana seguinte, de 8 a 12 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 utilizada. Não é necessário o registo para quem já se registou no 1.º projeto, podendo usar o mesmo *username* e *password*.

Deverão introduzir o vosso número de aluno e submeter. Irá ser gerada uma password que vos será enviada por email. Caso não recebam a password, verifiquem a vossa caixa de spam, e entrem em contacto com o corpo docente.

Uma vez criado o registo poderão fazer login no sistema Mooshak com o vosso número de aluno e a password recebida. O link para o sistema Mooshak é o seguinte:

http://deei-mooshak.ualg.pt/~dshak/