UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2020-2021

Projecto 3 – Tabelas de Símbolos

Introdução

No 3.º Projecto da cadeira de AED pretende-se explorar a criação e a utilização de tabelas de símbolos, com particular foco nas árvores de pesquisa binária e em *Treaps*.

Problema A: Implementação de Treap (17 valores)

O que é que chama ao cruzamento de uma binary search tree com um *heap*? Um *Treap*. Um *Treap* é uma árvore binária de pesquisa, em que cada nó é composto por uma chave, um valor, e uma prioridade (para além de dois filhos). Um *Treap* não só garante as propriedades de uma árvore binária para as suas chaves (chaves(nós à esquerda do nó) < chave(nó) < chaves(nós à direita do nó)) mas também garante as propriedades de um max-heap para as suas prioridades (prioridade(nó) >= prioridade(filhos do nó)). Na figura seguinte podemos ver um exemplo de um *Treap* (os números marcados com K correspondem a chaves, os números marcados com P correspondem às prioridades).

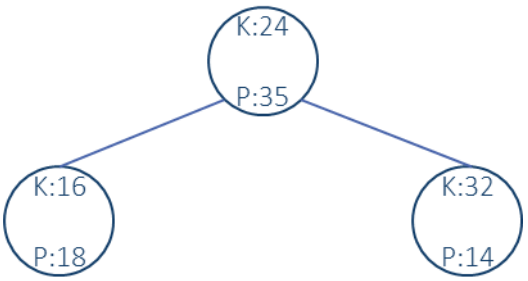

Figura 1. Exemplo de Treap. A etiqueta K representa chaves e a etiqueta P representa prioridades.

Esta estrutura de dados híbrida tem como objetivo lidar com um dos problemas principais de uma árvore binária de pesquisa: se os elementos forem introduzidos por uma má ordem (ex: por ordem crescente), irá dar origem a uma árvore muito mal balanceada, estragando por completo a eficiência do algoritmo. Para o resolver, esta estrutura de dados vai tirar partido das propriedades de um *heap*, e usando número aleatórios, vai tentar reestruturar a árvore, de modo que a árvore final obtida seja semelhante à arvore que se obteria colocando os elementos por ordem aleatória. A forma como isto é feito será detalhada mais à frente.

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

Treap<Key extends Comparable<Key>,Value> – Implementa uma tabela de símbolos ordenada, usando um *Treap*.	
`Treap()`	Cria um treap vazio.

	`Treap(Random r)`	Cria um *Treap* vazio. Recebe como argumento um gerador de números aleatórios, que deverá ser usado dentro do *Treap* para gerar as prioridades atribuídas a cada nó[1].
`int`	`size()`	Retorna o número de pares chave-valor guardados no *Treap*
`boolean`	`containsKey(Key k)`	Verifica se uma chave existe na *Treap*. Retorna *true* caso a chave exista e *false* caso contrário.
`Value`	`get(Key k)`	Retorna o valor guardado na tabela de símbolos para a chave recebida como argumento. Caso a chave não exista na tabela é retornado o valor *null*.
`void`	`put(Key k, Value v)`	Insere o valor v no *Treap* associando-o à chave k. Caso a chave já exista na tabela é feita uma actualização ao seu valor. Inserir o valor *null* numa tabela de símbolos 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.
`Treap[]`	`split(Key k)`	Dada uma chave K, devolve um array com dois *Treaps*, que resultam de um split (separação do Treap original em dois) usando a chave recebida.
`Key`	`min()`	Devolve a menor chave do *Treap*
`Key`	`max()`	Devolve a maior chave do *Treap*
`void`	`deleteMin()`	Remove a menor chave do *Treap*. Caso não exista, nada acontece.
`void`	`deleteMax()`	Remove a maior chave do *Treap*. Caso não exista, nada acontece.
`int`	`rank(Key k)`	Devolve o número de chaves que existem no *Treap*, que são menores que a chave k recebida.
`int`	`size(Key min, Key max)`	Devolve o número de chaves que estão no *Treap*, que estejam entre a chave min (inclusive) e entre a chave max(inclusive).
`Key`	`select(int n)`	Devolve a n-ésima menor chave que está no *Treap*. Por exemplo, para n=0, devolve a menor chave, para n=3 devolve a quarta menor chave. Caso não exista tal chave, deve ser retornado null.
`Iterable<Key>`	`keys()`	Devolve um objecto iterável que pode ser usado para percorrer (através de um foreach) todas as chaves do *Treap*. 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 do *Treap*. Os

[1] Este método será usado para testar a vossa implementação de alguns métodos, tendo controlo sobre a geração de números aleatórios.

		valores devem ser iterados pela mesma ordem que as chaves.
`Iterable<Integer>`	`priorities()`	Devolve um objeto iterável que pode ser usado para percorrer as prioridades de todos os nós do *Treap*. As prioridades devem ser iteradas pela mesma ordem que as chaves.
`Iterable<Key>`	`keys(Key min, Key max)`	Recebe duas chaves, e devolve um objeto iterável que pode ser usado para percorrer (através de um foreach) todas as chaves do *Treap* que estão entre a chave min (inclusive) e a chave max (inclusive). As chaves devem ser percorridas obrigatoriamente da menor para a maior chave.
`Treap`	`shallowCopy()`	Retorna uma cópia superficial do *Treap*. Ou seja, retorna um novo *Treap* com uma estrutura semânticamente equivalente ao *Treap* recebido, mas sem fazer uma cópia dos objectos (chaves e valores) que estão no Treap.
`void`	`main(string[] args)`	Método main, que deverá ser usado para testar os métodos acima.

Objectivos/Requisitos Técnicos

Método get

O método *get* deverá funcionar como o método get tradicional de uma árvore de pesquisa binária. Comparamos a chave que estamos à procura com a chave que está no nó, se for igual, encontrámos o nó que estava à procura, se for menor temos de continuar à procura para a esquerda, se for maior temos de continuar à procura para a direita. Se chegarmos a um nó nulo sem encontrar a chave, a chave não existe na árvore.

Método *put*

É no método *put* que um *Treap* se distingue de forma considerável de uma árvore binária tradicional. A primeira fase do método é equivalente à operação tradicional:

1) Procurar posição de inserção na árvore (olhando para as chaves)
2) Se a chave já existe, é apenas um update, nada a adicionar.
3) Se a chave não existe, quando chegarmos a um filho vazio, encontrámos o sítio de inserção ideal. Criamos um novo nó, atribuindo-lhe uma prioridade aleatória[2]
4) Esse novo nó deve ser adicionado ao *Treap* (definindo-o como filho do pai).
5) O novo nó não tem filhos, mas tem um pai, cuja prioridade pode não estar a respeitar as propriedades de um max-heap. Temos de comparar a prioridade do filho com a prioridade do pai, e se for superior, tem que haver uma troca, através de uma operação de rotação. Se o filho tiver prioridade <= prioridade do pai, não é preciso trocar nada (ver Figura 2).

[2] Deve-se gerar um número aleatório com o maior intervalo possível, para minimizar a probabilidade de dois nós com a mesma prioridade. No entanto, o intervalo de todos os inteiros já é suficientemente grande. Assim sendo, recomenda-se a utilização de uma variável do tipo inteiro para representar a prioridade e o Random.nextInt() do Java para gerar a próxima prioridade aleatória. Caso o *Treap* tenha sido inicializado com um gerador de números aleatórios, deve usar-se esse gerador para a geração.

6) As operações de rotação a aplicar são equivalentes às operações de rotação de uma red-black tree. Temos uma rotação para esquerda, e rotação para a direita. A rotação para a esquerda (ver Figura 3), é quando o nó filho está à direita do pai, e tem que ser rodado para cima. A rotação para a direita (ver Figura 4) é usada quando o nó filho está à esquerda do pai, e tem que ser rodado para cima.
7) Quando rodamos um filho para cima (e o tornamos pai), é possível que agora o novo nó pai não esteja alinhado em termos de prioridade com o seu pai. Por isso temos de voltar a repetir o processo 5, até chegarmos à raiz da árvore, onde já não há nada para rodar (tal como é feito num heapify bottom-up).

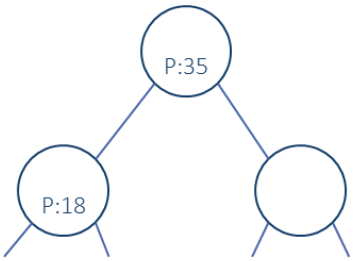

Figura 2. Não existe rotação a fazer

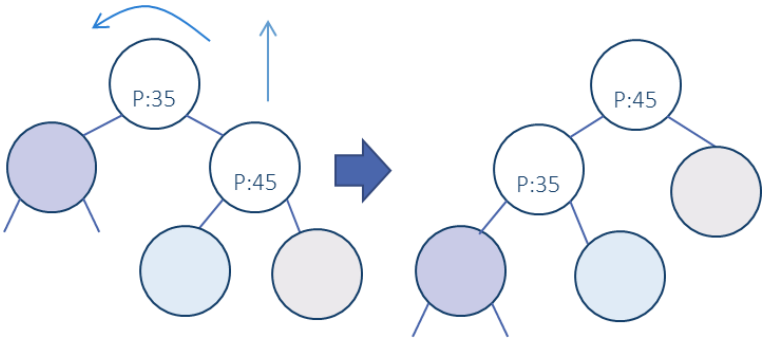

Figura 3. Rotação para a esquerda (o filho a subir está à direita). O nó filho P:45 é movido para o lugar do pai. O pai passa a ser o seu filho esquerdo.

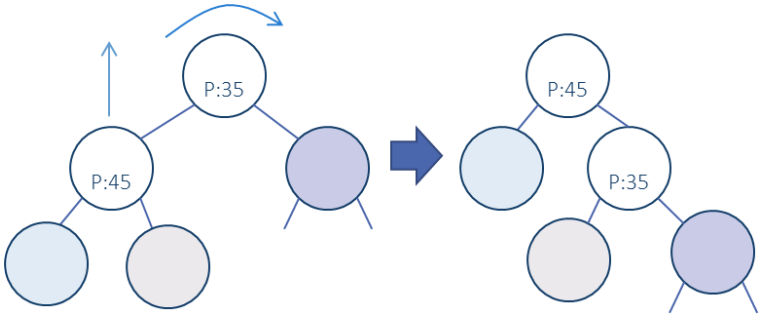

Figura 4. Rotação para a direita (o filho a subir está à esquerda). O nó filho P:45 é movido para o lugar do pai. O pai passa a ser o seu filho direito.

A razão pela qual isto funciona, é que a operação de rotação é uma operação local que preserva a ordem relativa das chaves, mantendo as restrições de uma árvore de pesquisa binária. Ao mesmo tempo, estamos a mover nós de forma que também passem a respeitar as propriedades de um max-heap para os valores das prioridades. No final da operação de put vamos ter uma árvore binária que respeita ambas as propriedades. Como as prioridades são determinadas de

forma aleatória, isto irá corresponder a uma árvore em que a estrutura local é equivalente à de uma árvore em que os elementos foram introduzidos de forma aleatória.

Método delete

Para apagarmos uma chave, a primeira coisa a fazer é encontrar a chave a ser removida. Se não for encontrada tal chave, não existe nada a remover.

Uma vez encontrada a chave, a remoção consiste em eliminar o nó que a contém. Quando um nó a remover é um nó folha (i.e. não tem filhos), o processo de remoção é trivial. Basta fazer com que o seu pai substitua o seu ponteiro por *null*, e atualize o seu tamanho.

Quando o nó a remover não é um nó folha, temos de ter mais trabalho, pois não podemos simplesmente eliminar o nó. Num *Treap*, isto pode ser feito movendo o nó para o fundo da árvore. Para o fazer, podemos tornar a sua prioridade -INF, e fazer um processo semelhante a um heapify top-down para o levar para o fundo. Comparamos o nó com o maior dos filhos[3], e fazemos a rotação correspondente para trocar o maior filho com o pai. Repetimos o processo até chegar ao fim da árvore, e já não houver mais filhos para trocar. Quando chegamos ao fim da árvore, agora sim já podemos eliminar o nó com segurança.

Método split

O método split recebe uma chave e irá dividir o *Treap* em dois *treaps* mais pequenos. O primeiro é um *Treap* com todas as chaves inferiores à chave recebida. O segundo é um *Treap* com todas as chaves superiores à chave recebida. Os *Treaps* são devolvidos num array de 2 elementos, correspondendo o índice 0 ao Treap com as chaves menores.

A implementação deste método torna-se relativamente simples com a utilização de um Treap. Começamos por procurar a chave no Treap original (caso exista). Caso não exista, podemos sempre usar colocar um novo nó com a chave. De qualquer forma, uma vez encontrado o nó com a chave, devemos aumentar a sua prioridade para um valor +INF, e fazer as rotações necessárias para esse nó ser transportado para a raiz da árvore. Uma vez feito este processo, iremos ter um Treap correspondente a um nó com a chave recebida, e dois filhos correspondentes a um Treap com chaves menores, e outro Treap com as chaves maiores. Basta apenas colocar esses dois nós em 2 Treaps novos, e devolvê-los como a solução esperada.

Outros métodos

Uma grande parte dos métodos a implementar não necessita da utilização das prioridades e funciona da mesma forma que funcionaria para uma árvore de pesquisa binária. Pode numa primeira fase fazer uma implementação de uma árvore de pesquisa binária, para conseguir implementar e testar todos os restantes métodos.

Testes à eficiência

Para testar a eficiência de um Treap, podemos focar-nos principalmente nos método get, put e delete. Utilize métodos empíricos, e ensaios de razão dobrada para determinar a eficiência do método get num Treap com um elevado número de elementos. Deve conseguir demonstrar que a eficiência deste método é independente da forma como o Treap foi preenchido. Se o Trep tiver

[3] Na realidade não precisamos de comparar, pois sabemos que o seu valor é -INF e qualquer outro valor de um filho deverá ser superior.

sido preenchido de forma aleatória, deverá ter uma performance equivalente a ter sido preenchido por uma ordem crescente. Teste ambas as situações. Deverá testar também o caso em que a chave existe dentro do Treap, e o pior caso, quando a chave não existe dentro do Treap.

Não precisa de ser tão exaustivo para os métodos put e delete. Pode apenas testar o funcionamento deles perante o caso mais geral de um Trep com um elevado número de elementos.

Escreva um sumário dos testes e resultados, e uma breve descrição como comentários no ficheiro Java a submeter.

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

Condições de realização e avaliação

O projecto deve ser realizado individualmente. Projectos 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 projecto.

Avaliação em laboratório

O 3.º Projeto tem uma componente explícita de avaliação durante o laboratório. Durante o vosso turno de laboratório a que estão inscritos, na semana de 13 a 17 de Dezembro, terão de participar de forma obrigatória. A vossa participação será avaliada, e poderei pedir-vos para implementarem (ou discutirem a implementação) de um dos métodos mais simples. Esta avaliação vale **3 valores da nota final** do projeto. Quem não comparecer a esta avaliação terá 0 neste componente do projecto.

Mooshak – verificação automática

O código do projecto deverá ser entregue obrigatoriamente por via electrónica, através do sistema Mooshak, **até às 23:59 do dia 19 de Dezembro.** As validações terão lugar na 1.ª semana letiva de Janeiro. Os alunos terão de validar o código juntamente com o docente **durante** o horário de laboratório correspondente ao turno em que estão inscritos, que muito provavelmente decorrerá remotamente via Discord. **A avaliação e correspondente nota do projecto só terá efeito após a validação do código pelo docente.**

A avaliação da execução do código é feita automaticamente através do sistema Mooshak, a partir do início da próxima semana, 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 projecto, podendo usar o mesmo username e password. Para quem ainda não se tenha registrado, poderá fazê-lo usando o seguinte link:

http://deei-mooshak.ualg.pt/~dshak/cgi-bin/getpass-aed21

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/