# **Trabalho Prático 0**: Tree + Heap = Treap

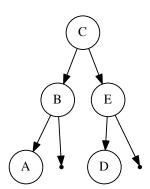
Entrega: 02/04/2015

#### March 9, 2015

### 1 Introdução

Em Algoritmos e Estruturas de Dados II (AEDS II) você deve ter aprendido sobre duas estruturas de dados bastante interessantes: Árvores Binárias de Pesquisa e Heaps. Vamos relembrar como cada uma dessas estruturas funciona.

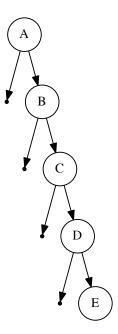
Uma Árvore Binária de Pesquisa é uma árvore enraizada em que cada vértice contém uma chave e até dois filhos: o filho da direita e o da esquerda. Essas árvores tem a propriedade de que a chave de um nó é sempre *maior* que as chaves de todos os nós à esquerda e *menor* que as chaves de todos os nós a direita. Nessas árvores, os vértices que não tem filhos são chamados de *folhas*. Eis um exemplo de árvore binária de pesquisa contendo nós com as chaves "A", "B", "C", "D", "E":



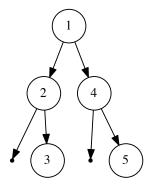
Arvores desse tipo são muito úteis para implementar um tipo abstrato de dados que você já conhece: o *dicionário*. Um dicionário armazena um conjunto de pares chave-valor. Ele executa operações como encontrar o valor associado a uma chave e inserir ou remover pares chave-valor.

Como você deve se lembrar, cada árvore desse tipo tem uma altura, que é o maior número de arestas num caminho entre a raiz e algum vértice folha. O tempo de execução dos algoritmos

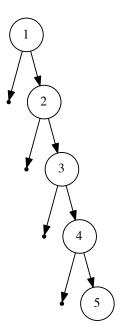
que executam essas operações depende dessa altura. Quanto mais alta a árvore, pior. A árvore abaixo, por exemplo, contém as mesmas chaves da árvore anterior. Porém, ela é tão alta quanto seja possível.



Outra estrutura de dados que é vista nos cursos de AEDS II é o *heap*. Assim como as árvores binárias de pesquisa, heaps são árvores enraizadas em que cada vértice contém uma chave e até dois filhos: um à esquerda e outro à direita. A diferença entre heaps e árvores binárias de pesquisa é a restrição imposta sob as chaves. Em um heap, a chave de um vértice é sempre menor que as chaves de todos os vértices que são descendentes dele. Por descendentes, nos referimos aos filhos do vértice, aos filhos desses filhos, os filhos dos filhos desses filhos e assim por diante. Eis um exemplo de heap construído com as chaves 1, 2, 3, 4, 5.



Heaps também padecem do problema de serem potencialmente muito mais altos do que o necessário. Considere o heap acima, por exemplo. Com as mesmas chaves que foram utilizadas para construí-lo, é possível obter o seguinte heap.

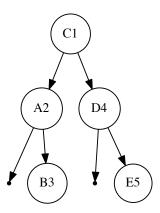


Para evitar que árvores binárias de pesquisa e heaps fiquem muito mais altos do que o necessário, várias estratégias de *balanceamento* foram propostas ao longo dos anos. Você deve ter visto pelo menos uma delas em AEDS II. Talvez essas estratégias tenham até lhe causado pesadelos.

A ideia por trás dessas estratégias é adaptar os algoritmos que manipulam arvores de pesquisa e heaps para que essas estruturas de dados nunca fiquem altas demais. Essas adaptações normalmente consistem em manipular alguns ponteiros demoníacos em cada chamada recursiva de modo a evitar que um dos dois lados de um vértice fique muito mais alto que o outro.

Nesse trabalho, você verá que um pouco de sorte é tudo o que é necessário para que não seja preciso utilizar essas estratégias de balanceamento. Para tal, você implementará uma estrutura de dados que é ao mesmo tempo uma árvore binária de pesquisa e um heap. Essa estrutura é conhecida pelo nome não muito criativo de "treap" (do inglês "tree" + "heap").

Sem mais delongas, eis um exemplo de treap.



Cada vértice em um treap contém não uma, mas sim *duas* chaves. Para evitar ambiguidades, vamos chamar uma dessas chaves de chave e a outra de prioridade. Treaps se comportam como árvores binárias de pesquisa quando você analisa apenas as chaves. E quando você olha para as prioridades, eles se comportam como heaps. O treap acima, por exemplo, foi construído com base no conjunto de pares chave-prioridade A2, B3, C1, D4, E5 em que as letras denotam chaves e os números inteiros são as prioridades.

A combinação de árvores binárias de pesquisa e heaps é muito poderosa. Para que você entenda o porque, faremos algumas observações sobre essas estruturas de dados completamente geniais.

A primeira observação é que ao contrário de árvores binárias de pesquisa e heaps, os treaps são *únicos*. Isso significa que dado um conjunto de pares chave-prioridade em que nem chaves nem prioridades se repetem, só há um *treap* que pode ser construído. Nós não apresentaremos uma prova formal desse fato aqui, mas indicaremos a intuição por trás da prova.

Suponha que você queira construir um treap a partir de um conjunto de pares chave-prioridade. Se você fosse construir uma árvore binária de pesquisa, qualquer chave poderia servir para ser colocada na raiz. Mas como se trata de um treap, só há uma possibilidade: a chave que está no par chave-prioridade de *menor* prioridade. No caso da figura acima, estamos falando do par C1. Agora restaram vários pares chave-prioridade e eles naturalmente precisam de ir parar em algum lugar. Nesse ponto, o fato de que um treap é uma árvore binária de pesquisa ajuda. Sabemos que os pares

chave-prioridade em que a chave é *menor* que a chave da raiz precisam ir para o lado esquerdo, enquanto os demais pares precisam ir para o lado direito. No caso da figura, isso significa que A2 e B3 precisam aparecer do lado esquerdo da raiz, enquanto D4 e E5 precisam aparecer do lado direito. Afinal, A e B vem antes de C no alfabeto. Agora é preciso construir um sub-treap com base nos pares chave-prioridade em cada lado. Mas isso é fácil, porque o problema é idêntico ao de construir um treap do zero. Então basta selecionar o par que tem menor prioridade e continuar o processo recursivamente.

A segunda observação sobre treaps tem a ver com a altura que essas árvores podem atingir. A ideia é a seguinte: suponha que as prioridades sejam escolhidas de maneira *aleatória*. Será que as chances de se obter um treap muito alto são grandes? A resposta é que não. De fato, usando um pouco de probabilidade, é possível mostrar que, em média, um treap construído dessa forma terá altura  $O(\log n)$ , em que n é o número de pares chave-prioridade. Você deverá assumir que os treaps sendo manipulados tem essa altura para as análises de complexidade. No entanto, probabilidade não é um pré-requisito de Algoritmos e Estruturas de Dados III. Portanto, não apresentaremos a prova aqui. No entanto, pode ser interessante ao leitor enumerar alguns exemplos de treaps a fim de se convencer do seguinte fato: para um mesmo conjunto de chaves, treaps muito altos são bem raros.

Nesse ponto, é provável que o leitor esteja se perguntando: como operações como inserção, remoção e busca de chaves podem ser implementadas num treap? E além disso, como essas operações podem ser implementadas sem a necessidade de balanceamento? Bom... Essa não seria a especificação de um trabalho prático se nós contássemos como fazer tudo:)

### 2 Código

Você deve implementar, em linguagem C, um tipo abstrato de dados *treap* em que chaves e prioridades sejam números inteiros. Sua implementação deve suportar as seguintes operações:

- Fusão: Dados *dois* treaps A e B tais que qualquer chave de A é menor do que qualquer chave de B, retorne um treap C contendo os pares de chave-prioridade de A e os de B. Tanto A quanto B deixam de existir no processo, o que significa que C pode ser obtido modificando ponteiros para nós filho que aparecem em A e B.
- Corte: Dado um treap A e uma chave k que não necessariamente aparece em A, divida o treap em dois treaps B e C tais que qualquer chave de B seja menor ou igual a k e qualquer chave de C seja maior que k.
- Inserção de um par chave-prioridade na árvore.
- Remoção de um par chave-prioridade da árvore.
- Detecção de se uma certa chave aparece no treap.

Seu tipo abstrato de dados deve alocar os nós desse treap utilizando as funções malloc ou calloc da biblioteca padrão de C. Tome cuidado para liberar toda a memória que foi alocada

utilizando a função free. Se você se esquecer disso e deixar vértices voando por aí quando o programa terminar de executar receberá menos pontos. Existe uma ferramenta chamada *valgrind* que detecta esse tipo de erro. Use-a:)

A entrada do trabalho prático consistirá numa séries de operações de busca, inserção e remoção que deverão ser realizadas com um treap. Você também deve escrever código que leia a descrição de quais operações devem ser realizadas a partir da entrada e imprima a saída de acordo. Apresentaremos uma descrição mais precisa de entrada e saída na seção a seguir.

Eis uma observação importante: você deve implementar inserção e remoção com base nas duas primeiras operações. Para inserir um par chave-prioridade (k,p) você pode usar um corte para quebrar o treap em um treap A contendo chaves menores que k e um treap B contendo chaves maiores. Então você pode fundir A ao treap de um único nó que contém (k,p) e depois fundir o resultado com B, obtendo um único treap de novo. Essa observação é interessante porque tanto fusões quanto cortes podem ser implementados de maneira simples.

Já para remover um vértice que contenha uma chave k, você pode utilizar dois cortes para separar o treap original em três treaps A, B, e C tais que A contenha vértices com chaves *menores* que k, B contenha um único vértice com chave k e C contenha vértices com chaves *maiores* que k. Após fazer isso, basta eliminar o único vértice do treap B e fundir os treaps A e C.

#### 3 Entrada e Saída

Seu programa deverá ler a entrada da entrada padrão (stdin) e gravar a saída na saída padrão (stdout). A entrada começa com um inteiro N ( $1 \le N \le 100000000$ ) que indica o número de operações que devem ser processadas. Em seguida há N linhas, cada uma descrevendo uma operação. As seguintes operações são possíveis:

**INSERT** k p Insere um nó no treap com chave k e prioridade p. Não imprima nada na tela. Se já houver um nó com a mesma chave no treap, não faça nada.

**REMOVE** k Remove o vértice que contem a chave k, se existir. Não imprimima nada na tela.

**LOCATE** k Encontre o vértice cuja chave é k e imprima o caminho da raiz até ele numa linha. Se o vértice não existir, imprima uma linha contendo "-1". O caminho deve ser especificado por uma sequência de caracteres  $c_1c_2...c_n$  em que cada caractere  $c_i$  pode ser L ou R. O caractere L indica que você tomou o vértice a esquerda e o caractere R indica que você seguiu pelo filho da direita. No treap que foi utilizado como exemplo anteriormente, o caminho LR especifica o vértice R especifica o vértice R especifica o vértice R0.

Em todos os casos,  $0 \le k \le 10^9$  e  $0 \le p \le 10^9$ . Lembre-se de **seguir rigorosamente** a especificação de entrada e saída acima, pois a correção será automatizada. Cada linha na saída deve terminar em \n. Não inclua outros espaços em branco nem termine linhas com \r\n. NÃO inclua system ("pause"); em seu código. Isso pode fazer com que ele não execute adequadamente no PRATICO. A seguir, apresentamos alguns exemplos de entrada e saída.

ENTRADA	SAÍDA
14	RL
REMOVE 4	LR
INSERT 3 1	R
INSERT 2 6	-1
INSERT 1 4	
INSERT 11 2	LLR
INSERT 8 3	
INSERT 9 5	
LOCATE 8	
LOCATE 2	
LOCATE 11	
REMOVE 3	
LOCATE 3	
LOCATE 11	
LOCATE 2	

Table 1: Toy example: teste o qual o TP deve gerar a saída correta para ser elencado para uma entrevista.

ENTRADA	SAÍDA
14	-1
LOCATE 3	
INSERT 3 1	-1
LOCATE 3	
REMOVE 3	
LOCATE 3	
9	LL
INSERT 2 4	
INSERT 3 3	L
INSERT 4 2	
INSERT 5 1	
LOCATE 3	
REMOVE 3	
REMOVE 5	
LOCATE 4	
LOCATE 2	

Table 2: Outros exemplos de teste.

### 4 O que entregar

Você deve submeter uma documentação de até 10 páginas contendo uma descrição de como cada uma das operações foi implementada, além de uma análise de complexidade de tempo dos algoritmos envolvidos e de complexidade de espaço da estrutura de dados. Siga as diretrizes sobre como fazer uma documentação que foram disponibilizadas no portal *minha.ufmg*.

Além da documentação, você deve submeter um arquivo compactado no formato .tar.gz contendo todos os arquivos de código (.c e .h) que foram implementados. Além dos arquivos de código, esse arquivo compactado deve incluir um makefile. Consulte os tutoriais disponibilizados no minha.ufmg para descobrir como fazer um. Finalmente, lembre-se de não incluir nenhuma pasta no arquivo compactado.

## 5 Avaliação

Seu trabalho será avaliado quanto a documentação escrita e à implementação. Eis uma lista **não exaustiva** de critérios de avaliação utilizados.

#### Documentação

- **Introdução** Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.
- **Solução do Problema** Você deve descrever a solução do problema de maneira clara e precisa. Para tal, artificios como pseudo-códigos, exemplos ou figuras podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é preciso incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando os mesmos influenciem o seu algoritmo principal, o que se torna interessante.
- **Análise de Complexidade** Inclua uma análise de complexidade de tempo dos principais algoritmos implementados e uma análise de complexidade de espaço das principais estruturas de dados de seu programa. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita.
- Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos. Por exemplo: se esse trabalho fosse sobre ordenação, seria interessante mostrar como o tempo de execução de cada algoritmo varia quando o número de items a serem ordenados aumenta. Para tal, um gráfico mostrando o tempo de execução em função do tamanho da entrada pode ser interessante. Você também deve interpretar os resultados obtidos. Comente sobre cada gráfico ou tabela que você apresentar mostrando o que é possível concluir a partir dele.
- **Limite de Tamanho** Sua documentação deve ter no máximo 10 páginas. Todo o texto a partir da página 11, se existir, será desconsiderado.

Guia Há um guia e exemplos de documentação disponíveis no moodle.

#### Implementação

#### Linguagem e Ambiente

- Implemente tudo na linguagem C. Você pode utilizar qualquer função da biblioteca padrão da linguagem em sua implementação, mas não deve utilizar outras bibliotecas. Trabalhos em outras linguagens de programação serão zerados. Trabalhos que utilizem outras bibliotecas também.
- Os testes serão executados em Linux. Portanto, garanta que seu código compila e roda corretamente nesse sistema operacional. A melhor forma de garantir que seu trabalho rode em Linux é escrever e testar o código nele. Há dezenas de máquinas com Linux nos laboratórios do DCC que podem ser utilizadas. Você também pode fazer o download de uma variante de Linux como o Ubuntu (http://www.ubuntu.com) e instalá-lo em seu computador ou diretamente ou por meio de uma máquina virtual como o VirtualBox (https://www.virtualbox.org). Há vários tutoriais sobre como instalar Linux disponíveis na web.

Casos de teste Seu trabalho será executado em um conjunto de entradas de teste.

- Essas entradas não serão disponibilizadas para os alunos até que a correção tenha terminado. Faz parte do processo de implementação testar seu próprio código.
- Você perderá pontos se o seu trabalho produzir saídas incorretas para algumas das entradas ou não terminar de executar dentro de um tempo limite pré-estabelecido.
   Esse tempo limite é escolhido com alguma folga. Garanta que seu código roda a entrada de pior caso em no máximo alguns minutos e você não terá problemas.
- A correção será automatizada. Esteja atento ao formato de saída descrito nessa especificação e o siga precisamente. Por exemplo: se a saída esperada para uma certa entrada o número 10 seguido de uma quebra de linha, você deve imprimir apenas isso. Imprimir algo como "A resposta e: 10" contará como uma saída errada.
- Os exemplos mostrados nessa especificação são parte dos casos de teste.
- Os testes disponíveis no PRATICO são apenas preliminares. Os monitores executarão os tranalhos nos demais casos de teste.
- Você deve entregar algum código e esse código deve compilar e executar corretamente para, pelo menos, o toy example da tabela 1, que é o primeiro teste do PRATICO. Se isso não ocorrer, a nota do trabalho prático será zerada.

**Alocação Dinâmica** Você deve alocar os vértices do treap dinâmicamente utilizando as funções malloc() ou calloc() da biblioteca padrão ce C. Você também deve liberar *tudo* o que for alocado utilizando free().

*Makefile* Inclua um makefile na submissão que permita compilar o trabalho.

Qualidade do Código Seu código deve ser bem escrito:

- Dê nomes a variáveis, funções e estruturas que façam sentido.
- Divida a implementação em módulos que tenham um significado bem definido.
- Acrescente comentários sempre que julgar apropriado. Não é necessário parafrasear o código, mas é interessante acrescentar descrições de alto nível que ajudem outras pessoas a entender como sua implementação funciona.
- Evite utilizar variáveis globais.
- Mantenha as funções concisas. Seres humanos não são muito bons em manter uma grande quantidade de informações na memória ao mesmo tempo. Funções muito grandes, portanto, são mais difíceis de entender.
- Lembre-se de indentar o código. Escolha uma forma de indentar (tabs ou espaços)
  e mantenha-se fiel a ela. Misturar duas formas de indentação pode fazer com que
  o código fique ilegível quando você abri-lo em um editor de texto diferente do que
  foi utilizado originalmente.
- Evite linhas de código muito longas. Nem todo mundo tem um monitor tão grande quanto o seu. Uma convenção comum adotada em vários projetos é não passar de 80 caracteres de largura.
- Tenha bom senso.

## 6 Considerações Finais

- Essa especificação não é isenta de erros e ambiguidades. Portanto, se tiver problemas para entender o que está escrito aqui, pergunte a nós, monitores.
- A penalização por atraso seguirá será de  $\frac{2^{d-1}}{0.32}\%$ , em que d é o número de dias **úteis** de atraso.
- Você não precisa de utilizar uma IDE como Netbeans, Eclipse, Code::Blocks ou QtCreator para implementar esse trabalho prático. No entanto, se o fizer, não inclua os arquivos que foram gerados por essa IDE em sua submissão.
- Esteja atento ao tamanho da entrada e às complexidades de seus algoritmos. Uma solução que implemente as operações com complexidade linear, por exemplo, pode ser lenta demais para rodar algumas das entradas que são possíveis dentro dos limites estabelecidos. Além disso, uma solução de complexidade linear é tão ruim quanto a solução trivial de armazenar todos os pares chave-prioridade em uma lista encadeada. Não faz muito sentido implementar uma árvore binária de pesquisa que seja mais lenta que a solução trivial...
- Seja honesto. Você não aprende nada copiando código de terceiros nem pedindo a outra
  pessoa que faça o trabalho por você. Se a cópia for detectada, sua nota será zerada e os
  professores serão informados para que tomem as devidas providências.
- É bastante provável que o código que implementa o treap fique muito menor do que essa especificação :)