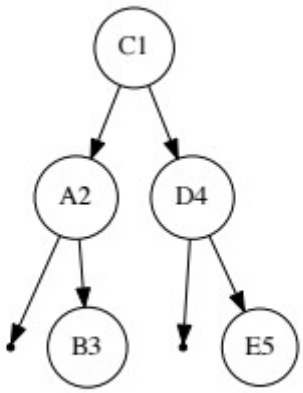
Trabalho Prático 0: Tree + Heap = Treap

1. Introdução

Nesse trabalho temos como objetivo a implementação da estrutura de dados conhecida como Treap. Essa estrutura é a mistura entre uma árvore binária e um heap, e possui grande utilidade, pois forma uma árvore que geralmente é bem balanceada, e isso torna operações de inserção, busca, remoção, etc bem eficientes. Nessa estrutura, cada elemento possui dois valores associados: uma chave e uma prioridade. Se examinamos as chaves, a estrutura funciona como uma árvore binária, e se examinamos as prioridades, se comporta como um heap. Abaixo temos um exemplo de estrutura desse tipo, no qual as letras representam as chaves, e os números representam as prioridades:



Exemplo dado na especificação

Para correta implementação da estrutura, devemos ser capazes de inserir pares chave-prioridade no treap, removê-los, ou encontrá-los dada sua chave. Para facilitar as funções de remoção e inserção, criaremos funções de corte e fusão. O corte recebe uma chave, e separa um treap em dois subtreaps, sendo que um deles contém todos os pares cuja chave seja menor ou igual à chave passada para o corte, e o outro contém todos os pares cuja chave seja maior. A fusão recebe dois treaps A e B, sendo que todas as chaves de A são menores que todas as chaves de B. Essa função então une A e B e retorna o resultado.

2. Solução do problema

Para a implementação do modelo dado, primeiramente criamos uma estrutura que armazena um valor de chave, um valor de prioridade, e ponteiros para a direita e a esquerda do elemento atual. Essa é a nossa parte estrutural.

A parte mais relevante do trabalho consiste na solução dos problemas de localização de uma chave no treap, inserção e remoção de um par, e principalmente fusão e corte de treaps. Examinaremos cada problema individualmente:

- **Localização:** Consiste em procurar uma chave no treap, mostrando o caminho percorrido. Caso a chave não exista no treap, devemos exibir o valor -1. Para isso, nossa função recebe um ponteiro de char no qual vai inserir os caracteres “R” quando for para a direita no treap durante a pesquisa, ou “L” quando for para a esquerda. Uma solução possível seria exibir na saída padrão os caracteres diretamente, à medida que fôssemos navegando pelo treap. Nesse caso, porém, teríamos um problema, pois caso uma chave não existisse, mostraríamos todo o caminho até onde ela se localizaria, e logo depois exibiríamos o -1. Isso estaria em desacordo com a especificação.

  O algoritmo dessa função procede da seguinte maneira: pegamos a raiz do treap. Caso a chave dessa raiz seja igual à chave procurada, retornamos a raiz e não inserimos nada na string com as letras do caminho percorrido. Do contrário, caso a chave dessa raiz seja maior que a chave procurada e essa raiz possua um par à sua esquerda, inserimos a letra “L” na string do caminho, e chamamos a função de localização recursivamente passando o par à esquerda da raiz. Do contrário, caso a chave dessa raiz seja menor que a chave procurada e essa raiz possua um par à sua direita, inserimos a letra “R” na string do caminho, e chamamos a função de localização recursivamente passando o par à direita da raiz. Do contrário, caso a chave da raiz seja igual à chave procurada, retornamos a raiz, pois encontramos a chave. Caso não tenhamos caído em nenhum caso anterior, não encontramos a chave e não temos mais onde procurar, portanto, "apagamos" a string de caminho, printamos -1 na saída e retornamos NULL.
- **Inserção:** A inserção de um par chave-prioridade no treap é feita de forma simples, utilizando as funções de corte e fusão que serão explicadas posteriormente. Caso o treap passado esteja vazio, apenas criamos um elemento nesse treap com a chave e a prioridade passadas. Do contrário, fazemos um corte no treap, separando-o em um treap com chaves menores ou iguais à passada, e outro com chaves maiores. Usamos a função de localização para checar se a chave passada já se encontra no treap. Caso já se encontre, não criamos um novo par chave-prioridade e fundimos as partes do treap total de volta. Do contrário, criamos o novo par com os dados recebidos, fundimos o treap de chaves menores com o novo par, e esse novo treap com o treap de chaves maiores, estando feita assim a inserção.
- **Remoção:** Para remover uma chave do treap, cortamos o treap em dois, sendo que um dos treaps tem valores de chave menores ou iguais à chave a ser removida, e o outro contém valores de chaves maiores. Depois disso, pegamos o treap menor e o cortamos em valores menores que a chave passada e um único par que tem a chave passada (para isso, cortamos ele passando chave - 1 como ponto de corte, pois sabemos que as chaves devem ser valores inteiros). Por fim, liberamos a memória do par que contém a chave passada, se ele existir (caso não exista, essa chave não estava contida no treap) e fazemos a fusão dos treaps de

valores menores e de valores maiores, obtendo o treap sem a chave passada.

- **Fusão:** Temos um treap A e um treap B, sendo que todos os elementos de A tem chave menor que qualquer chave de B. Para fazer a fusão desses treaps, primeiramente checamos se a prioridade da raiz do treap A é maior que a da raiz do treap B. Caso não seja, trocamos os treaps: o treap B se torna o treap A e vice versa. Em seguida, seguimos o treap B procurando o ponto de inserção da raiz do treap A, como uma árvore binária, ou seja: se a chave do par que estamos examinando em B é maior que a chave da raiz de A, examinamos o par à esquerda do atual. Se for menor, examinamos o par à direita. Fazemos isso enquanto a prioridade do par examinado em B for menor que a prioridade da raiz de A. Quando encontrarmos um par com prioridade superior à da raiz de A, colocamos o treap A no lugar desse par, e esse par, junto com seus descendentes, se torna o novo treap A. Repetimos o processo até que a busca pelo local de A leve à uma direção em um par que tenha ponteiro para NULL. Quando isso ocorre, fazemos o ponteiro dessa direção do par encontrado apontar para A, e a fusão está completa.

Pseudo-código:

```
par = B.raiz
while( A != NULL )
      if(par.chave > A.raiz.chave)
            if( atual.direita == NULL )
                  atual.direita = A
                  A = NULL
            else if(par.direita.prioridade > A.raiz.prioridade)
                 troca(A, par.direita)
                 par = par.direita
            else
                  par = par.direita
     else
            if( atual.esquerda == NULL )
                  atual.esquerda = A
                  A = NULL
            else if(par.esquerda.prioridade > A.raiz.prioridade)
                 troca(A, par.esquerda)
                 par = par.esquerda
            else
                  par = par.esquerda

return B
```

- **Corte:** Temos um treap T e uma chave, e devemos dividir esse treap em dois, sendo que um dos treaps resultantes possui apenas chaves menores ou iguais à chave recebida para corte, e o outro possui apenas chaves maiores. Caso a raiz de T tenha chave igual à chave recebida, ele já está cortado, teoricamente. Assim, o menor treap fica sendo o treap T, e o maior fica sendo o subtreap de raiz igual à direita da raiz de T. Depois disso, apenas passamos o valor NULL para o ponteiro para a direita da raiz de T, separando os dois treaps.

  Caso não tenha caído no caso anterior, marcamos a raiz com o ponteiro de "maior" caso sua chave seja maior que a chave passada, ou com o ponteiro de "menor" em caso contrário. A seguir, começamos a varrer o treap. Seguimos o treap pela ordenação de árvore binária (se a chave do par é maior que a chave procurada, vamos para a esquerda, do contrário, vamos para a direita), procurando por pontos onde haja mudança de direção (por exemplo, estamos caminhando para a direita no treap X vezes consecutivas, e encontramos um par que nos faz ir para a esquerda, ou vice versa). Quando houver uma mudança de direção da direita para a esquerda, marcamos o penúltimo par encontrado, que chamaremos de A, como ponto de inserção no menor treap, e inserimos o par à direita de A no treap maior. Em seguida, fazemos com que a direita de A aponte para NULL, e em seguida examinamos o par que estava à direita de A anteriormente. No caso da mudança de direção ser da esquerda para a direita, marcaremos A como ponto de inserção no maior, inserimos o par à esquerda de A no menor, fazemos com que a esquerda de A aponte pra NULL e examinamos o par à esquerda de A anteriormente.

  Quando chegamos em um ponto em que a direção que devemos seguir possui um ponteiro para NULL, ou chegamos na chave procurada, teremos um subtreap com valores menores ou iguais à chave, e outro com valores maiores que a chave. Basta dividí-los e inserí-los nos subtreaps de menores valores e maiores valores obtidos anteriormente.

3. Análise de complexidade

Primeiramente vamos analisar a complexidade de tempo das principais funções implementadas. Lembramos que, segundo a especificação, devemos assumir um treap de altura log(n), ou seja, totalmente balanceado. Consideramos n a quantidade de elementos no treap.

- **Fusão (função fusao):** Na função de fusão, um exemplo do pior caso seria um caso em que teríamos que percorrer toda a altura de B procurando um local para A, e quando chegássemos no último par a ser examinado, ele tivesse prioridade maior que a de todos os elementos de A. Assim, inserimos A no lugar desse par, E teremos agora que varrer toda a altura de A procurando um local para o par restante de B. Dessa forma, teríamos complexidade **O(log n1 + log n2)**, considerando que n1 é o número de elementos de A e n2 é o número de elementos de B.
- **Corte (função corte):** Essa função, em questão de complexidade, se resume a uma busca em árvore binária, pois procura a chave especificada para fazer o corte, e seu pior caso será quando o treap passado for varrido em toda sua altura, ou seja, o corte será **O(log n)**.

- **Localização (função encontrar):** Nessa função, analisamos um par do treap, e em seguida chamamos a função novamente para um dos lados do par atual. Ao fazer isso, descartamos metade dos pares do treap, considerando um treap balanceado. Portanto, teremos a seguinte fórmula de recorrência:

  T(n) = T(n/2) + O(1)

  Utilizando o teorema mestre, teremos que T(n) = **θ(log n)**

- **Inserção (função inserir):** A função de inserção chama a função de corte no treap inteiro (log n) e em seguida chama a função de fusão nos treaps com um elemento (o par a inserir) e no feito de pares com chaves menores que a chave a inserir (log n1 + log 1 = log n1, sendo n1 o número de elementos do treap menor, portanto n1 <= n), e em seguida fundimos também o treap obtido na fusão anterior com o treap com os elementos restantes do treap original (log (n1 + 1) + log n2, sendo que n2 é a quantidade de elementos do treap maior, portanto n2 <= n). No final, para obter a complexidade total da operação de inserção, basta somar as complexidades:

  O(log n + log n1 + log (n1 + 1) + log n2) = **O(log n)**

- **Remoção (função remover):** A função de remoção chama a função de corte no treap inteiro (log n), e em seguida chama o corte no treap de chaves menores ou iguais à chave a ser removida (log n1, sendo n1 o número de elementos do treap menor, portanto n1 <= n). Por fim, fazemos a fusão dos treaps de valores menores e de valores maiores (log n1 + log n2, sendo que n2 é a quantidade de elementos do treap maior, portanto n2 <= n). No final, para obter a complexidade total da operação de remoção, basta somar as complexidades:

  O(log n + log n1 + log n1 + log n2) = **O(log n)**

A complexidade, podemos constatar trivialmente, será **θ(n)**, pois teremos que armazenar na memória um registro para cada chave no treap.

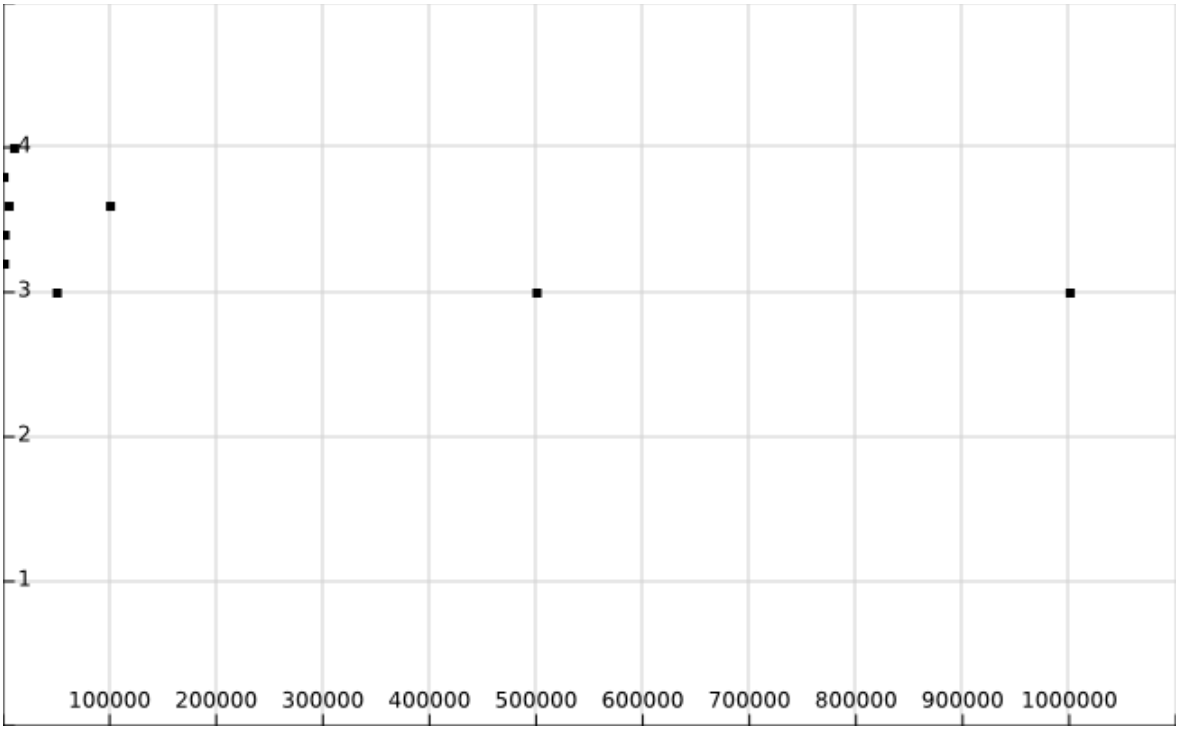
4. Análise experimental

Para a análise experimental do código, foram gerados treaps de variados tamanhos. Para gerar esses treaps, foi utilizado o algoritmo de Fisher-Yates para embaralhar dois vetores com números de 0 até n, sendo n o número de pares no treap, e os valores nesses vetores foram utilizados como chaves e prioridades de cada par. Apos a geração dos vetores, inserimos os pares no treap, pegando a chave do vetor de chaves e a prioridade do vetor de prioridades. Após esse processo estar completo, o programa utiliza as funções de localização, remoção e inserção no treap, medindo o tempo levado por cada operação. Para medir o tempo passado, utilizamos a biblioteca <time.h> para contar o número de clocks necessários para executar cada operação, e usamos a variável CLOCKS_PER_SEC para calcular o tempo tomado por cada operação. Rodamos o programa 5 vezes para cada tamanho de treap, e para determinar o tempo tomado por cada operação, fazemos a média dos tempos obtidos. Os testes foram feitos em uma máquina com processador Core I5 com frequência de 2.6GHz e 4GB de memória RAM.

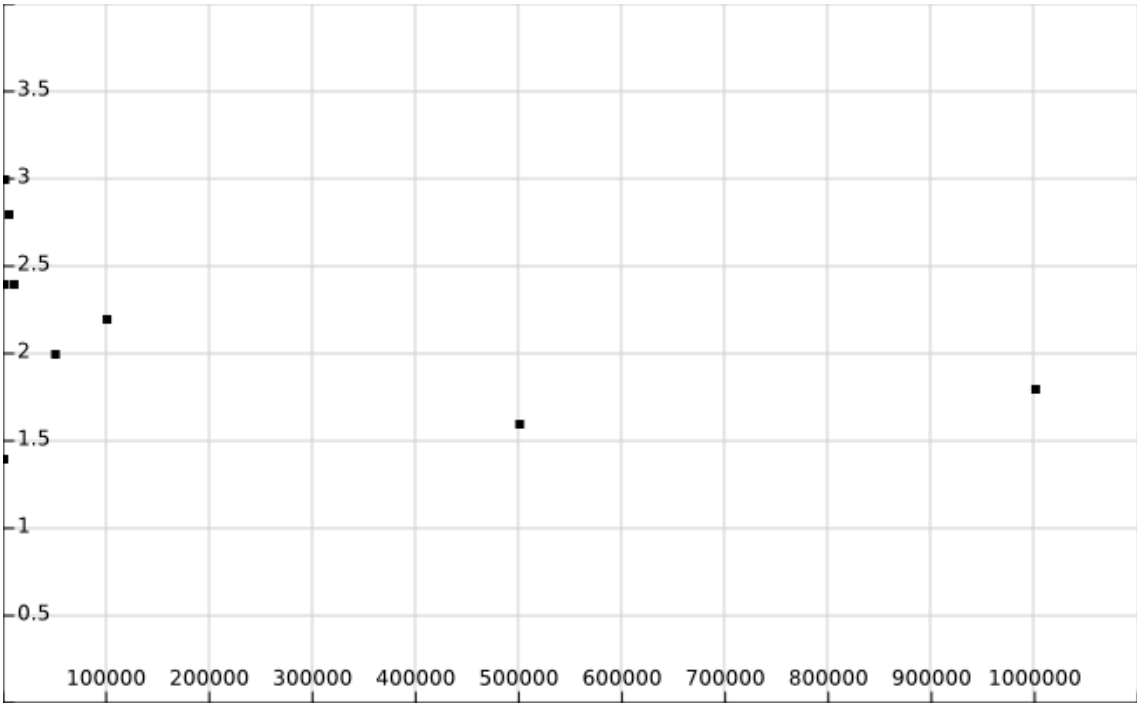
Foram medidos os valores para treaps de tamanhos: 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000.

O eixo horizontal do gráfico representa a quantidade de elementos no treap, e o eixo vertical representa o número de microssegundos tomados pela operação da qual o gráfico trata.

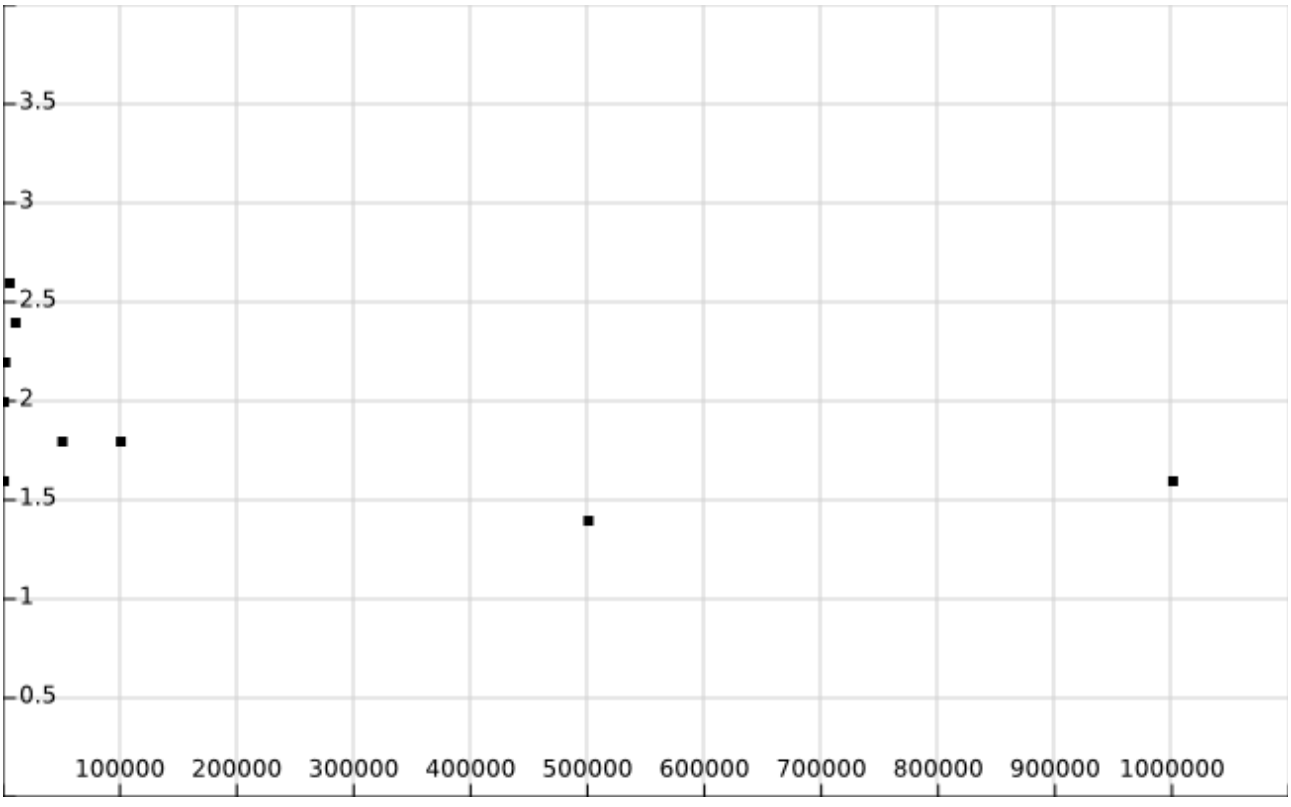
O gráfico (a) mostra os dados para a localização de um elemento no treap, o gráfico (b) mostra para a remoção de uma chave, e o gráfico (c) mostra para a inserção de um par no treap.



(a) Gráfico Tamanho x Tempo para a localização de chave



(b) Gráfico Tamanho x Tempo para a remoção de chave



(c) Gráfico Tamanho x Tempo para a inserção de par chave-prioridade

Na análise experimental, pudemos ver que os resultados obtidos foram consideravelmente diferentes do que esperávamos. Enquanto o que suporíamos seria uma curva sublinear (logarítmica), pois todas as operações eram O(log n), obtivemos gráficos sem formato bem definido, com valores dentro de determinada faixa, qualquer que fosse o tamanho do treap, sem crescimento evidente. Esse comportamento estranho pode ter vários motivos, entre eles:

- O método de benchmarking pode não ter sido preciso o suficiente;
- A criação de pares para inserção pode estar sendo tendenciosa e gerando treaps muito específicos;
- Como na análise de complexidade descartamos todos os termos constantes, eles podem estar sendo mais custosos que os termos não-constantes até os tamanhos de treap avaliados, e as flutuações ocorridas foram flutuações normais de benchmarking, sendo que a complexidade avaliada está sendo desprezível se comparada aos termos constantes.

5. Conclusão

Ao fim do trabalho, pudemos implementar um treap com operações de localização, inserção, remoção, corte e fusão. Pudemos ver que o uso do corte e fusão para as outras operações facilita bastante lidar com essa estrutura, e nos permite criar soluções de boa performance. A análise experimental levou a dados inesperados, porém pudemos constatar que o tempo necessário para as operações em treaps pequenos ou em treaps grandes é bem similar.