UNIVERSIDADE FEDERAL DE LAVRAS

João Pedro Fachini Alvarenga

Implementação de Heap em complexidade linear

LAVRAS-MG

2019

Introdução

Heap é uma estrutura de dados baseada em árvore em que existe uma regra específica que ordena os nós pais em comparação com os seus respectivos nós filhos. Por exemplo, ela poderia ser uma ordem decrescente (Heap máximo) ou crescente (Heap Mínimo). Neste trabalho, no caso, foi utilizado um heap mínimo como objeto de estudo, ou seja, os nós pai são sempre menores que os seus filhos.

Dentro do contexto, a heap é muito utilizada em vários algoritmos como uma fila de prioridade para auxiliar suas implementações, a complexidade assintótica para inserir N elementos nela é O(N*lg(N)), no pior caso. Em alguns casos de algoritmos que utilizam essa estrutura, como o de Dijkstra, sua complexidade impacta diretamente na complexidade do algoritmo.

Inspirado nesse problema, visto que seria interessante a otimização do algoritmo de construção de uma heap, este trabalho é dedicado a projetá-lo em complexidade linear.

Desenvolvimento

A ideia para projeto de um algoritmo de construção de heap em complexidade linear seria então fazer uma diferente abordagem do problema. Na abordagem mais comum, a cada inserção de elementos, ele é colocado na última posição do heap e é chamado um algoritmo para “corrigir” a situação atual do heap, botá-lo em ordem, esse algoritmo é usualmente chamado de “corrige subindo” pois ele vai trocando os nós pai com filhos que estiverem em ordem diferente da do heap (no caso de um heap mínimo, caso o pai for maior que o filho, por exemplo), assim ele vai subindo pela árvore até ela estiver completamente ordenada por assim dizer.

Então, no pior caso de um algoritmo desses, a cada inserção ele haveria que percorrer a altura do heap, como há “N” inserções, isso explica sua complexidade de O(N*lg(N)), por isso para projetar um algoritmo mais otimizado, uma outra abordagem deve ser utilizada.

Uma possível outra abordagem para o problema seria armazenar todos os elementos do heap em primeiro lugar e depois de todos serem armazenados, aplicar algum algoritmo para corrigir o heap em cada nó, assim, seria garantido que no final ele estaria construído perfeitamente. Mas qual algoritmo poderia ser usado? Poderia ser o próprio corrige subindo ?

Algoritmo 1 – Inserção de elementos na Heap Otimizada

Entrada: tamanho n do Heap, n elementos do Heap.
Saída: Heap mínimo.

```
1. INÍCIO
2.         inicializar heap vazio;
3.         PARA i <- 0 até i < n FAÇA
4.                 heap.inserir(elemento);
5.         FIM-PARA
6.         heap.Heapify();
7.         RETORNAR heap;
8. FIM
```

Imagem 1 - pseudocódigo da forma proposta para construção do heap.

Se tivéssemos o heapify como um corrige subindo sendo aplicado em todos os nós, sabendo que em cada nível de um heap há $n/2^{k+1}$ nós, sendo n a quantidade de elementos do heap em questão e k o número do nível em questão, começando de 0, o nível dos nós folhas, até h, nível da raiz. A quantidade de iterações em cada nível do heap seria $(h-k) * n/2^h$, sendo h sua altura, ou seja, $h = log2(n)$. O somatório de todas iterações em todos níveis seria:

$$h * n/2 + (h-1) * n/4 + (h-2) * n/8 + ... + 0 * 1$$

Neste caso, já sabemos a priori que a complexidade deste algoritmo no pior caso seria no mínimo O(n*lg(n)), pois logo no começo do somatório há $h * n/2$ que é a mesma coisa que $log2(n) * n/2$, portanto, essa abordagem não funcionará.

Entretanto, poderia ser feito uma abordagem semelhante só que com um algoritmo diferente aplicado em cada nó para a construção do heap, por exemplo, o conhecido como "Corrige Descendo". O algoritmo corrige descendo se trata de, ao contrário do corrige subindo, partir de um nó e corrigir a árvore formada em baixo dele, trocando os nós filhos com o nó atual, que seria o pai.

Se aplicássemos o corrige descendo em cada nó, a quantidade de iterações em cada nível do heap seria $k * n/2^{k+1}$ e o somatório de todas iterações em todos níveis seria:

$$0 * n/2 + 1 * n/4 + 2 * n/8 + ... + h * 1$$

Obviamente, esse somatório é bem diferente e menor que o anterior, pois em cada iteração de cada nível, ela está em função apenas de k, e não de h, como era

com o corrige subindo. Por fim então, podemos tentar analisar esse algoritmo e tentar provar que ele, de fato, é linear.

Algoritmo 2 - Heapify otimizado

Entrada: vetor H com elementos do futuro heap inseridos, tamanho do vetor.

Saída: Heap mínimo.

```
1.  INÍCIO
2.      PARA CADA noh não folha em H FAÇA
3.          pai <- nohAtual;
4.          i <- filhoAEsquerda(pai);
5.          terminou <- FALSE;
6.          ENQUANTO i possuir filhos e terminou = false
7.              SE H[i] > H[i+1] incrementar i;
8.              SE H[pai] <= H[i] terminou = true;
9.              SENÃO
10.                 trocar(H[pai], H[i]);
11.                 pai <- i;
12.                 i <- filhoAEsquerda(pai);
13.         FIM-ENQUANTO
14.     FIM-PARA-CADA
15.     RETORNAR H;
16. FIM
```

Imagem 2 - pseudocódigo do algoritmo proposto para construção de heap.

Como metade dos nós de uma árvores sempre são folhas, então, um heap tem n/2 nós não folhas, portanto na linha 2 terá $(n/2) + 1$ repetições, as linhas 3 até 5 e a linha 14 terão $n/2$ repetições e a linha 15 acontecerá apenas uma vez, resta saber como será o comportamento do laço que começa na linha 6, que será o laço que comandará o comportamento assintótico do algoritmo.

A cada iteração da linha 6, no pior caso, o corrige descendo fará trocas em todos os níveis da subárvore abaixo do nó atual, isso significa que ela irá repetir o número da altura dessa subárvore que é o chamado acima de k, o nível da altura.

Então nas n/2 iterações será chamado um número que variará entre elas, para facilitar a análise, podemos separar então as repetições pelos níveis comentados no texto acima, pois assim podemos reconhecer um padrão que é dado pela equação $k * n/2^h$ já descrita. Isso significa que, no primeiro nível teríamos n/4

elementos e cada corrige descendo neles iria fazer uma operação que teria apenas uma única iteração, no segundo nível teríamos n/8 elementos e duas iterações em cada, até chegar no último nível que teríamos apenas um elemento e um total de h iterações.

Esse somatório descrito é exatamente o mesmo da análise anterior a imagem do pseudocódigo, que é:

$$\sum_{k=1}^{h} \frac{k \cdot n}{2^{k+1}}$$

Para analisá-lo melhor, podemos então realizar uma sequência de artifícios algébricos, primeiro:

$$\sum_{k=1}^{h} \frac{k \cdot n}{2^{k+1}} = \frac{n}{4} \cdot \sum_{k=1}^{h} \frac{k}{2^{k-1}}$$

Como um somatório finito sempre será menor que o mesmo somatório infinito:

$$\frac{n}{4} \cdot \sum_{k=1}^{h} \frac{k}{2^{k-1}} < \frac{n}{4} \cdot \sum_{k=1}^{\infty} \frac{k}{2^{k-1}}$$

Utilizando mais um artifícios matemáticos para facilitar:

$$\frac{n}{4} \cdot \sum_{k=1}^{\infty} k x^{k-1} = \frac{n}{4} \cdot \frac{d}{dx} \sum_{k=1}^{\infty} x^{k} \text{ , sendo } x = \frac{1}{2}.$$

Agora, por conhecimento de “Séries geométricas”, sabemos que:

$$\sum_{k=0}^{\infty} x^{k} = \frac{1}{(1-x)}$$

Portanto, a derivada ficará:

$$\frac{n}{4} \cdot \frac{d}{dx} \left[\frac{1}{(1-x)}\right] = \frac{n}{4} \cdot \frac{1}{(1-x)^2}$$

Por fim, substituindo x por seu valor de 1/2:

$$\frac{n}{4} \cdot \frac{1}{(1-1/2)^2} = n$$

Desta forma, de acordo com a análise, o limite superior da função que dita o comportamento do algoritmo descrito é linear e, portanto de fato, este algoritmo que utiliza o corrige descendo em cada nós, começando no primeiro nó não folha até o raiz de fato possui um comportamento assintótico linear.

Resultados

Com a implementação do algoritmo em linguagem C++ (verificar código no apêndice A), obteve-se resultados empíricos interessantes, como observados a seguir:

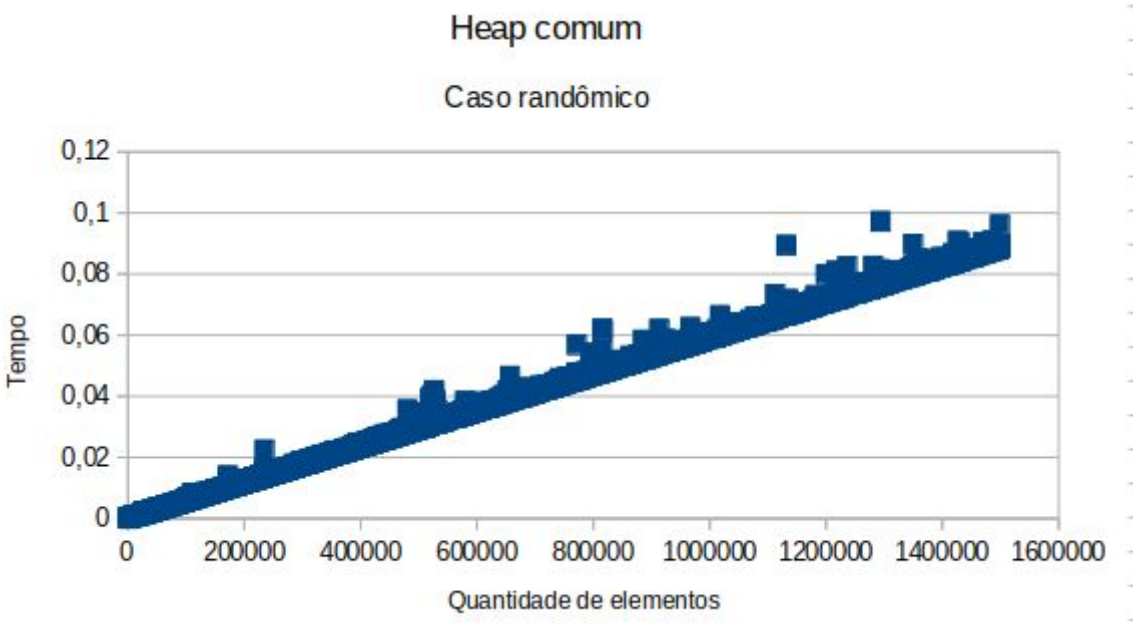

Imagem 3 - Gráfico do comportamento de uma heap comum recebendo elementos aleatórios.

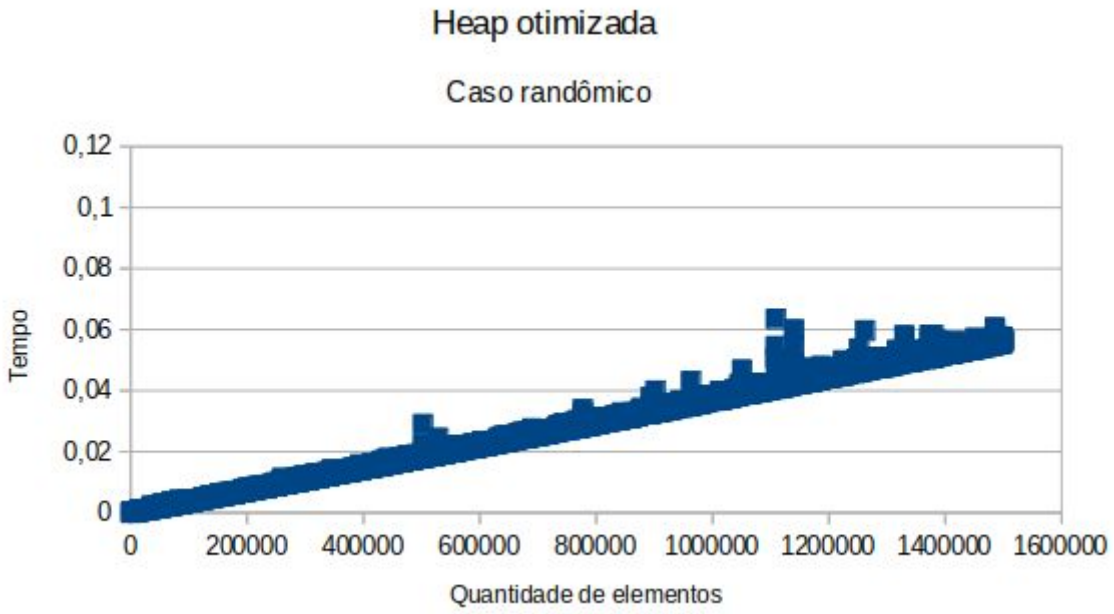

Imagem 4 - Gráfico do comportamento da heap projetada recebendo elementos aleatórios.

Como observado nos gráficos acima, o comportamento da heap otimizada se aproxima, de fato, a um comportamento linear na prática e é muito mais rápida do que a heap comum recebendo a mesma quantidade de dados.

Outro resultado empírico observável foi sobre o comportamento dos dois estilos de abordagem de construção de heap no pior caso.

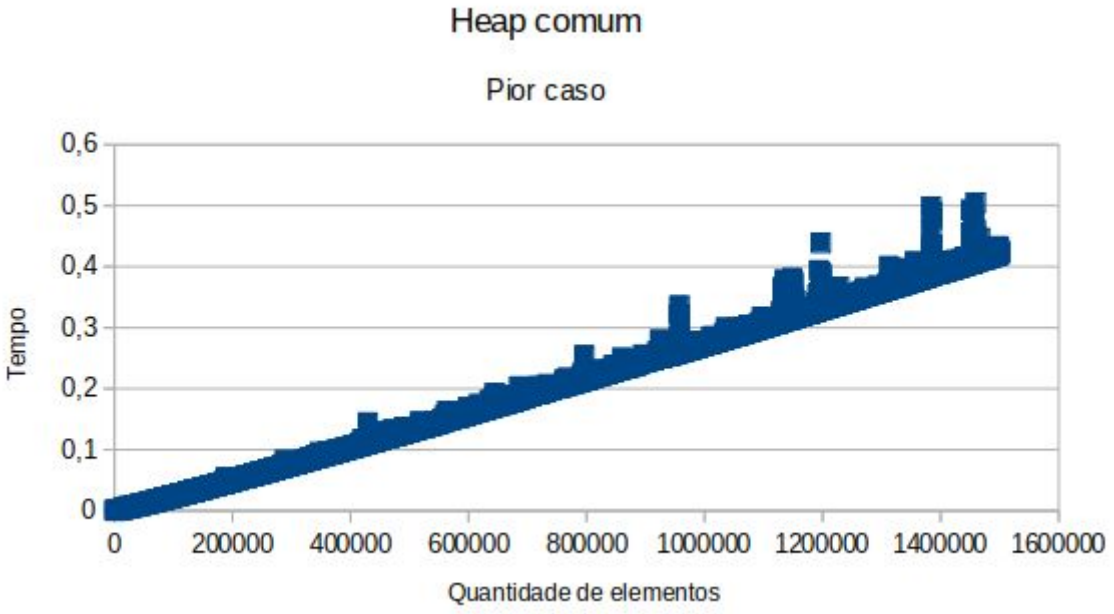

Imagem 5 - Gráfico do comportamento de uma heap comum no pior caso.

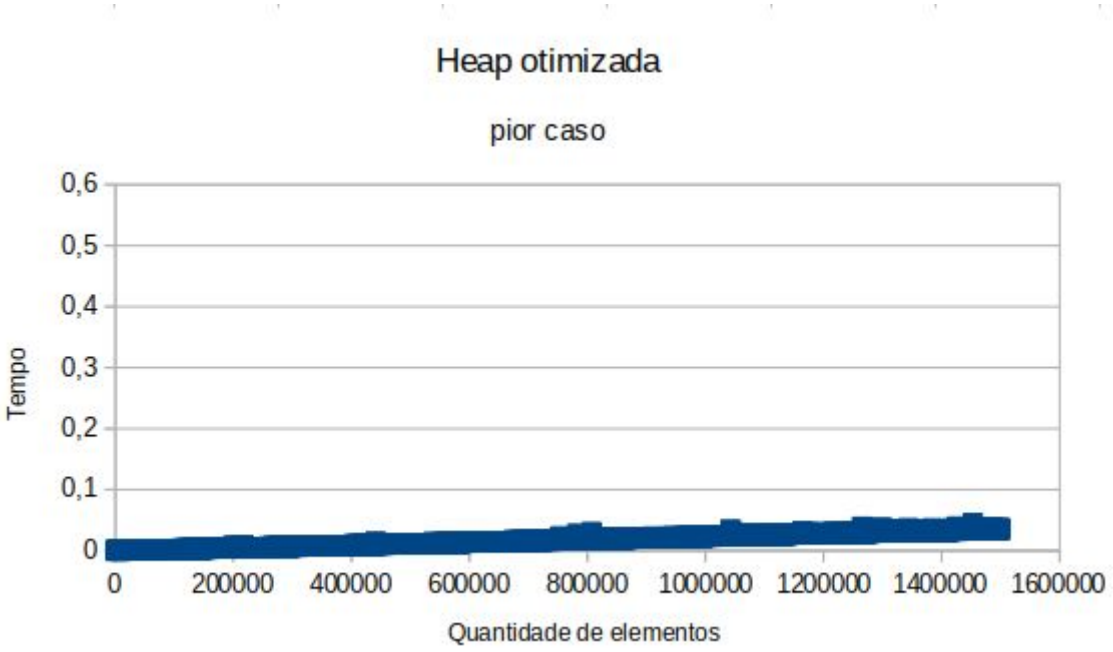

Imagem 6 - Gráfico do comportamento da heap projetada no pior caso.

Como observado nos gráficos acima, no pior caso a diferença é absurda entre as duas abordagens, o crescimento da função que dita o comportamento do algoritmo da heap otimizada é muito menor do que a da heap comum.

Conclusão

Dessa forma, conclui-se que é possível projetar uma heap com complexidade de tempo linear utilizando uma abordagem diferente da usualmente usada, de forma que os elementos sejam inseridos primeiramente e após isso ocorra a aplicação de um algoritmo corrige descendo em cada nó da heap.

Não se pode afirmar cegamente que essa abordagem pode ser utilizada amplamente em todos os casos que uma estrutura de dados como a heap é utilizada, pois em alguns casos em que a inserção na heap tem que ser necessariamente dinâmica no algoritmo essa abordagem proposta pode não ser útil. Entretanto, em casos que não é necessário isso, provavelmente essa implementação da heap irá tornar o algoritmo muito mais rápido e eficiente.

Referências

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/heap.html
<Acessado última vez dia 15/05/2019, 21:42>
https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity <Acessado última vez dia 15/05/2019, 15:35>
https://www.youtube.com/watch?v=MiyLo8adrWw
<Acessado última vez dia 15/05/2019, 18:22>

Apêndice

Apêndice A - Código da construção do heap e do código utilizado para testes

https://github.com/Sidovsk/BuildHeapInLinear