DOCUMENTAÇÃO TP1 – AEDS II

ALUNO: Bruno Maletta Monteiro
MATRÍCULA: 2017015150
DATA: 05/11/2017

1. Introdução

O projeto consistiu no desenvolvimento de um Tipo Abstrato de Dados (TAD), **TLista**, que contivesse uma lista unicamente encadeada, em que cada célula possui também uma sublista. A aplicação para o TAD foi apresentada como comentários em postagens do Facebook, de forma que cada comentário possa ter seus subcomentários, indefinidamente.

A entrada deveria ser lida de um arquivo de entrada, contendo as operações **adicionar comentário**, **responder a um comentário** e **remover comentario**, conforme codificado. A saída é dada em um arquivo de saída, contendo, para cada caso de teste, a representação dos comentários da forma como foram adicionados, e, em seguida, após ordenados.

O projeto foi de fácil implementação, entretando, houveram alguns pontos que causaram uma maior dificuldade, como a ordenação da lista encadeada, que foi feita a partir do algoritmo *bubble sort* e o liberamento de memória, de forma que não houvesse *memory leak* (vazamento de memória). Entretando, de forma geral, os problemas foram contornados com relativa facilidade e o resultado foi um programa sucinto e de fácil entendimento.

Vale ressaltar que, na ordenação, foi usado o algoritmo *bubble sort* porque ele apenas percorre a lista em uma direção, o que é bom, pois se trata de uma lista unicamente encadeada. Algoritmos que são *O(n log(n))*, como o *quick sort*, o *merge sort* e o *heap sort* seriam de muito difícil implementação, e, dentre os algoritmos quadráticos, o *bubble sort* pareceu ser o melhor para o presente problema.

2. Desenvolvimento

a) Operação **adicionar comentário**: *O(1)*

Para a implementação dessa operação, foi criada uma função *add*, que recebe um ponteiro para a lista e o *id* do comentário a ser adicionado. Fazendo uso do campo **tail** da lista, que aponta para a última célula da mesma, eu então adiciono uma célula com respostas vazia e *id* igual ao *id* que foi

passado para a função depois da célula **tail**, e atualizo o **tail** para apontar para a célula que foi agora adicionada.

Como eu faço uso de um ponteiro que aponta para a última posição da lista, eu não preciso percorrê-la. Como a única função chamada por essa operação é a função *new_list*, que é *O(1)*, o custo da operação **adicionar comentário** é *O(1)*.

b) Operação **responder a um comentário**: *O(n)*

Para a implementação dessa operação, fiz uso de uma função *answ*, que recebe um ponteiro para a lista, o *id1*, que é o comentário a ser respondido, e o *id2*, que é o comentário a ser adicionado.

Assim, eu itero pela lista, e para cada elemento da lista eu testo se ele é o comentário que queremos responder. Se sim, eu adiciono o comentário com *id2*, utilizando a função *add*. Senão, eu chamo a função *answ* para as respostas do comentário atual. Dessa forma, é feita uma busca por profundidade (DFS) nos comentários. Como adicionar é *O(1)* e, no pior caso, cada comentário é visitado uma única vez, temos que, para **n** comentários totais, a operação **responder a um comentário** é *O(n)*.

c) Operação **remover comentário**: *O(n)*

Para a operação em questão, é feita uma busca semelhante à da operação **responder a um comentario**, uma busca por profundidade. Entretando, quando o comentário é encontrado, em vez de chamar a função *add*, a função *clear* é chamada. Esta função recebe uma lista (no caso, as respostas do comentário a ser removido) e remove todos os comentários dessa lista recursivamente, e, também, a própria lista. Essa função é linear no pior caso, pois nunca visita o mesmo comentário mais de uma vez.

Assim, quando o comentário é encontrado, suas respostas (e subrespostas) são removidas usando a função *clear*, o comentário em questão é removido, e o comentário que apontava pra ele passa a apontar para o comentário seguinte.

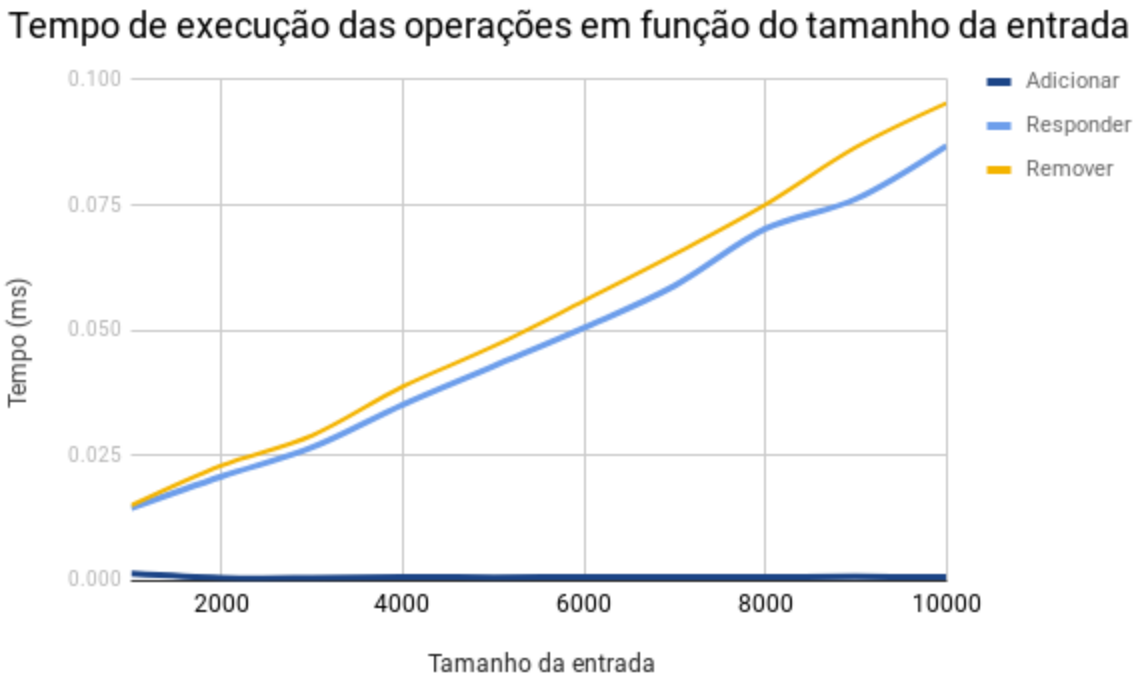
Como no pior caso a busca é linear, e a função *clear* é linear, a operação **remover comentário** também é linear, ou seja, é *O(n)*, para **n** comentários totais.

3. Análise experimental

Para realizar a análise experimental, criei um programa em *Python* que cria uma entrada com números variados de operações **adicionar comentário**, de forma que a lista fique com um certo tamanho definido. Em seguida, uma das operações é realizada, com enfoque na última célula no caso das operações **responder a um comentário** e **remover comentário**. Seu tempo de execução é calculado e é feito uma média de 100 execuções. Os resultados podem ser observados na seguinte tabela :

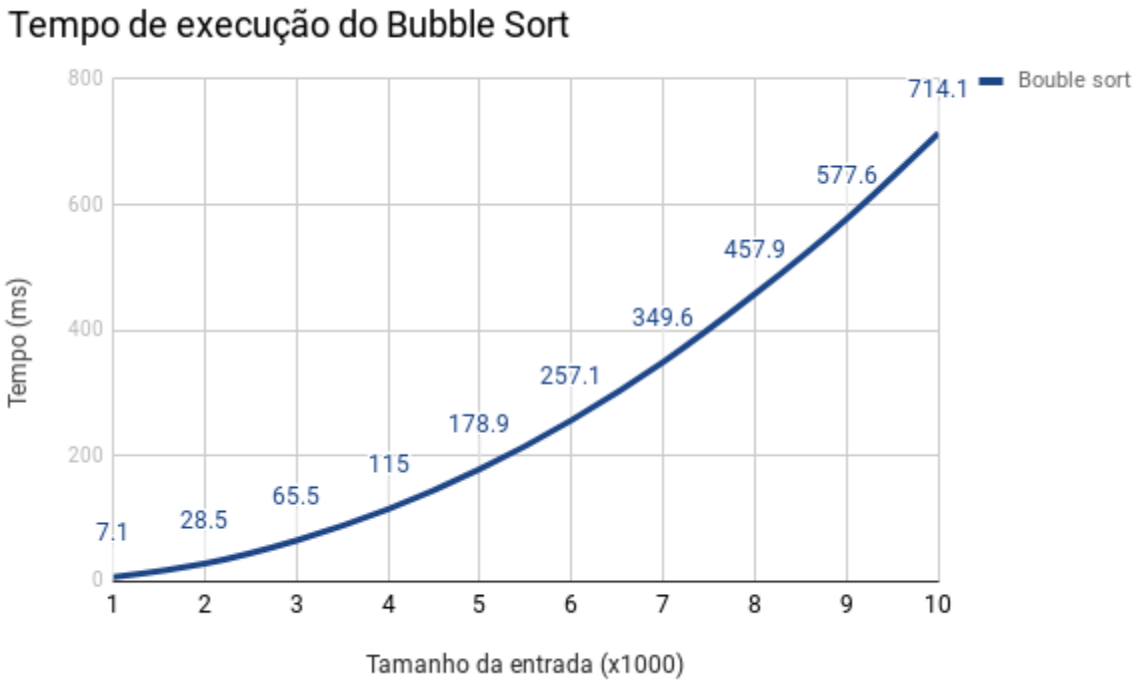
Tempo, em milisegundos, para realizar cada uma das operações (média de 100)

Tamanho	Adicionar	Responder	Remover
1000	0.001430	0.014510	0.015050
2000	0.000590	0.020830	0.023040
3000	0.000560	0.026650	0.029030
4000	0.000760	0.035200	0.038790
5000	0.000710	0.042850	0.046820
6000	0.000760	0.050520	0.055900
7000	0.000760	0.058930	0.065160
8000	0.000780	0.070260	0.075030
9000	0.000840	0.076220	0.086500
10000	0.000780	0.086840	0.095730



Observa-se que, de fato, as operações **responder a um comentário** e **remover comentário** exibiram um comportamento linear, como era o de se esperar, por serem *O(n)*. Além disso, a operação **adicionar comentário** se mostrou ser de custo constante, *O(1)*.

Podemos também observar o funcionamento quadrático do *bubble sort* no seu pior caso:



4. Conclusão

Conclui-se que a estrutura de dados utilizada é eficiente, pois as operações tem complexidade baixa e fácil implementação. Acredito que seja uma boa opção para a reslução do problema.

Entretanto, a ordenação com lista encadeada pode ser um pouco complicada. Se os tamanhos são pequenos, um algoritmo quadrático, como o que foi usado, já basta para ordenar em tempo aceitável. Mas, caso a entrada seja muito grande, pode ser necessário implementar algoritmos de ordenação mais complexos, o que seria bastante trabalhoso em uma lista encadeada.

Além disso, para cada uma das operações, assim como para a ordenação, o comportamento experimental observado foi condizente com as previsões (complexidades) teóricas dos algoritmos.