Relatório - REO 4

Arthur Henrique Sousa Cruz

Universidade Federal de Lavras

1 Execução

O arquivo **main.py** foi criado para execução de testes.

Um arquivo pode ser passado por linha de comando, permitindo que os resultados obtidos sejam salvos no arquivo. Os resultados serão exibidos na tela do terminal caso seja passado o parâmetro ou não. Para executar o código utilize o seguinte comando:

python3 main.py (arquivo-de-saida)

2 Organização de arquivos

Os arquivos deste projeto foram organizados da seguinte forma:

- **algoritmos_ordenacao**: Arquivo que implementa as funções de ordenação utilizadas na geração de instâncias de teste;
- **arvore_binaria_busca.py**: Contém a implementação da Árvore Binária de Busca (ABB)
- **avl.py**: Contém a implementação da AVL.
- **main.cpp**: Arquivo que gera e executa testes;
- **teste_envio.txt**: Arquivo com os resultados dos experimentos realizados;

3 Implementação

Os algoritmos e estruturas foram implementados em *Python* e os testes são gerados aleatoriamente com a biblioteca *random*. A implementação do algoritmo quicksort (feita na semana passada) foi utilizada para a ordenação do vetor de testes.

3.1 Árvore Bináriade Busca (ABB)

Para a implementação da ABB foram criadas duas classes: *NohBusca* e *ArvoreBinariaBusca*. Ambas as implementações estão no arquivo **arvore_binaria_busca.py**. Uma instância da classe *NohBusca* possui os seguintes atributos:

- **indice**: item armazenado.
- **esquerda**: ponteiro para o filho à esquerda.

- **direita**: ponteiro para o filho à direita.
- **pai**: ponteiro para o pai.

Além destes atributos, um *NohBusca* também possui os seguintes métodos:

- **eh_maior(noh)**: retorna verdadeiro se o item que está armazenando é maior se comparado ao do noh passado por parâmetro.
- **__init__(indice)**: construtor da classe.
- **__str__(indice)**: sobrecarga do operador de escrita da classe.

Por sua vez, uma *ArvoreBinariaBusca* possui apenas a **raiz** da árvore como atributo e os seguintes métodos:

- **adiciona(inidice)**: adiciona um novo elemento na árvore. O método percorre a árvore comparando os nós da árvore ao novo nó até alcançar uma folha. Após isso a novo elemento é inserido como filho da folha encontrada.
- **busca(item)**: percorre a árvore comparando os nós da árvore ao novo não até que o encontre ou que alcance uma folha. O método retorna se o item foi encontrado e quantos passos foram gastos.
- **get_texto_itens_em_ordem()**: percorre a árvore e gera uma *string* com os elementos da árvore em ordem crescente.
- **get_texto_itens_descendo()**: percorre a árvore e gera uma *string* partindo da raiz. Utilizado para realizar verificações durante o desenvolvimento.

Além desses, a classe possui um construtor que inicializa a raiz e métodos com o final **_recursivo** no nome. Esses métodos são auxiliares para os listados acima.

3.2 AVL

Para a implementação da AVL foram necessárias alterações na ABB. O arquivo **avl.py** contém duas estruturas, *NohAVL* e *AVL*. A primeira contém os mesmos elementos da classe *NohBusca* (Sessão 3.1) acrescida do atributo **altura**.

A classe *AVL*, contudo, sofreu mais modificações. Os métodos alterados são listados abaixo:

- **adiciona(indice)**: a função de inserção foi dividida em duas, sendo uma delas auxiliar. Esse novo método percorre a árvore até encontrar uma folha, insere o novo elemento e depois balanceia a árvore chamando o método **balanceia**.
- **balanceia(noh)**: esse novo método verifica se a AVL está balanceada e, caso não esteja, aplica rotações para cada um dos possíveis casos de desbalanceamento.
- **rotaciona_direita**: executa uma rotação para a direita centrada em um nó específico. Ao fim da rotação o nó passa a ser filho do vértice à sua esquerda e seu filho à esquerda passa a ser o antigo filho à direita de seu novo pai.

- **rotaciona_esquerda**: executa uma rotação para a esquerda centrada em um nó específico. Ao fim da rotação o nó passa a ser filho do vértice à sua direita e seu filho à direita passa a ser o antigo filho à esquerda de seu novo pai.
- **calcula_balanceamento(noh)**: calcula o balanceamento de um nó utilizando a altura de subárvores de seus filhos.

4 Comparação dos algoritmos

O arquivo com os resultados dos testes realizados é o arquivo **teste_envio.txt**. Se o programa for reexecutado e receber como parâmetro de entrada um arquivo de mesmo nome ele será sobrescrito.

Utilizando a ABB foi possível encontrar o último elemento do conjunto *US* em 14 passos. Para o último elemento do conjunto *OS* foram necessários 100 passos. Esses valores refletem a variação de complexidade que uma árvore de busca binária sofre dependendo de seu balanceamento. Uma árvore completamente balanceada teria uma complexidade de $O(\log n)$, visto que a cada nível da árvore são armazenados o dobro de elementos do nível anterior. Para alcançar o elemento mais distante da raiz, seria necessário percorrer um vértice por nível na árvore cujo tamanho é $\log n$. O pior caso, porém, seria a inserção de um vetor ordenado. Nesse caso a árvore se comportaria da mesma forma que uma lista encadeada e a complexidade de busca em uma lista é de $O(n)$.

Por outro lado, a AVL permanece sempre balanceada. Isso se reflete na quantidade de passos necessário para realizar a busca: tanto para o conjunto *US* quanto para o *OS* foram necessários 7 passos. A complexidade de tempo para a busca se mantém $O(\log n)$ independente da entrada, já que a árvore estará sempre balanceada. O custo para manter o balanceamento é que a inserção de novos elementos na AVL tem um pequeno aumento: é necessário a atualização da árvore, que tem complexidade de $\log n$ (no pior há uma rotação com complexidade $O(1)$ por nível). Esse aumento, contudo, não aumenta a complexidade de tempo, já que tem-se o tempo de inserção $O(\log n)$ somado ao tempo de atualização $O(\log n)$. Isso mantém o comportamento assintótico de $\log n$ no pior caso.