Algoritmos e Estruturas de Dados I: 2° trabalho

Murilo Leandro Garcia
Glauco Fleury Corrêa de Moraes

1 Introdução

Neste projeto foi implementado um TAD de conjuntos na linguagem C. Seguindo as especificações do projeto, foram criadas as funções básicas de uma ED: criação, deleção, inserção de elementos, remoção de elementos, e impressão. Além disso, as seguintes funções especificas de conjuntos: união e intersecção. Foram utilizadas duas estruturas de dados para armazenamento dos elementos do conjunto: Árvore AVL e Left-Leaning Red Black Tree.

2 Estruturas de Dados implementadas

De todas as estruturas de dados ensinadas na disciplina, a escolha clara para este projeto fora utilizar árvores, especificamente as duas ensinadas em aula: Left Leaning Red Black Tree e AVL. Comparando as complexidades de suas operações básicas com filas, listas, pilhas e deques, fica evidente o fato de serem a melhor escolha: buscar, inserir e remover são efetuados com complexidade $O(\log n)$, enquanto todos os demais apresentam pelo menos uma destas com complexidade $O(n)$, tornando-as menos eficientes (ex: inserção em lista ordenada).

3 Conjunto: operações principais

3.1 Pertence

A operação de 'pertence' requer uma explicação sobre a busca em árvores. Como ambas as estruturas de dados são baseadas em árvores binárias de busca (ABB), a busca por um elemento terá complexidade $O(\log n)$, onde n é o tamanho do conjunto sendo inserido. Isso porque a cada nó visitado, ao escolher ir para a esquerda ou para a direita, o número de nós que poderia ser inserido, em um caso ideal, é cortado pela metade, ou seja, se visitam $\approx \log_2 n$ (busca binária).

Todavia, isso é apenas no caso ideal. É possível que no processo de busca em uma ABB a árvore fique degenerada e se torne algo próximo de uma linked list (no pior caso, igual uma linked list). Por isso, se usaram duas árvores que se balanceiam automaticamente, a AVL e a LLRBT. O processo de balanceamento aumenta ligeiramente a complexidade do algoritmo, mas torna em geral o processo mais rápido. Considere b como constante simbólica do peso das comparações.

Na árvore AVL, a altura h de uma árvore com n nós satisfaz:

$$\log_2(n+1) \leq h < \log_\phi(n+2) - \frac{\log_2 5}{2\log_2 \phi} + 2$$

Isso ocorre por conta do rebalanceamento que é feito após cada operação de inserção, que garante que a árvore não se torne degenerada ou muito desbalanceada. Dessa forma, é garantido no mínimo $(1,44\log_2 n)$ de etapas para chegar até o nó que está sendo procurado. No pior dos casos, você terá que andar todos os $(1,44\log_2 n)$ até achar o elemento, resultando em pior caso com complexidade igual a $O(1,44b\log_2 n) = O(\log n)$.

Já na árvore Left Leaning Red Black Tree (LLRBT), a lógica é a mesma, porém com pior caso da altura sendo $2\log_2 n$. Assim, no pior caso, $O(2b\log_2 n) = O(\log n)$ para achar o nó que contém o elemento.

3.2 Inserção

A inserção de elementos requer atravessar a árvore tal qual se estivesse buscando o elemento de inserção nela, de tal forma que, quando um ponteiro nulo for encontrado no local onde ele deveria estar, deve-se inseri-lo. Como inserir e balancear ocorrem com complexidade constante, é possível afirmar que a complexidade dessa operação é idêntica à de 'pertence'. Porém, são necessárias algumas explicações mais detalhadas a respeito disso.

Após a travessia da árvore e inserção do elemento, que ocorrem respectivamente em complexidade $O(\log n)$ e $O(1)$, é necessário em ambas as árvores voltar (via as chamadas recursivas) da folha à raiz rebalanceando os nós se necessário, sendo que cada operação de balanceamento também ocorre em complexidade constante. Considerando isso e assinalando uma constante k e j para essas operações em AVL e LLRBT, para ambas, teríamos complexidades algorítmicas de, respectivamente, $O(1,44k\log_2 n + 1,44b\log_2)$ e $O(2j\log_2 n + 2b\log_2 n)$ no pior caso, isto é, quando é necessário percorrer a altura inteira da árvore para inserir o elemento. Simplificando os big-Oh's, portanto, a inserção é de complexidade final $O(\log n)$.

3.3 Remoção

Remover um elemento de um conjunto implica percorrer a árvore procurando por ele ($O(\log n)$), removê-lo, e ajustar a árvore de modo a manter suas propriedades de balanceamento e de ABB. Em ambas as estruturas de dados, a remoção tem 3 casos: o nó é folha, tem 1 filho apenas (para ambos, o ajuste necessário é $O(1)$), ou tem 2 filhos; aqui, percorre-se a sub-árvore esquerda em busca do maior elemento, apagando-o após trocar sua chave com a do nó cujo elemento se deseja retirar (para qualquer situação desse tipo, complexidade do pior caso de remoção continua sendo $O(\log n)$).

Analisando mais especificamente: na AVL, remover um item requer percorrer a árvore na ida, removê-lo, e na volta balanceá-la com rotações, o que implica $O(1,44k\log_2 n + 1,44b\log_2)$ (tal qual na inserção); já na LLRBT, o mesmo pode ser dito, porém deve-se também considerar as operações extras a mais na ida recursiva que propagam a aresta vermelha até o nó que será removido (dado que somente aqueles com incidência vermelha podem ser eliminados), com um número q de operações envolvidas nisso. Logo, detalhadamente, remover na Rubro-Negra tem complexidade $O(2j\log_2 n + 2bq\log_2 n)$. Logo, remover para ambos os casos acontece em $O(\log n)$, simplificando.

3.4 União

O algoritmo tanto para o caso da AVL quanto para o caso da LLRBT baseia-se em: Criar um conjunto C e inserir todos os elementos dos conjuntos de entrada (A e B). Para inserir todos elementos de um conjunto, a árvore é percorrida em-ordem e para cada nó visitado, é inserido sua chave na árvore do conjunto C. Se o elemento já estiver contido na árvore, na hora da inserção, ela buscará a posição em que ele deveria estar no novo conjunto, mas sem efetivamente mudar nada, o que impede que haja elementos repetidos.

Assim, chamando $n = |A|$ e $m = |B|$, percorrer-se-ão todos os $n + m$ nós, e para cada um dos nós, será feita uma inserção de complexidade $O(\log(n+m))$. A complexidade total do algoritmo será portanto $O((n+m)\log(n+m))$. Especificamente, para a AVL e LLRBT teremos, considerando constantes com significado já préviamente estabelecido, respectivamente, $O(1,44(n+m)\cdot k\log_2(n+m))$ e $O(2(n+m)\cdot j\log_2(n+m))$.

3.5 Intersecção

O algoritmo de intersecção também tem a mesma estrutura para ambos as estruturas de dados. É criado um conjunto C, e percorre-se o conjunto de entrada A (da mesma forma que a anterior, usando recursão em-ordem). Para cada elemento de A, é checado se ele também está contido em B. Caso seja, ele é inserido em C. No pior caso, os dois conjuntos são iguais, e é necessário percorrer um conjunto inteiro enquanto se checa se o elemento atual está presente e inserindo-o no novo conjunto criado, resultando em $O(2n\log n) = O(n\log n)$. Para AVL e LLRBT teremos, respectivamente, $O(1,44nk\log_2 n)$ e $O(2nj\log_2 n)$.

4 Outras operações do conjunto

As demais operações, por serem mais simples, serão explicadas nessa única seção.

Criar Conjunto: requer basicamente a criação de uma struct para ambas as árvores, o que é feito em $O(1)$.

Apagar Conjunto: percorre-se cada nó da árvore recursivamente, apagando seu conteúdo e liberando o espaço alocado para ele na memória. Como desalocar e apagar envolve um número fixo de operações k, e percorre-se a árvore inteira, teremos que: $O(kn) = O(n)$.

Imprimir: envolve o percurso em-ordem das árvores, imprimindo os elementos presentes; como executa-se uma ação (printar) um número n de vezes: $O(n)$.