TRABALHO 2 - ALGORITMO E ESTRUTURA DE DADOS

Relatório de Análise de Complexidade de Operações em Árvore AVL - Pokédex

Frederico Nizoli e Mateus Tieppo

---

1. Introdução

O objetivo deste relatório é analisar as operações realizadas em uma estrutura de dados baseada em uma árvore AVL (Árvore Binária Balanceada), que oferece balanceamento automático após cada inserção ou remoção. As operações em questão são inserção, remoção, busca por ID, listagem por pontos de combate e busca por tipo e nível. O foco principal será a análise da complexidade de tempo dessas operações, considerando a estrutura de dados utilizada e os algoritmos empregados.

2. Operações em Árvore AVL

2.1. Inserção (insert)

A operação de inserção em uma árvore AVL segue um processo recursivo. Inicialmente, realiza-se a busca pela posição correta para o novo nó, o que é feito de forma semelhante à inserção em uma árvore binária de busca. A principal diferença em uma árvore AVL é que, após a inserção, o balanceamento da árvore deve ser mantido, o que pode envolver rotações à esquerda ou à direita.

- **Custo de busca para inserção:** A busca por um local adequado na árvore para inserir um novo nó é realizada de forma eficiente, com complexidade **O(log n)**, pois a altura da árvore é logarítmica em relação ao número de elementos, dada a propriedade de balanceamento da árvore AVL.
- **Custo de balanceamento:** Após a inserção do nó, a árvore pode precisar ser reequilibrada. O processo de balanceamento pode envolver até duas rotações (uma à esquerda e uma à direita), mas cada rotação tem complexidade **O(1)**, uma vez que consiste em trocas locais de ponteiros entre os nós. O custo de verificação de balanceamento após a inserção também é **O(log n)**, devido à necessidade de percorrer a árvore para atualizar as alturas dos nós.

- **Inserção no índice de tipos:** A inserção no índice de tipos, representado por um mapa hash (HashMap), é realizada em **O(1)** em média, já que o acesso ao mapa é feito diretamente pela chave (o tipo do Pokémon).

Conclusão da Inserção:
A complexidade total da operação de inserção é **O(log n)**, onde n é o número de nós na árvore. Isso garante que a inserção ocorra de forma eficiente, mantendo a árvore balanceada.

2.2. Remoção (remove)

A remoção de um nó em uma árvore AVL segue um processo similar ao da inserção, mas com algumas diferenças. Após a remoção do nó, a árvore pode precisar ser reequilibrada para manter as propriedades de balanceamento.

- **Custo de busca para remoção:** A busca pelo nó a ser removido tem complexidade **O(log n)**, pois é necessário localizar o nó de forma eficiente utilizando a propriedade de árvore binária de busca.
- **Custo de balanceamento pós-removal:** Após a remoção, a árvore pode precisar ser reequilibrada. Isso é feito verificando as alturas dos nós e, se necessário, realizando rotações à esquerda ou à direita. O custo das rotações é **O(1)**, mas a verificação e atualização das alturas dos nós têm complexidade **O(log n)**, pois pode ser necessário percorrer até a raiz da árvore.

Conclusão da Remoção:
A operação de remoção, como a inserção, tem complexidade **O(log n)**, devido ao balanceamento automático da árvore AVL.

---

3. Busca por ID (findById)

A busca por ID é realizada em uma árvore AVL com o objetivo de encontrar o Pokémon correspondente ao ID fornecido.

- **Custo da busca:** Como a árvore é balanceada, a busca por um elemento segue a mesma lógica de busca em uma árvore binária de busca, com a diferença de que, no caso de árvores balanceadas, a altura da árvore é limitada a **O(log n)**. Isso garante que a busca seja eficiente mesmo para grandes conjuntos de dados.

Conclusão da Busca por ID:
A complexidade da busca por ID é **O(log n)**, uma vez que a árvore AVL mantém a altura balanceada.

4. Listagem de Pokémons por Pontos de Combate (listPokemonsByCombatPoints)

Esta funcionalidade realiza a travessia da árvore em ordem (in-order) para coletar todos os Pokémons e os insere em uma lista ordenada por pontos de combate. Essa operação envolve dois passos principais:

1. **Travessia da Árvore em Ordem (in-order traversal):**
   A travessia em ordem visita cada nó da árvore uma vez. Como cada nó (Pokémon) é visitado uma única vez, a complexidade dessa operação é **O(n)**, onde n é o número de nós na árvore.
2. **Ordenação dos Pokémons:**
   Após coletar os Pokémons, a lista de Pokémons precisa ser ordenada com base nos pontos de combate. A ordenação é realizada utilizando o método **sort()**, que possui complexidade **O(n log n)**, onde n é o número de elementos na lista de Pokémons.

Cálculo do Custo Total:
A complexidade total dessa operação é a soma do custo da travessia (**O(n)**) e o custo da ordenação (**O(n log n)**). Logo, o custo total da funcionalidade **listPokemonsByCombatPoints** é **O(n log n)**.

Conclusão da Listagem por Pontos de Combate:
A complexidade total da funcionalidade de listagem por pontos de combate é **O(n log n)**, devido à necessidade de ordenação da lista após a travessia da árvore.

---

5. Busca por Tipo e Nível (listPokemonsByTypeAndLevel)

A busca por tipo e nível utiliza um índice de tipos, implementado como um **HashMap<String, Set<Pokemon>>**, para acessar diretamente o conjunto de Pokémons de um tipo específico. Após encontrar o conjunto, a operação filtra os Pokémons de acordo com o nível mínimo especificado.

1. **Busca no HashMap:**
   A busca por um conjunto de Pokémons de um tipo específico é realizada em **O(1)** em média, já que o acesso ao HashMap é direto, utilizando a chave que representa o tipo do Pokémon.
2. **Filtragem por Nível:**
   Após localizar o conjunto de Pokémons do tipo, a função realiza a filtragem dos Pokémons de acordo com o nível mínimo. A iteração sobre os Pokémons tem complexidade **O(m)**, onde m é o número de Pokémons do tipo. No pior caso, onde todos os Pokémons são do mesmo tipo, m pode ser igual a n, resultando em uma complexidade **O(n)**.

Conclusão da Busca por Tipo e Nível:
A complexidade média da operação de busca por tipo e nível é **O(m)**, onde m é o número de Pokémons do tipo, o que geralmente será menor que n. No pior caso, a complexidade é **O(n)**, quando todos os Pokémons pertencem ao mesmo tipo.

6. Resumo das Complexidades de Tempo

Abaixo estão as complexidades de tempo das principais operações realizadas na árvore AVL:

- **Inserção: O(log n)**
- **Remoção: O(log n)**
- **Busca por ID: O(log n)**
- **Listagem por Pontos de Combate (listPokemonsByCombatPoints): O(n log n)**
- **Busca por Tipo e Nível (listPokemonsByTypeAndLevel): O(m) (m < n em média)**

---

7. Conclusão Final

As operações fundamentais de inserção, remoção e busca em uma árvore AVL são eficientes, com complexidade **O(log n)**, o que garante que o sistema seja capaz de lidar com grandes volumes de dados de forma eficaz. A funcionalidade de listagem por pontos de combate, que envolve uma ordenação adicional, tem complexidade **O(n log n)**. A busca por tipo e nível é eficiente, com complexidade média **O(m)**, onde m tende a ser menor que n, proporcionando um bom desempenho na maioria dos casos.

Em resumo, a utilização de uma árvore AVL como estrutura de dados para armazenar e gerenciar Pokémons oferece operações rápidas e eficientes, atendendo aos requisitos de performance para diferentes tipos de consultas e operações.