Código

Na Main, são definidas arrays e listas para inserir aleatoriamente os valores nas árvores. Criei esses métodos para que facilmente eu pudesse manipular os testes, bem como a quantidade de amostragem e quantidade de valores a serem adicionados. Também são criadas listas para guardar os valores em ms de cada etapa em cada árvore, assim posso calcular a média de tempo para cada método. O que acontece na main é a chamada em loop dos métodos em cada classe e armazenamento do tempo de execução delas, bem como a impressão dos dados performáticos.

No arquivo da classe da árvore binária existe a classe Node com o construtor de nós de esquerda e direita, pegando como parâmetro a chave que nada mais é do que o valor que o nó armazena. Na classe da árvore propriamente dita, é criado o nó raiz, que vai servir de base para a criação da árvore.

Na chamada do método inserir, o valor adicionado vai ser comparado com a raiz e como esse valor inicialmente é nulo, ele vai assumir o valor de raiz. Após isso, cada valor adicionado será comparado recursivamente para ser adicionado à árvore, se for menor, vai para esquerda, se for maior, vai para a direita, caso esteja sendo comparado com nulo, ele se torna uma nova raiz e é adicionado aquela posição.

No método inserir da árvore AVL, existe um passo a mais. Como ela deve ser balanceada, sempre é comparado o coeficiente de balanceamento, que nada mais é do que a diferença entre a altura de nós da direita pela altura de nós à esquerda. Ela deve ser balanceada sempre que essa altura for maior que 1 ou menor que -1. Para balanceamento, é utilizado o método chamado de rotação, que realoca a raiz para o lado desbalanceado. Isso ocorre pelos métodos rotateLeft e rotateRight que definem um novo valor de posição de altura para o nó raiz desbalanceado. Eles são chamados recursivamente até a estrutura ficar balanceada.

A buscar é mais simples e funciona de maneira igual para as duas árvores. Começando da raiz da árvore, cada valor é comparado com o valor da busca, caso seja maior, é procurado à direita, caso menor é procurado à esquerda, caso igual, o node é retornado como achado (true).

Os métodos de Delete são basicamente os métodos de busca e de inserção só que em vez de descer, todos os nós inferiores são realocados verticalmente para ocupar o lugar da raiz. No caso da árvore AVL, é realizado novamente o balanceamento e realocação de raiz.

Performance

Para análises de performance utilizei a métrica de tempo de execução de bloco de código do IntelliJ.

Também vale notar que utilizei amostragens de 10 execuções em loop com valores randômicos para poder tirar médias de tempo de execução, imprimindo também os valores de cada processo isoladamente para ter como parâmetro isolados.

Nos testes de 100, 500, 1000, 10000 e 2000, foi constatado que a árvore AVL se mostrou mais performática na Busca somente, o que era esperado pelo seu método de balanceamento. Em execuções com o escopo menor que 20000 itens, os valores variaram muito e não se mantiveram constantes.

Outra coisa que era esperada também, era que o tempo de inserção da árvore AVL fosse maior que o da árvore binária pelo fato de sempre balancear a cada inserção.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Adicionei para fins de teste mais três dezenas de milhares crescentes após o 20000. Os resultados se provaram iguais, porém, com uma margem muito menor entre os tempos de busca das duas árvores.

Texto

Descrição gerada automaticamente

O curioso foi quando utilizei valores muito grandes. No caso abaixo, adicionei 1000000 à lista de números de inserções. Eu esperava que fosse ser muito mais rápido para a árvore AVL realizar a busca de valores pelo seu balanceamento, porém, foi muito mais ineficiente em todos os aspectos. Texto

Descrição gerada automaticamente

A conclusão que eu cheguei, foi que para valores de baixo e médio porte, a árvore AVL é superior para casos de busca. Para os outros casos, por não possúir nenhuma métrica de balanceamento, a árvore binária se mostrou muito mais eficiente. Contraintuitivamente, para valores muito grandes, a árvore AVL se mostrou menos eficiente em todos os casos. Chequei minha implementação mas não consegui achar nada que interferisse no tempo de busca, até porque o método era o mesmo para ambas as árvores. Não consegui utilizar outras métricas pela limitação do meu sistema operacional, JConsole, por exemplo, não está disponível para macOS, e todas as configurações do IntelliJ estavam para desempenho máximo do código.