UNIVERSIDADE PRESBITERIANA MACKENZIE

FACULDADE DE COMPUTAÇÃO E INFORMÁTICA

ESTRUTURA DE DADOS II – Prof. Jean Marcos Laine

Alan Meniuk Gleizer – RA

Caio Vinicius Corsini Fillho – RA

Gilberto De Melo Júnior – RA 10419275

Análise de Performance de BSTs e AVLs Utilizando Dados de Alunos Matriculados em Centros de Estudos de Línguas da Rede Escolar Estadual de São Paulo

São Paulo, SP

20 de novembro de 2024

1.Sumário

2. Introdução

O armazenamento e processamento de informações em computadores é feito por diferentes tipos de estruturas de dados. O estudo delas se torna essencial para se entender como um computador funciona até na mais superficial das camadas. Neste projeto, dois tipos de árvores, *Binary Search Tree (BST)* e Árvore AVL (AVL) são utilizadas para armazenamento e processamento de informações de um conjunto de dados obtido do site de dados aberto do Governo do Estado de São Paulo.

Existem dois objetivos principais neste projeto: o primeiro é o contraste e a comparação das performances da BST e AVL em operações de inserção, busca e remoção de dados. O segundo, é a utilização das árvores para fazer uma análise exploratória dos dados e ajudar a responder perguntas pertinentes a respeito e sua importância em relação ao Objetivo de Desenvolvimento Sustentável (ODS) “Educação de Qualidade” da Organização das Nações Unidas (ONU).

3. Escolha do Dataset

5. Modelagem dos Dados em Árvores

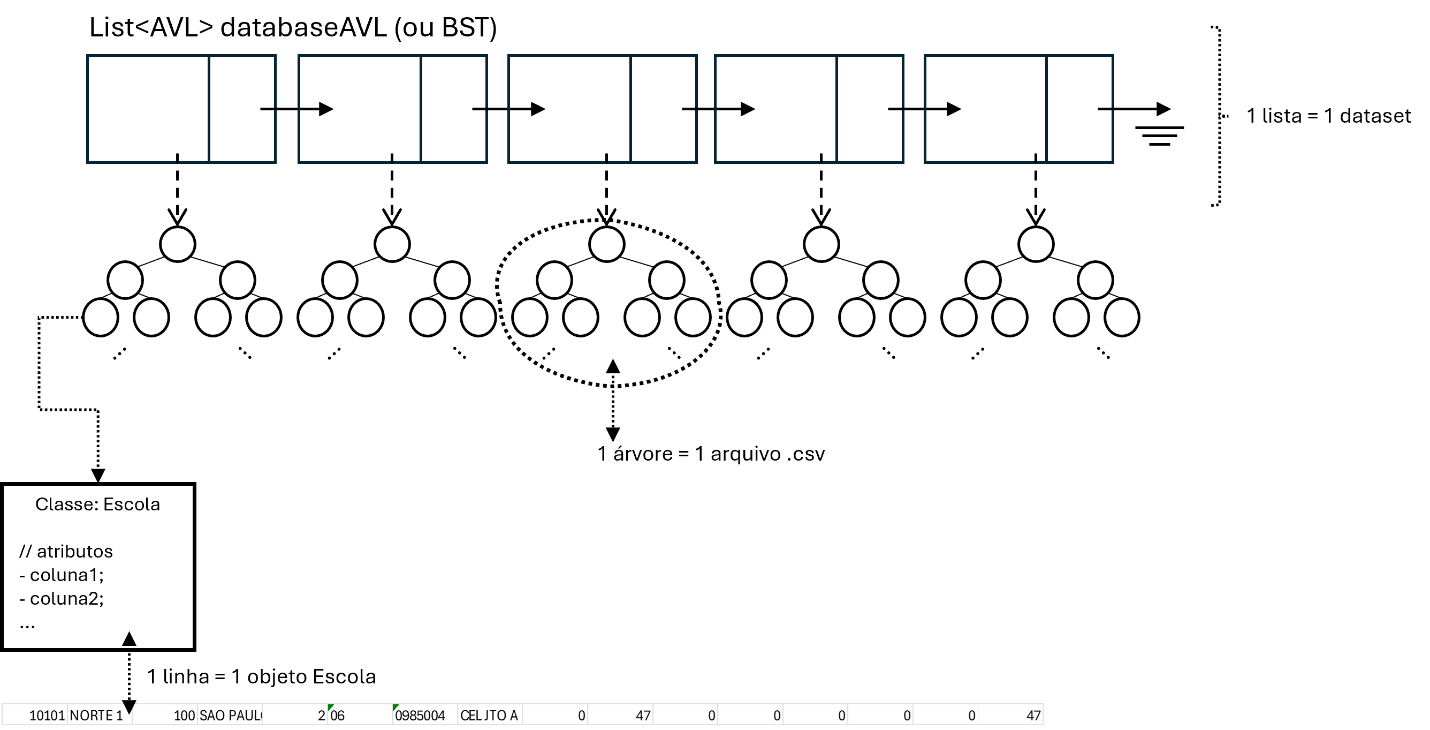
Neste projeto, o objetivo principal foi estruturar e organizar dados de instituições de ensino a partir de um conjunto de registros de escolas, de maneira que permitisse a busca e inserção eficientes. Para isso, utilizamos duas estruturas de dados baseadas em árvores: a árvore binária de busca (BST) e a árvore AVL, uma variação da BST que realiza auto-balanceamento. Na medida que o dataset selecionado armazena as informações segundo a lógica de uma escola por linha, e um arquivo .cvs por semestre, a organização dos dados nas árvores foi realizada de forma a espelhar a estrutura do dataset. Partindo de uma visão micro para macro, a organização do dataset se deu da seguinte forma:

A classe Escola é responsável por encapsular cada linha de dados, representando assim uma escola específica em um semestre específico. Cada coluna de dados de uma linha é representada por um atributo da Escola. Como o dataset possui um número grande de colunas, foi necessária uma pesquisa sobre a forma mais eficiente de lidar com todos esses atributos na codificação do projeto. Em alguns momentos, com na impressão dos atributos, a biblioteca de reflexão do Java foi usada para iterar sobre os atributos da classe Escola em tempo de execução, sem a necessidade de listá-los manualmente no código.

Cada instância da classe Escola é armazenada em um nó (classe Node) de uma árvore BST e outra árvore AVL, e serve como dado central de cada nó. O código da escola (int codEsc) foi escolhido como chave primária para ordenação, busca e balanceamento das estruturas. A escolha pelo codEsc foi bastante intuitiva, na medida que se trata do único código garantidamente único entre todas as linhas de cada .csv, e as árvores trabalhadas não permitem/preveem uso de chaves duplicadas. Também foi decidido armazenar cada codEsc como um inteiro, ao invés de string, dado que operações de comparação entre inteiros são mais rápidas que entre objetos complexos.

Dessa forma, cada .csv, e consequentemente cada semestre, é representado por uma coleção de Nodes em uma árvore AVL e BST. Assim, foi necessário escolher ainda mais uma estrutura de dado para representar os diversos semestres que compõe o dataset como um todo. Como o dataset é limitado a 10 arquivos CSV, optamos pelo uso de uma simples lista encadeada, na qual cada elemento corresponde a uma árvore.

A estrutura final do dataset dentro do programa pode ser bem representada pelo esquema abaixo:

Figura 1: representação da estrutura de organização do dataset

Vale ressaltar que a inserção dos dados nos objetos Escola, e a subsequente construção das árvores e listas de árvores foi realizada manualmente, sem uso de bibliotecas dedicadas à leitura de .csv. Para isso, e com o objetivo de respeitar os princípios da programação orientada a objetos, foram utilizadas duas classes distintas:

A classe CSVreader é responsável por ler e interpretar o conteúdo de cada arquivo CSV. Esse processo é dividido em várias etapas, das quais é interessante destacar a abertura de cada arquivo e criação de uma lista de strings, e a conversão da lista de strings em uma lista de objetos Escola, por meio da tokenização com base no “;”.

Cada .csv requer a criação de um objeto CSVreader, de forma que uma segunda classe DatabaseManager é responsável pelo gerenciamento dos vários CSVreader e criação das árvores e listas de árvores. Neste processo, cabe destacar abertura do diretório que contém os arquivos .csv e criação de uma lista de strings com os nomes dos arquivos, a criação de um CSVreader para cada .csv, e a subsequente criação de árvores AVL e BST a partir das listas de Escola criadas nos CSVreaders. A classe DatabaseManager possui, dessa forma, dois atributos distintos, uma lista de AVLs e uma lista de BSTs que são passados como cópia para chamadas de fora da classe, afim de proteger a integridade dos dados originais.

A estrutura final dos dados na classe DatabaseManager pode ser representada conforme o esquema abaixo:

DatabaseManager

└── AVLdatabase: List<AVL>

├── AVL Semestre 1

│ ├── Node (Escola 1)

│ │ ├── Escola: { codEsc: 1001, nome: "Escola A", ... }

│ ├── Node (Escola 2)

│ │ ├── Escola: { codEsc: 1010, nome: "Escola B", ... }

│ └── ...

├── AVL Semestre 2

│ ├── Node (Escola 3)

│ │ ├── Escola: { codEsc: 1023, nome: "Escola C", ... }

│ ├── Node (Escola 4)

│ │ ├── Escola: { codEsc: 1035, nome: "Escola D", ... }

│ └── ...

└── AVL Semestre N

├── Node (Escola N1)

│ ├── Escola: { codEsc: 2001, nome: "Escola X", ... }

├── Node (Escola N2)

│ ├── Escola: { codEsc: 2010, nome: "Escola Y", ... }

└── ...

└── BSTdatabase: List<BST>

Figura 2: Esquema dos principais atributos de DatabaseManager

**6. Implementação das Operações Sobre a Árvore**

**7. Comparação de Desempenho entre BST e AVL**

Para realizar a comparação de desempenho entre as operações das árvores, foram feitos sucessivos testes de cada operação utilizando diferentes quantidades de objetos. Como foram criadas duas listas com dez árvores cada, as operações foram feitas em cada árvore e então, a média do tempo em nanosegundos foi calculado junto com a quantidade média de comparações feitas.

Para explicar melhor, imagine que foi feita inserção de 50 objetos do tipo Escola em cada item da lista de BSTs. Então, da primeira à última árvore foram inseridos 50 objetos em cada, e desse modo foi calculado a média do tempo e o número médio de comparações. Isso também foi feito com a lista de AVLs. Foram feitos testes das três operações com as quantidades de 10, 30, 50 e 100 objetos do tipo Escola.

Primeiramente, é possível concluir que as BSTs performaram melhor que as AVLs nos testes de inserção. A figura 3 mostra como a inserção de apenas 10 objetos foi muito mais onerosa nas AVLs, exigindo mais que o dobro de tempo em alguns casos. Isso também vale para a inserção de uma quantidade maior de objetos. Como visto na figura 4, a inserção de 100 objetos foi relativamente bem mais demorada nas AVLs. No entanto, é possível ver que à medida que houve um aumento de objetos inseridos, o tempo médio de inserção nas árvores diminuiu, mas as diferença entre os dois tipos são notáveis.

A graph with red and blue bars

Description automatically generatedFigura 3: Gráfico comparando o tempo de inserção de 10 objetos em BSTs e AVLs.

A graph with blue and red bars

Description automatically generatedFigura 4: Gráfico comparando o tempo de inserção de 100 objetos em BSTs e AVLs.

Quando o número de comparações feitas durante a inserção nas árvores é feito, é visto que a BST está em desvantagem. Tanto para 10 itens e 100 itens, percebe-se que as BSTs têm um número médio maior de comparações, chegando a 50% maior em alguns casos.

A graph of different colored bars

Description automatically generatedFigura 5: Gráfico do número médio de comparações na inserção de 10 objetos em BSTs e AVLs.

A graph with blue and pink bars

Description automatically generatedFigura 6: Gráfico do número médio de comparações na inserção de 100 objetos em BSTs e AVLs.

Além disso, observa-se que a diferença do número médio de comparações entre as BSTs e AVLs se torna ainda mais evidente à medida que o número de itens inseridos aumenta. Com 100 objetos, a AVL continua em vantagem, realizando menos comparações em relação à BST, o que evidencia que a eficiência da AVL em manter sua estrutura balanceada mesmo com um volume maior de dados.

Essa vantagem da AVL é devido à sua diferença estrutural em relação à BST. Como a BST não possui um mecanismo de balanceamento automático, ela pode se tornar desbalanceada à medida que mais objetos são inseridos. Com isso, isso pode levar a altura da BST a ficar maior, o que aumenta o número de comparações necessárias para inserir novos elementos. Enquanto isso, a AVL mantém-se balanceada automaticamente. Depois de cada inserção, a AVL executa uma rotação simples ou dupla para garantir que a diferença de altura das duas subárvores esquerda e direita não tenha uma diferença maior que 1. Por isso, o balanceamento constante reduz o número de comparações necessárias.

Para as operações de busca, também foram feitas com quantidades de 10, 30, 50 e 100 objetos. Nessa operação, também é evidente a vantagem das AVLs. Nos testes realizados, pode-se ver que em alguns casos que o tempo necessário para busca é até 50% menor do que em BSTs. Como os testes de busca de 10 e 30 objetos apresentaram resultados bem diferentes entre as árvores do mesmo tipo, aqui será discutido os resultados das buscas de 50 e 100 objetos.

A graph with blue and red bars

Description automatically generated

Figura 7: Gráfico do tempo médio de busca de 50 itens em BSTs e AVLs

A graph with blue and red bars

Description automatically generated

Figura 8: Gráfico do tempo médio de busca de 100 itens em BSTs e AVLs

A figura 7 mostra como é mais demorado fazer uma busca em uma BST do que em uma AVL. Por exemplo, na primeira BST da lista, o tempo chega a ser quase 10 vezes maior do que o tempo de busca na primeira árvore da lista de AVLs.

Ainda mais, nota-se que quando o número de dados buscados é maior, o tempo médio da busca diminui na AVL, chegando a ser cinco vezes menor em alguns casos. Novamente, o fato de manter uma altura reduzida dá essa vantagem para a AVL, o que otimiza o tempo de busca. Com um número maior de dados, essa vantagem é ainda mais perceptível, já que a AVL continua realizando busca com eficiência, independente da quantidade de elementos armazenados.

Em contrapartida, a BST acaba realizando buscas em árvores desbalanceadas, e o tempo para encontrar o nó desejado aumenta. Por isso, quando o número de dados é maior, conclui-se que a BST tem uma queda de desempenho.

Em termos de número de comparações nas operações de busca, a AVL também é mais eficiente. Figuras 9 e 10 mostram claramente que quando se faz uma busca de 50 ou 100 objetos, a AVL consegue ser bem-sucedida com menos da metade da quantidade de comparações da BST. Novamente, isso se deve às diferenças estruturais entre as árvores, onde uma BST desbalanceada vai exigir muito mais comparações.

A graph with blue and pink bars

Description automatically generatedFigura 9: Gráfico da quantidade média de comparações de busca de 50 objetos em BSTs e AVLs.

A graph with blue and pink bars

Description automatically generatedFigura 10: Gráfico da quantidade média de comparações de busca de 100 objetos em BSTs e AVLs.

A graph with blue and red bars

Description automatically generated

Figura 11: Gráfico do tempo médio de remoção de 10 itens em BSTs e AVLs.

A graph with blue and red squares

Description automatically generated

Figura 12: Gráfico do tempo médio de remoção de 100 em BSTs e AVLs.

As figuras acima mostram claramente que a AVL é mais eficiente do que a BST na operação de remoção, especialmente quando o número de objetos removidos é maior. Na Figura 11, ao remover apenas 10 objetos, já dá para perceber que a AVL, mesmo fazendo balanceamento automático, consegue manter o tempo remoção menor. Em algumas árvores, o tempo de remoção da BST chega a ser o dobro do tempo na AVL.

Na Figura 12, a diferença de desempenho se torna ainda mais visível. O tempo médio de remoção nas BSTs continua mais alto, enquanto as AVLS mantêm uma performance mais uniforme e rápida. Isso acontece porque, ao remover elementos, a AVL ajusta a estrutura da árvore para evitar o desbalanceamento, o que reduz a profundidade dos nós.

11. Reflexão Final

11.1. Alan Meniuk Gleizer

Este projeto apresentou uma oportunidade interessante de lidar com bases de dados reais e bastante relevantes em um contexto no qual foi possível e importante focar também na implementação de estruturas de dados cujos detalhes costumamos abstrair. Desde o princípio, ficou claro que as árvores com as quais trabalhamos, ainda que eficientes para buscas e ordenações, não eram necessariamente as estruturas mais adequadas para uma análise do conteúdo. Ainda assim, foi desafiador adequar essas estruturas ao nosso uso e pensar na melhor maneira de representar os vários .csv como um único banco de dados. A solução final de listas de árvores envolveu bastante discussão no grupo e diversas modelagens da classe Escola e seus atributos.

Também gostei muito de tentar aplicar os princípios de programação orientada a objetos e de modelagem de software, que estamos estudando na disciplina de Projeto de Software, ao desenvolvimento do projeto. Mesmo se tratando de um programa bastante pequeno e simples, conceitos como encapsulamento, modularidade e responsabilidade única realmente ajudaram a estruturar um sistema coeso e bastante fácil de alterar conforme novas necessidades do grupo surgiram. Especificamente, classes, como Escola, CSVreader e DatabaseManager, têm responsabilidades bem definidas: Escola armazena os dados de cada escola/linha, CSVreader cuida da leitura e processamento de um arquivo CSV (por instância), e DatabaseManager coordena o armazenamento em árvores e a organização geral dos dados. Os métodos das funções também foram modularizados de forma que, antes mesmo da seleção do dataset, seria possível adequá-las para a leitura de outros tipos e outras quantidades de arquivos.