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 10342005

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

**Sumário**

**1. 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 utilizados 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).

**2. Escolha do conjunto de dados**

Para este projeto, foi usado o conjunto de dados “Quantidade de alunos matriculados em Centros de Estudo de Línguas (CEL)” (SÃO PAULO, 2023). Ele inclui informações sobre os CELs da Rede Estadual de Ensino por idioma ensinado, sendo os idiomas, neste caso: Inglês, espanhol, francês, italiano, alemão, japonês e mandarim, contendo dados referentes às quantidades de alunos estudando essas línguas, as escolas que ensinam, a região onde estão localizadas, entre outras informações. Todos os atributos estão presentes na tabela abaixo disponibilizada pelo próprio site do governo. O link do conjunto de dados e o dicionário podem ser encontrados nas referências em SÃO PAULO, 2023.

O conjunto de dados estava armazenado na forma de arquivos de extensão CSV divididos por semestres de análise, havendo dados desde o primeiro semestre de 2019 até o segundo semestre de 2023.

Os dados incluem o código da escola como também incluem o nome município onde está localizada, com o código do município e o código da diretoria de ensino a qual a escola pertence (por exemplo: Norte 1, Norte 2, Centro, Leste 1). Informações geográficas como essas são importantes para realizar as análises geográficas deste projeto. Igualmente, o conjunto de dados indica a quantidade de alunos matriculados em cada língua ensinada por cada instituição ensino, e seus valores totais. Ele também divide a quantidade de alunos matriculados nas instituições de ensino tanto durante a semana quanto em finais de semana.

De certo, o Objetivo de Desenvolvimento Sustentável (ODS) da Organização das Nações Unidas (ONU), ao qual os dados melhor se encaixam é o ODS 4: Educação de Qualidade, já que, com as informações podem-se tomar conclusões relevantes em relação ao ensino de línguas estrangeiras no Estado de São Paulo.



Figura x: ícone do ODS 4 da ONU.

O ensino de línguas estrangeiras é um tópico importante no mundo globalizado que vivenciamos atualmente. Em um mundo onde a internet conecta cada vez mais pessoas de diversos lugares do mundo por meio da internet, onde as redes sociais e o trabalho remoto em países estrangeiros se tornam mais comum, se torna cada vez mais importante saber ao menos uma segunda língua além do português (DAY, 2012).

Infelizmente, o Brasil ainda enfrenta desafios no ramo da educação como um todo, e o ensino de línguas está incluso nisso. Esse problema é histórico e já existe desde a época do Brasil império, no século XIX. Os alunos ainda têm dificuldade de perceber a relevância que línguas como o inglês terão no futuro deles, os professores não são bem remunerados e o ensino de línguas estrangeiras ainda prevalece mais nas instituições de ensino privadas (BRITISH COUNCIL, 2015).

**3. Modelagem dos Dados em Árvores**

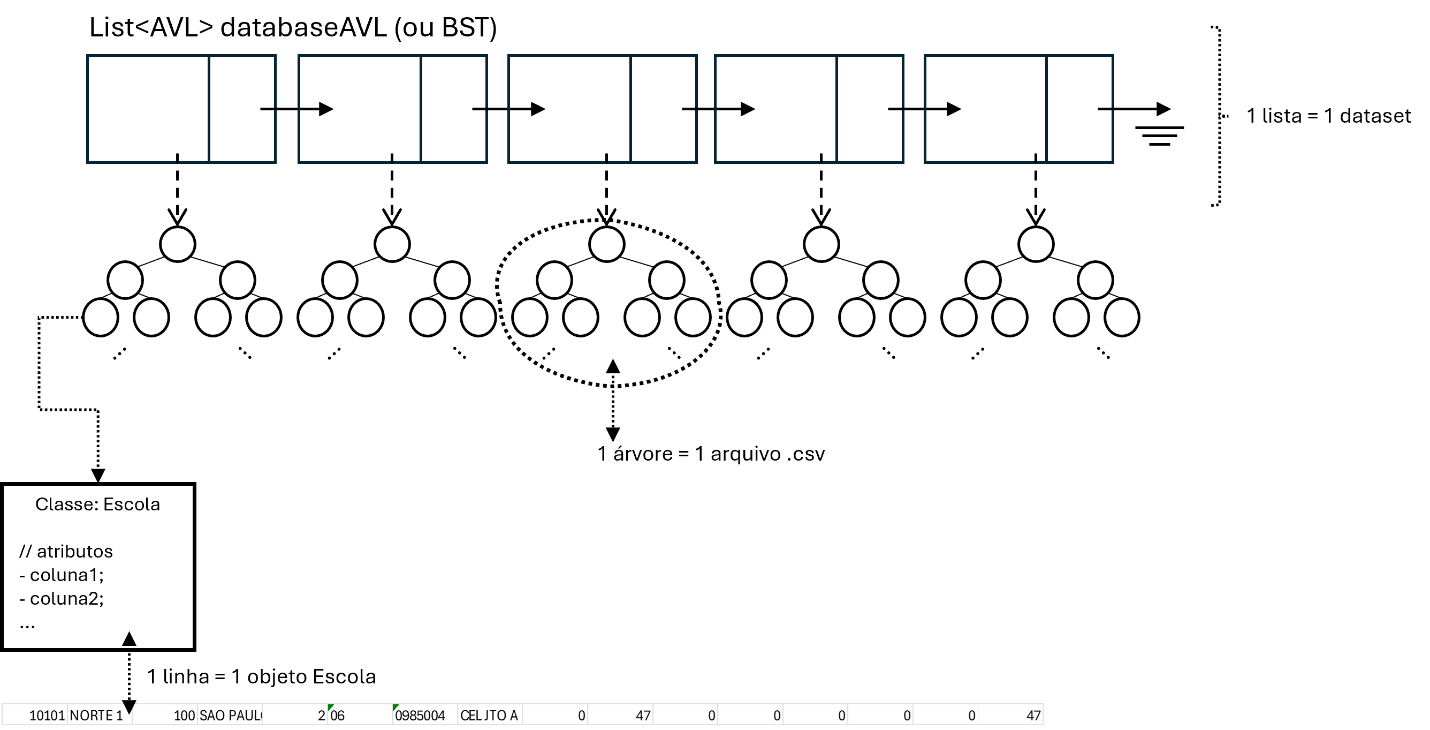
Neste projeto, o objetivo principal foi estruturar e organizar dados de instituições de ensino a partir de um conjunto de registros de escolas, de maneira que permitisse a busca e inserção eficientes. Para isso, utilizamos duas estruturas de dados baseadas em árvores: a árvore binária de busca (BST) e a árvore AVL, uma variação da BST que realiza autobalanceamento. Na medida que o conjunto de dados selecionado armazena as informações segundo a lógica de uma escola por linha, e um arquivo CSV por semestre, a organização dos dados nas árvores foi realizada de forma a espelhar a estrutura dos dados. Partindo de uma visão micro para macro, a organização 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 do arquivo 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 arquivo 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 arquivo 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 os dados como um todo. Como o conjunto de dados 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 dos dados dentro do programa pode ser bem representada pelo esquema abaixo:

Figura 1: representação da estrutura de organização dos dados

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 arquivo 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, a fim 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

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

As operações sobre a árvore foram implementadas por meio de métodos dentro de suas respectivas classes. A classe BST tem a implementação dos métodos que são usados pelas duas árvores por meio de herança. A classe AVL, além de herdar todos os métodos da BST, também possui métodos próprios exclusivos dela.

A BST possui implementado todas as principais formas de travessia por árvore binária: pré ordem, em ordem, pós ordem e por nível. Porém, ao invés da travessia aqui ser usada para imprimir todos os elementos da árvore, ela é usada para devolver todos os elementos da árvore na forma de uma arrayList (importada da biblioteca java.util.ArrayList) de dados do tipo Escola. Para cada nó visitado, ela adiciona a escola à ArrayList por meio do método add dessa classe (Figura x). A travessia em ordem é particularmente interessante pois permite manter as escolas ordenadas (BARNETT & TONGO, 2008; GOODRICH et al., 2011). por meio de seus números de identificação na ArrayList. Para todas essas travessias, tinha o método principal que é o chamado por outras classes, e um método auxiliar privado que é o que efetivamente faz a travessia, e é chamado pelo método principal. A implementação da maior parte deles foi recursiva.

A screen shot of a computer code

Description automatically generated

Figura 3: Implementação da travessia em ordem na classe BST.

Basicamente, o que diferencia esses métodos de travessia é a ordem em que o dado é adicionado à ArrayList e quando os nós da esquerda e direita são acessados. No caso em ordem, primeiro é acessado o nó da esquerda, depois a escola é adicionada à ArrayList e por último é acessado o nó da direita. A única travessia implementada de forma iterativa foi a travessia por nível, a qual usa um loop while e uma fila, por isso foi necessário importar também as classes LinkedList e Queue do próprio Java.

O método de busca foi implementado de forma recursiva e usa o código da escola como chave para determinar para qual lado seguir durante a busca. Se o elemento sendo procurado for maior que o código de escola atual, a busca continua para direita, caso contrário, vai para esquerda. Foi escolhido o código da escola como chave pois é o único atributo do conjunto de dados escolhido que não possui elementos repetidos. De forma parecida com a travessia, tem o método principal público e o método auxiliar privado, uma estratégia que foi usada para muitos outros métodos da implementação (Figura x).

// Busca por elementos na arvore

    Node search(int value){

        if(root==null) return null;

        return searchAux(root, value);

    }

    // Auxiliar de search()

    private Node searchAux(Node currentNode, int value){

        if(currentNode.getData().getCodEsc() == value) return currentNode;  // Caso tenha encontrado o valor

        // Verifica qual lado eh menor e qual eh maior para determinar onde inserir

        if(value>currentNode.getData().getCodEsc()) return searchAux(currentNode.getRight(), value);

        if(value<currentNode.getData().getCodEsc()) return searchAux(currentNode.getLeft(), value);

        return null;

    }

Figura 4: implementação do método de busca na classe BST.

Uma variante do método de busca inclui o searchContagem, o qual faz o mesmo tipo de busca, porém, retorna a quantidade de comparações realizadas, necessário para a comparação dos dois tipos de árvores, utilizando a classe Contagem para isso. Foram feitas variantes parecidas com os métodos de inserção e de remoção, também com o intuito de auxiliar na comparação de performance dos dois tipos de árvores.

O método inserção funciona de forma similar ao de busca, porém ela continua em execução até ele encontrar um nó vazio, isso é, sem elemento, que é o lugar onde o novo nó deve ser inserido (Figura x).

// Para inserir novos nós (sem respeitar AVL)

    void insert(Escola value){

        root = insertAux(root, null, value);

    }

    private Node insertAux(Node currentNode, Node parentNode, Escola value){

        Node newNode = new Node(value);

        newNode.setParent(parentNode);

        if(currentNode==null) return newNode;

        // Verifica qual lado eh menor e qual eh maior para determinar onde inserir

        if(value.getCodEsc()<currentNode.getData().getCodEsc()) currentNode.setLeft(insertAux(currentNode.getLeft(), currentNode, value));

        else currentNode.setRight(insertAux(currentNode.getRight(), currentNode, value));

        return currentNode;

    }

Figura 5: Implementação do método de busca na classe BST.

O método getMax retorna o nó com o maior código escola, o qual é usado pelo método de busca do antecessor. O getMax percorre a árvore continuamente para a direita até não terem mais nós por onde percorrer (Figura x). De forma parecida, o getMin, que retorna o menor código de escola, percorre continuamente para a esquerda (Figura x).

// Retorna o maior elemento da arvore

    Node getMax(){

        return getMaxAux(root);

    }

    // Auxiliar de getMax

    // Tambem pode ser usado para pegar o maior elemento de uma subarvore especificamente

    private Node getMaxAux(Node currentNode){

        if(currentNode == null) return null;

        if(currentNode.hasRightChild()) return getMaxAux(currentNode.getRight());  // Percorre continuamente para a direita

        else return currentNode;

    }

    // Retorna o menor elemento da arvore

    Node getMin(){

        return getMinAux(root);

    }

    // Auxiliar de getMin

    // Tambem pode ser usado para pegar o menor elemento de uma subarvore especificamente

    private Node getMinAux(Node currentNode){

        if(currentNode == null) return null;

        if(currentNode.hasLeftChild()) return getMinAux(currentNode.getLeft());  // Percorre continuamente para a esquerda

        else return currentNode;

    }

Figura 6: Implementações dos métodos getMax e getMin.

Além disso, também foram implementadas funções que retornam o antecessor e o sucessor de um determinado nó. O antecessor busca pelo maior elemento da subárvore esquerda do nó escolhido. Caso não exista uma subárvore, então é feita uma travessia “para cima” até chegar em um nó que seja filho à direita de seu pai (Figura x). Um mecanismo análogo é feito para achar o sucessor, porém usa o getMin e caso contrário procura por um filho esquerda de seu pai.

// Acha o sucessor de um nó

    Node getSuccessor(Node currentNode){

        if(currentNode.hasRightChild()) return getMinAux(currentNode.getRight()); // Vai para a subarvore se tiver

        Node parentNode = currentNode.getParent();

        while(parentNode!=null && parentNode.getRight() == currentNode){ // Sobe se não tiver subarvore

            currentNode = parentNode;

            parentNode = currentNode.getParent();

        }

        return parentNode;

    }

    // Acha o antecessor de um nó

    Node getAntecessor(Node currentNode){

        if(currentNode.hasLeftChild()) return getMaxAux(currentNode.getLeft()); // Vai para a subarvore se tiver

        Node parentNode = currentNode.getParent();

        while(parentNode!=null && parentNode.getLeft() == currentNode){ // Sobe se não tiver subarvore

            currentNode = parentNode;

            parentNode = currentNode.getParent();

        }

        return parentNode;

    }

Figura 7: implementações dos métodos que retornam o sucessor e o antecessor de um dado nó na classe BST.

O método de remoção é um dos mais complexos no programa, pois ele deve analisar vários diferentes casos. De início ele simplesmente busca pelo nó a ser removido, reutilizando a função de busca para isso. O caso 1 é o mais simples é quando a remoção é de uma folha, pois o nó é removido diretamente. O caso 2 é quando a remoção é de um nó interno com um filho, o que necessita com que seja feita uma realocação das referências de forma que uma referência “pule” o nó removido para chegar no filho único desse nó. O caso 3 é quando tem 3 filhos, o que exige que seja feita uma troca de dados entre o nó a ser removido e seu sucessor.

// Remove um nó com determinado valor

    void removeNode(Escola value){

        Node nodeToRemove = search(value.getCodEsc());  // Primeiro procura o Node em si, reutiliizando a funcao search

        if(nodeToRemove == null) return;  // Caso nao ache o Node para remover

        Node parentNode = nodeToRemove.getParent();  // Pega o pai de quem vai ser removido. Necessario para a atualizacao das referencias.

        if(nodeToRemove.isLeaf()){  // Caso 1:  Se o removido for uma folha

            if(parentNode != null){

                // Atualiza referencias do Node pai do removido

                if(parentNode.getRight() == nodeToRemove) parentNode.setRight(null);

                else if(parentNode.getLeft() == nodeToRemove) parentNode.setLeft(null);

            } else root = null;

            return;

        }

        // Caso 2:  Se o Node removido tem apenas um filho

        if(nodeToRemove.getDegree() == 1){

            nodeToRemove.setParent(null);  // Remove referencia ao pai do removido

            Node aux;

            if(nodeToRemove.hasLeftChild()) {  // Se o unico filho esta a esquerda do removido

                aux = nodeToRemove.getLeft();

                nodeToRemove.setLeft(null);  // Remove referencia a esquerda do removido

            }

            else {  // Se o unico filho esta a direita do removido

                aux = nodeToRemove.getRight();

                nodeToRemove.setRight(null);  // Remove referencia a direita do removido

            }

            // Atualizacao das referencias dos Nodes

            if(parentNode.getRight() == nodeToRemove) parentNode.setRight(aux);

            if(parentNode.getLeft() == nodeToRemove) parentNode.setLeft(aux);

            return;

        }

        if(nodeToRemove.getDegree() == 2){  //Caso 3: Se o removido tem 2 filhos

            Escola successorData = getSuccessor(nodeToRemove).getData();  // Removido troca de dado com o sucessor (antecessor tambem funcionaria)

            removeNode(successorData);

            nodeToRemove.setData(successorData);  // Apos a troca, remove o sucessor (o qual agora esta com o dado do removido)

        }

    }

Figura 8: Implementação do método de remoção na classe BST.

Quanto aos métodos exclusivos da classe AVL, temos os métodos de rotação, os quais são necessários para manter o balanceamento da árvore AVL durante os processos de remoção e inserção (BARNETT & TONGO, 2008; GOODRICH et al., 2011). No caso da rotação da direita, primeiro, ele identifica o nó auxiliar, que é o filho esquerdo do nó passado como parâmetro. Esse nó auxiliar, aux, se tornará a nova raiz da subárvore após a rotação. Em seguida, o código move a subárvore direita de aux para o filho esquerdo de currentRoot, dessa forma, mantendo a estrutura da árvore (Figura x).

// Rotacao para a direita

    Node rightRotate(Node currentRoot){

        Node aux = currentRoot.getLeft();

        currentRoot.setLeft(aux.getRight());

        aux.setRight(currentRoot);

        aux.setParent(currentRoot.getParent());

        currentRoot.setParent(aux);

        currentRoot.updateHeight();

        aux.updateHeight();

        return aux;  // Retorna a raiz da subarvore

    }

    // Rotacao para a esquerda

    Node leftRotate(Node currentRoot){

        Node aux = currentRoot.getRight();

        currentRoot.setRight(aux.getLeft());

        aux.setLeft(currentRoot);

        aux.setParent(currentRoot.getParent());

        currentRoot.setParent(aux);

        currentRoot.updateHeight();

        aux.updateHeight();

        return aux;  // Retorna a raiz da subarvore

    }

Figura 9: implementação das rotações para esquerda e direita.

O processo inverso é feito para a rotação da esquerda (Figura x). Ademais, nos casos das rotações direita esquerda e esquerda direita, as funções são reutilizadas porém primeiro para o filho do nó desbalanceado e depois o nó desbalanceado em si.

A inserção na AVL é feita de forma similar à inserção na BST porém, após realizar a inserção, por meio da volta recursiva, cada nó anterior tem o seu fator de balanceamento recalculado para determinar se a árvore continua balanceada ou não (Figura x). Se algum fator de balanceamento resultar em 2 ou -2, é necessário realizar rotações. Vale lembrar que nesta implementação, o fator de balanceamento é calculado como altura da esquerda menos altura da direita. Logo, se o fator for dois, é porque a árvore pesa mais para a esquerda, enquanto -2 pesa mais para a direita.

// Funcao para inserir novo Node na arvore, porem de forma que respeita o balanceamento

    void insertAVL(Escola value){

        root = insertAVLAux(root,null,value);

    }

    // Auxiliar de insertAVL

    private Node insertAVLAux(Node currentNode, Node parentNode, Escola value){

        // Encontrou o local de insercao

        if(currentNode == null){

            Node newNode = new Node(value);  // Instancia o novo Node e insere ele

            newNode.setParent(parentNode);

            return newNode;

        }

        if(value.getCodEsc()<currentNode.getData().getCodEsc()) {

            currentNode.setLeft(insertAVLAux(currentNode.getLeft(), currentNode, value));  // Insere na esquerda caso o valor seja menor

            currentNode.updateHeight();  // Atualiza altura e verifica balanceamento do no

            if(currentNode.balancingFactor() == 2){  // Mais pesado na esquerda

                if(value.getCodEsc()<currentNode.getLeft().getData().getCodEsc()) currentNode = rightRotate(currentNode);  // Filho do desbalanceado na esquerda. Necessita rotacao simples direita

                else if(value.getCodEsc()>currentNode.getLeft().getData().getCodEsc()) currentNode = leftRightRotate(currentNode);  // Filho do desbalanceado na direita. necessita rotacao esquerda direita

            }

        }

        else if(value.getCodEsc()>currentNode.getData().getCodEsc()){

            currentNode.setRight(insertAVLAux(currentNode.getRight(), currentNode, value));;  // Insere na direita caso o valor seja maior

            currentNode.updateHeight();  // Atualiza altura e verifica balanceamento do no

            if(currentNode.balancingFactor() == -2){  // Mais pesado na direita

                if(value.getCodEsc()>currentNode.getRight().getData().getCodEsc()) currentNode = leftRotate(currentNode);  // Filho do desbalanceado na direita. Necessita rotacao simples esquerda

                else if(value.getCodEsc()<currentNode.getRight().getData().getCodEsc()) currentNode = rightLeftRotate(currentNode);  // Filho do desbalanceado na esquerda. necessita rotacao direita esquerda

            }

        }

        currentNode.updateHeight();  // Atualiza a altura do no atual

        return currentNode;  // Retorna raiz da subarvore

    }

Figura 10: Implementação de inserção na classe AVL.

Similarmente, o método de remoção na AVL, durante sua volta recursiva, também recalcula os fatores de balanceamento dos pais e faz as devidas rotações se necessário. Os mesmos caso da BST também são analisados, isso é, se o nó removido é uma folha, se é um nó interno com um filho ou se é um nó interno com dois filhos.

// Funcao para remover um Node da AVL. Override

    void removeNode(int value){

        root = removeNodeAVLAux(root, value);

    }

    // Remocao mantendo o balanceamento da AVL. Auxiliar de removeNode

    private Node removeNodeAVLAux(Node currentNode, int value) {

        if (currentNode == null) return currentNode; // Caso nao tenha nada para ser removido

        if (value < currentNode.getData().getCodEsc()) {  // Procurando pelo Node a ser removido

            currentNode.setLeft(removeNodeAVLAux(currentNode.getLeft(), value));  // Menor, entao vai para esquerda

        } else if (value > currentNode.getData().getCodEsc()) {

            currentNode.setRight(removeNodeAVLAux(currentNode.getRight(), value));  // Maior, entao vai para direita

        } else {

            // No com nenhum ou apenas um filho

            if (currentNode.getLeft() == null || currentNode.getRight() == null) {

                Node temp = currentNode.getLeft() != null ? currentNode.getLeft() : currentNode.getRight();

                // Caso 1:  Sem filhos

                if (temp == null) {

                    temp = currentNode;

                    currentNode = null;

                } else {

                    currentNode = temp; // Caso 2: Apenas um filho

                }

            } else {

                Node temp = currentNode.getRight();

                while (temp.getLeft() != null) {

                    temp = temp.getLeft();

                }

                // Copiando o dado do sucessor

                currentNode.setData(temp.getData());

                // Deletando o sucessor

                currentNode.setRight(removeNodeAVLAux(currentNode.getRight(), temp.getData().getCodEsc()));

            }

        }

        // Se a arvore soh contem um no retornar

        if (currentNode == null) return currentNode;

        // Checando o balanceamento

        int balance = currentNode.balancingFactor();

        if (balance == 2) {  // Desbalanceado para a esquerda

            // Vendo de que lado esta desbalanceado

            if (currentNode.getLeft().balancingFactor() >= 0) {  // Inserido a esquerda do desbalanceado

                currentNode = rightRotate(currentNode);  // Rotacao simples a direita

            } else {

                currentNode = leftRightRotate(currentNode);  // Rotacao esquerda direita

            }

        } else if (balance == -2) {  // Desbalanceado para a direita

            if (currentNode.getRight().balancingFactor() <= 0) {  // Inserido a direita do desbalanceado

                currentNode = leftRotate(currentNode);  // Rotacao simples a esquerda

            } else {

                currentNode = rightLeftRotate(currentNode);  // Rotacao direita esquerda

            }

        }

        return currentNode;

    }

Figura 11: Implementação da remoção na classe AVL.

É por meio dessas implementações que todos os dados do database são guardados. Cada nó contém um dado do tipo escola o qual contém todas as informações da dada escola para um ano específico. Como dito anteriormente, o atributo usado como chave para a construção da árvore é o código da escola.

**5. 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 diferenças entre os dois tipos são notáveis.

A graph with red and blue bars

Description automatically generatedFigura 12: Gráfico comparando o tempo de inserção de 10 objetos em BSTs e AVLs.

A graph with blue and red bars

Description automatically generatedFigura 13: 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 14: 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 15: Gráfico do tempo médio de busca de 50 itens em BSTs e AVLs

A graph with blue and red bars

Description automatically generated

Figura 16: 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 17: 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 18: 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 19: Gráfico do tempo médio de remoção de 10 itens em BSTs e AVLs.

A graph with blue and red squares

Description automatically generated

Figura 20: 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.

A graph with blue and pink bars

Description automatically generated

Figura 21: Gráfico da quantidade média de comparações de remoção de 10 objetos em BSTs e AVLs.

A graph with blue and pink bars

Description automatically generated

Figura 21: Gráfico da quantidade média de comparações de remoção de 100 objetos em BSTs e AVLs.

Os gráficos acima ilustam bem como as árvores AVL são mais eficientes nas quantidades de comparações feitas durante a remoção de itens, tanto para 10 quanto para 100 objetos. Na Figura 13, com a remoção de 10 itens, fica claro que as AVLs necessitam de menos comparações na maior ia dos casos, comparado às BSTs. Esse menor número de comparações reflete o balanceamento das AVLs, que mantém os nós mais próximos da raiz, facilitando o acesso aos elementos e diminuindo o trabalho necessário para localizar e remover os nós.

Quando o número de objetos removidos aumenta, como na Figura 14 com 100 itens, essa diferença se mantém. A AVL continua se destacando com uma quantidade média de comparações menor, reforçando a eficiência desse tipo de árvore em operações de remoção. Como a AVL está constantemente balanceada, a profundidade média dos nós se mantém baixa, o que reduz a quantidade de comparações necessárias para encontrar e remover os elementos.

No geral, os resultados mostram que as árvores AVLs levam vantagem sobre as BSTs nas três operações. A AVL mantém um desempenho mais consistente e eficiente graças ao balanceamento automático, que limita a altura da árvore e reduz o número de comparações e o tempo necessário para cada operação. Em operações de busca e remoção, essa diferença se torna ainda mais evidente, com a AVL sendo mais rápida e fazendo menos comparações, enquanto a BST sofre com o desbalanceamento, ficando mais lenta à medida que os dados aumentam. Então, em aplicações que exigem muitas operações e dados variados, a AVL é a escolha ideal, garantindo uma performance superior e mais estável.

**6. Testes e Resultados**

Para este projeto, a manipulação de dados foi feita com o propósito de responder à pergunta: “Entre 2019 e 2023, houve um aumento médio do número de alunos matriculados nos cursos de Língua Inglesa e Língua Espanhola nos CELs do Estado de São Paulo?”. A metodologia consistiu nos seguintes passos:

**6.1 Agregação dos Dados**

O conjunto inicial de dados era composto por diversos arquivos CSV, cada um representando semestres de diferentes anos. Esses dados foram consolidados em uma estrutura única, facilitando a análise e permitindo o cálculo do número total de alunos matriculados por idioma, diretoria e semestre.

**6.2 Estruturação da Análise**

Para cada diretoria de ensino, a quantidade de alunos matriculados em cursos de idioma foi extraída e organizada em uma tabela, separa por semestre. Essa estrutura possibilitou a comparação do crescimento ou declínio nas matrículas ao longo do tempo, e para este projeto, a análise foi feita para Inglês e Espanhol.

**6.3 Análise dos Resultados**

A partir da tabela gerada, foi possível observar tendências de matrículas. É importante considerar que o período entre os anos 2019 e 2023 foram atípicos devido à pandemia do Covid-19, na qual muitos alunos das escolas públicas ficaram sem acesso aos recursos educacionais normalmente disponíveis. Por isso, é possível ver oscilações significativas de um semestre para o outro. Também, é importante notar que o número de alunos matriculados pode variar bastante de um semestre para o outro devido à fatores políticos, como a decisão de transferir um Centro de Ensino de Línguas para outra diretoria, ou simplesmente fechá-lo. Aqui, não há como lavar em consideração esses acontecimentos.

Primeiramente, é possível concluir que entre 2019 e 2023, o número de alunos matriculados em Centro de Ensino de Línguas na Rede Estadual de São Paulo diminuiu. Os detalhes dos cálculos podem ser vistos do Jupyter Notebook referente à análise dos resultados dos Centros de Estudo de Línguas, no qual o link está no Apêndice A. Para os alunos matriculados em inglês, houve uma diminuição média de aproximadamente 17% no número de alunos matriculados em 2023 em comparação a 2019. Porém, esse resultado não é uniforme por todas as diretorias de ensino.

A table with numbers and letters

Description automatically generated

Tabela 1: As dez diretorias de ensino com o maior aumento de alunos matriculados em cursos de inglês

A Tabela 1 mostra as dez diretorias de ensino onde houve os maiores aumentos nos números de alunos matriculados em aulas de inglês. Em alguns casos, como na Diretoria de Ensino Centro Sul, houve um aumento de quase 300%, mostrando que em pelo menos alguns lugares, há demanda pelo aprendizado de inglês. É importante notar que das dez diretorias de ensino listadas, seis estão na região metropolitana de São Paulo.

A table with numbers and numbers

Description automatically generated

Tabela 2: As dez diretorias de ensino com o maior declínio de alunos matriculados em cursos de inglês.

Ao mesmo tempo, a Tabela 2 mostra as dez diretorias de ensino onde houve as maiores diminuições de alunos matriculados em 2023 em comparação a 2019. Enquanto média no Estado de São Paulo foi de 17%, a média das diretorias listadas na tabela é de quase 60%. Vale ressaltar que muitas dessas diretorias de ensino estão localizadas em centros populacionais do estado, como a própria capital São Paulo, Presidente Prudente, Guarulhos etc.

Igualmente, houve uma diminuição no número de alunos matriculados em cursos de Língua Espanhola. Porém, essa redução foi bem mais drástica. De acordo com os dados, houve uma diminuição de mais de 40% entre 2019 e 2023. Como pode ser visto na Tabela 3, somente cinco diretorias de ensino tiveram um aumento no número de alunos nos cursos de espanhol.

A table with numbers and numbers

Description automatically generated

Tabela 3: As dez diretorias de ensino com o maior aumento, ou menor declínio, de alunos matriculados em cursos de espanhol

**6.4 Relação com a ODS 4**

A análise do problema apresentado pode ser relacionada com a ODS 4, que visa garantir uma educação inclusiva e de qualidade. Essa conexão pode ser feita a partir dos seguintes pontos:

**6.4.1 Educação Inclusiva**

O declínio observado nas matrículas, especialmente em regiões afastadas, indica deságios na inclusão de estudantes em programas educacionais de qualidade, como os cursos de línguas. A falta de acesso, acentuada pela pandemia, reforça as desigualdades já existentes no sistema educacional. Uma meta que poderia ser implementada seria garantir que todos os alunos concluam o ensino básico com resultados relevantes de aprendizagem de línguas. Por exemplo, uma metodologia a ser seguida seria o Quadro Europeu Comum de Referência para as Línguas (QECR), que tem sido usado para descrever resultados de aprendizagem de uma língua estrangeira não só na Europa, mas em vários países do mundo.

**6.4.2 Promoção de Competências Globais**

Os cursos de idiomas, como inglês e espanhol, são fundamentais para preparar os alunos para o mercado de trabalho e ampliar sua participação em uma sociedade global. O declínio das matrículas representa uma barreira ao cumprimento da ODS 4.4, que visa aumentar substancialmente o número de jovens e adultos com competências relevantes para o emprego e o empreendedorismo.

**6.4.3 Desigualdades Regionais**

A análise de diferenças entre diretorias de ensino mostra que regiões metropolitanas, como São Paulo, apresentam maior acesso ou recuperação na oferta de cursos, enquanto regiões mais afastadas enfrentam um declínio mais significativo. Isso demonstra uma desigualdade no cumprimento do ODS 4.5, que busca eliminar disparidades na educação, garantindo igualdade de acesso para todos os níveis de ensino.

**6.4.5 O Papel do Investimento Público**

O fechamento de Centros de Ensino de Línguas ou sua transferência para outras diretorias reflete a influência de decisões políticas na oferta educacional. Para alinhar-se ao ODS 4, é fundamental assegurar investimentos contínuos na manutenção e expansão de programas educacionais, garantindo oportunidades de aprendizagem para todos os alunos.

**6.4.5 Conclusão sobre a ODS**

O declínio de matrículas de inglês e espanhol nos Centros de Ensinos de Línguas da Rede Estadual de São Paulo evidencia a necessidade de esforços mais concretos para alcançar os objetivos da ODS 4. Isso inclui ações como: melhorar a infraestrutura educacional para atender todos os alunos, garantir a continuidade dos programas em tempos de crise, incentivar o aprendizado de línguas como ferramenta para o desenvolvimento pessoal e profissional.

**7. Reflexão Final**

**7.1. 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 arquivos 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.

**7.2. Caio Vinicius Corsini Filho**

A APL II foi um desafio muito interessante e gostei da proposta do projeto. Apesar de projetos com um enfoque mais teórico como da APL I também serem interessantes quanto ao aprendizado da disciplina eu sinto que também gosto bastante de projetos mais focados na aplicação dos conhecimentos que adquirimos no curso de Ciência da Computação.

No meu caso, eu trabalhei nas implementações das árvores BST e AVL, e participei um pouco das análises de performance com a implementação da classe Contagem e com o desenvolvimento de alguns dos métodos de contagem. Me surpreendeu muito como as AVLs demoram mais para realizar inserção comparado às BST. Pensava que a AVL seria mais rápida devido ao seu balanceamento, mas ao que parece as rotações realmente deixam a execução muito mais lenta. É bom poder ver esse tipo de fato na prática e serve para mostrar que não existe uma “bala de prata” quando se fala de estrutura de dados e algoritmos.

Certamente, meu maior desafio foi implementar a AVL, especialmente as rotações, pois não é um algoritmo muito intuitivo, e precisei até fazer testes de mesa e desenhos para conseguir entender o que acontecia. Mas, no final das contas, isso me ajudou a deixar muito mais claro como essa TAD funciona.

**7.3 Giberto De Melo Júnior**

Quando aprendemos sobre ciência de dados e as ferramentas utilizadas, é mais comum se aprofundar na análise das informações do que como elas são processadas. Bibliotecas da linguagem Python como Pandas e Matplotlib tiram toda a complexidade da extração dos dados. Porém, neste projeto, foi interessante estudar as diferentes possibilidades de extrair, transformar e apresentar os dados pesquisados utilizando estruturas de dados em Java.

Como a utilização da BST e da AVL eram necessárias para a execução do projeto, ficaram poucas opções de como organizar os objetos criados a partir do conjunto de dados. Porém, para a transformação dos dados para apresentar os resultados, foi interessante a utilização de HashMaps, sendo que ficou claro que essas estruturas têm um mapeamento mais linear com estruturas já tabeladas como arquivos CSV. Assim, não é difícil de imaginar que possa haver situações em que bibliotecas como a Pandas tenham limitações, e que será necessário para o analista implementar suas próprias estruturas de dados para casos específicos. Talvez isso seja um diferencial de alguém que estuda computação e não apenas ciência de dados: saber implementar soluções específicas e entender os processos subjacentes ao invés de apenas consumir ferramentas externas.

Por último, foi interessante trabalhar com dados que refletem diretamente a situação de que muitos estudantes da rede pública de São Paulo passaram: o descaso e falta de apoio durante a pandemia de Covid-19. Os números refletiram a assimetria dos recursos estaduais para o ensino de idiomas no estado, e como algumas regiões se recuperaram rapidamente enquanto outras não se recuperaram.

**7. Referências**

BARNETT, Granville & TONGO, Luca del. **Data Structures and Algorithms**. 1.ed. 2008.

BRITISH COUNCIL. O Ensino de Inglês na Educação Pública Brasileira. 1. Ed. São Paulo, 2012. Disponível em: chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.britishcouncil.org.br/sites/default/files/estudo\_oensinodoinglesnaeducacaopublicabrasileira.pdf

DAY, Kelly. **ENSINO DE LÍNGUA ESTRANGEIRA NO BRASIL: ENTRE A ESCOLHA OBRIGATÓRIA E A OBRIGATORIEDADE VOLUNTÁRIA**. **REVISTA ESCRITA**, v. 2012, n. 15, 12 dez. 2012. DOI [10.17771/PUCRio.escrita.20850](https://doi.org/10.17771/PUCRio.escrita.20850). Disponível em: <http://www.maxwell.vrac.puc-rio.br/Busca_etds.php?strSecao=resultado&nrSeq=20850@1>. Acesso em: 10 nov. 2024.

GOODRICH, M. T.; TAMASSIA, R., MOUNT, M.N. **Data Structures and Algorithms** in C++. 2.ed. New Yok: Wiley, 2011.

SÃO PAULO. **Quantidade de alunos matriculados em Centros de Estudo de Línguas (CEL)**. Dados abertos sobre educação. Coordenadoria de Governo Aberto, 2024. Disponível em: <https://dados.educacao.sp.gov.br/dataset/quantidade-de-alunos-matriculados-em-centros-de-estudo-de-l%C3%ADnguas-cel>.

8. Apêndice A

1. Link para Jupyter Notebook contendo a análise de performance das árvores BST e AVL: <https://github.com/caiocorsini/APL-II---Estrutura-de-dados/blob/main/informacoes_projeto/analise_dados_performance.ipynb>
2. Link para Jupyter Notebook contendo a análise dos dados sobre os Centros de Ensino de Línguas por Diretoria de Ensino: <https://github.com/caiocorsini/APL-II---Estrutura-de-dados/blob/main/datasets/analise_resultados_cel.ipynb>