



UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

ALYSSON ALEXANDRE DE OLIVEIRA ARAÚJO

JOÃO ALMIR DA COSTA JUNIOR

Quixadá-CE

Introdução

O objetivo do projeto se trata da implementação da Árvore AVL através da programação orientada a objetos, para fazer uma pesquisa rápida e efetiva das pessoas, para isso, utiliza-se 3 árvores contendo um dado dessas pessoas como chave, sendo esses: CPF, nome e Data de Nascimento.

Para fazer uma classe árvore que aceite esse 3 tipos de dados, é necessário criar uma árvore genérica através de templates, tornando assim, uma árvore que aceita vários tipos como chave.

Os dados da pessoas são CPF, nome, sobrenome, data de nascimento e cidade de nascimento. Excluindo o CPF todos os outros dados podem se repetir, é preciso que a árvore aceite dados repetidos.

A entrada feitas através de um arquivo data.csv que possui uma tabela com todos os dados das pessoas.

Implementação

Foram feitas 3 árvores genéricas através de template com CPF, nome e data de nascimento como chaves.

Árvore CPF

Na árvore do CPF a entrada é uma string com ponto no caractere 4 e 8 além de um traço no caractere 12.

xxx.xxx.xxx-xx

Para colocar essa string cpf como chave na árvore é preciso converter para um inteiro, só que, um int não suporta números tão grandes assim, portanto é necessário transformar em um long int.

Para realizar isso foi feito um laço que percorre a string CPF e adiciona à uma string auxiliar sem os pontos e traço, depois se converte para long int através da função `std::stol`

Árvore Nome

Na Árvore nome a chave é somente o nome, o sobrenome é ignorado.

As chaves são organizadas por ordem alfabética, ou seja, os nomes mais próximos de “A” ficam na esquerda e os mais próximos de “Z” ficam na direita

Árvore data de nascimento

Foi bem difícil no início saber qual tipo usar como chave, João pensou em fazer 3 ints (mês,dia,ano), depois Alison sugeriu string, e no final João pensou um long int que tivesse um peso diferente para ano,mês,dia representando o cálculo abaixo.

No começo o cálculo era ano * 1000000 mas o João percebeu que era um exagero pois ficava com vários zeros inúteis, até que testando ele percebeu que é possível deixar cada casa representando ano,mês e dia.

Na Árvore data de nascimento a entrada é feita pelo formato

MM/DD/AAAA

Para usar como chave, foi feito o seguinte cálculo

(ano * 10000 + mês * 100 + dia) pois assim o peso do ano é bem maior que o do mês e mesmo se o mês for 12(o máximo) ainda não afeta as casas do ano, o mesmo para o mês com o dia

Cálculo da chave data

MM/DD/AAAA

AAAAMMDD

Entrada-> 10/17/2000 = 2000*10000 + 10 * 100 + 17=20001017 <- Chave

Funcionamento das principais funções

● Funções da árvore avl

OBS: A descrição dessas funções estão presentes no arquivo avl.h

Todas as funções foram feitas usando templates. Algumas delas são apenas usada especificamente para cada uma das 3 árvores criadas (árvore nome, árvore CPF e árvore data de nascimento)

- *Função* clear(Node<Tkey> *no);

```
template<typename Tkey>
Node<Tkey>* avl<Tkey>::clear(Node<Tkey> *no) {
    if (no != nullptr) {
        no->left = clear(no->left);
        no->right = clear(no->right);
        delete no;
    }
    return nullptr;
}
```

Complexidade: $O(n)$ pois todos os nós da árvore serão acessados

A função clear percorre a árvore em sentido pré-ordem, deletando todos os nós no qual ele visita e, por fim, ele retorna `nullptr` para o Node raiz criado na main.cpp, assim limpando por completo a árvore.

- *Função* create_node(Tkey key, Pessoa pes);

```
template<typename Tkey>
Node<Tkey>* avl<Tkey>::create_node(Tkey key, Pessoa pes) {
    Node<Tkey> *node = new Node<Tkey>;
    node->key = key;
    node->pes = pes;
    node->left = nullptr;
    node->right = nullptr;
    node->height = 1;
    return node;
}
```

Complexidade: $O(1)$ apenas atribuições

Função na qual vai criar um novo nó para a árvore. Nela será contida uma chave e um objeto Pessoa que contém todos os dados de uma pessoa pega no arquivo.csv. Vai retornar esse novo nó.

- *Função* rightRotation(Node<Tkey> *no);

```
template<typename Tkey>
Node<Tkey>* avl<Tkey>::rightRotation(Node<Tkey> *no) {
```

```

Node<Tkey>* aux;
aux = no->left;
no->left = aux->right;
aux->right = no;

// ##### atualizará as alturas dos nós #####
no->height = std::max(avl_height(no->left), avl_height(no->right)) + 1;
aux->height = std::max(avl_height(aux->left), avl_height(aux->right)) + 1;

```

Complexidade: $O(1)$ apenas atribuições

Rotação à direita. Suponha que existe um nó desbalanceado b que tem um filho esquerdo x e o x tem um filho esquerdo y , deixando o fator de balanceamento de b ser igual a -2 . Para fazer o balanceamento, será preciso fazer uma rotação a direita no b



- *Função* `leftRotation(Node<Tkey> *no);`

```

template<typename Tkey>
Node<Tkey>* avl<Tkey>::leftRotation(Node<Tkey> *no) {
    Node<Tkey> *aux = no->right;
    no->right = aux->left;
    aux->left = no;
    // ##### atualizará as alturas dos nós #####
    no->height = 1 + max(avl_height(no->left),
                       avl_height(no->right));
    aux->height = 1 + max(avl_height(aux->left),
                       avl_height(aux->right));
    // ##### #####
    return aux; // nova raiz
}

```

Complexidade: $O(1)$ apenas atribuições

Rotação à esquerda. Suponha que existe um nó desbalanceado b que tem um filho direito x e o x tem um filho direito y, deixando o fator de balanceamento de b ser igual a 2. Para fazer o balanceamento, será preciso: fazer uma rotação a esquerda no b

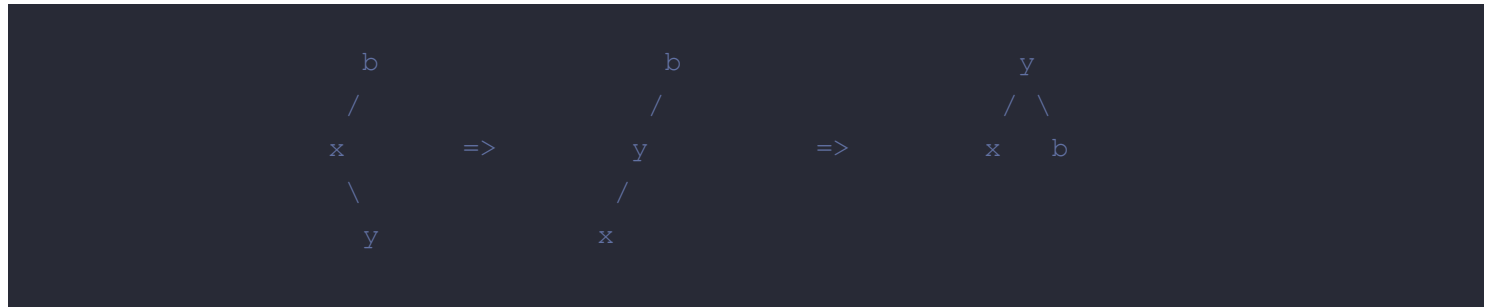


- *Função* rightLeft(Node<Tkey>* no);

```
//Rotação dupla a esquerda.
template<typename Tkey>
Node<Tkey>* avl<Tkey>::rightLeft(Node<Tkey>* no) {
    no->right = rightRotation(no->right);
    return leftRotation(no);
}
```

Complexidade:O(1) apenas atribuições

Rotação dupla à direita. Suponha que existe um nó desbalanceado b que tem um filho esquerdo x e o x tem um filho direito y, deixando o fator de balanceamento de b ser igual a -2. Para fazer o balanceamento, será preciso fazer uma rotação a esquerda no x e uma rotação a direita no b

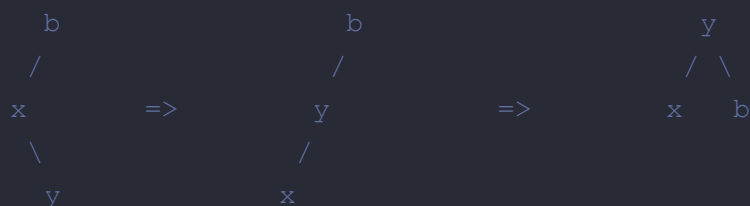


- *Função* leftRight(Node<Tkey>* no);

```
//Rotação dupla a direita.
template<typename Tkey>
Node<Tkey>* avl<Tkey>::leftRight(Node<Tkey>* no) {
    no->left = leftRotation(no->left);
    return rightRotation(no);
}
```

Complexidade: $O(1)$ apenas atribuições

Rotação dupla à direita. Suponha que existe um nó desbalanceado b que tem um filho esquerdo x e o x tem um filho direito y, deixando o fator de balanceamento de b ser igual a -2. Para fazer o balanceamento, será preciso: fazer uma rotação a esquerda no x e uma rotação a direita no b



- Função `balancing_factor (Node<Tkey> *node);`

```
template<typename Tkey>
int avl<Tkey>::balancing_factor (Node<Tkey> *node) {
    if(node == nullptr) {
        return 0;
    }
    return avl_height (node -> right) - avl_height (node -> left);
}
```

Complexidade: $O(1)$ Apenas um teste e retorno

Fator de balanceamento, onde o seu resultado é fazer subtração da altura do nó direito com a altura do nó esquerdo. O fator de balanceamento tem que ser somente -1, 0 ou 1. Caso o fator de balanceamento não tenha nenhum desses 3 resultados, significa que o nó em questão está desbalanceado. Ele é usado na função `avl_balance`.

- *Função* `avl<Tkey>::avl_balance(Node<Tkey> *no, Tkey key);`

```
template<typename Tkey>
Node<Tkey>* avl<Tkey>::avl_balance(Node<Tkey> *no, Tkey key) {
    if(balancing_factor(no) < -1 && key < no->left->key)
        return rightRotation(no);

    else if(balancing_factor(no) < -1 && key >= no->left->key)
        return leftRight(no);

    else if(balancing_factor(no) > 1 && key >= no->right->key )
        return leftRotation(no);

    else if(balancing_factor(no) > 1 && key < no->right->key)
        return rightLeft(no);

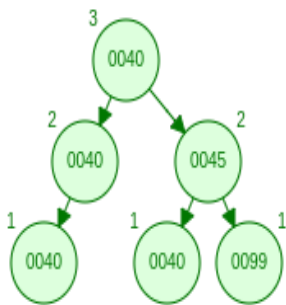
    return no;
}
```

Complexidade: $O(\log n)$ no pior caso vai da raiz até uma folha

A função `avl_balance` é responsável em fazer o balanceamento da árvore `avl`. No entanto, o problema que o projeto descreve é que existe nomes e datas de nascimento que são iguais, e isso é um problema já que vai existir duas árvores onde suas chaves serão nomes e data de nascimento, respectivamente. Essa situação não ocorre quando a chave é um CPF, pois o CPF são exclusivos unicamente para cada pessoal. Como a construção da `avl` que aprendemos não permitia chaves repetidas, precisamos modificar o algoritmo de inserção da árvore para ela aceitar chaves repetidas. Primeiro, consideramos inserir, as chaves que são repetidas, mais a direita (\geq) para que não houvesse conflito na hora de verificar caso fossem duas verificações, no caso “ \leq ” e “ \geq ”. O código referente a inserção descrita acima está mais abaixo.

Agora em relação a `avl_balance`, foi observado pelo Alysson que havia um padrão de rotações nas inserções de chaves repetidas na árvore. Foi observado que, as únicas rotações feitas para a fazer o balanceamento da árvore após a inserção da chave repetida foram a rotação **dupla à direita** e a **rotação à esquerda**. Esse padrão de rotações foi observado no site [AVL Tree Visualization](#).

Mas antes de colocar no código, fizemos simulações de inserir chaves repetidas em desenhos e no código do programa, testando todos os possíveis casos de rotações para manter a árvore balanceada, inclusive houve uma em específico que chamou a nossa atenção, onde ocorreu uma rotação dupla à direita e um dos nós que estavam na sub árvore esquerda da raiz, acabou indo parar na sub árvore direita da raiz da árvore, mas não acarretou problemas no balanceamento da árvore e nem na impressão da ordem simétrica da árvore. Mais a frente, isso interferiria nas funções de buscas na árvore, já que as chaves iguais poderiam estar tanto na subárvore esquerda quanto na direita. Exemplo :



- *Função* `avl_imprime_csv(Node<Tkey> *node, Tkey key);`

```
template<typename Tkey>
void avl<Tkey>::avl_imprime_csv(Node<Tkey> *node, Tkey key) {

    if(node == NULL) {                                     //retorna NULL se não encontrar
        cout << "O seguinte CPF não foi encontrado: " << key << endl;
        return;
    }

    if(key < node->key) //chama recursivamente para esquerda se key for menor que key da
chave esquerda
        avl_imprime_csv(node->left, key);

    else if(key > node->key)
        avl_imprime_csv(node->right, key); //chama recursivamente para direita se key
for maior que key da chave direita
```

```

else{
    node->pes.imprime_csv();          //Imprime todos os dados ao encontrar

}
}

```

Complexidade: $O(\log n)$ no pior caso vai da raiz até uma folha.

Percorre toda a árvore em ordem crescente, imprimindo os seus dados. Essa função de busca foi feita especialmente para a árvore com chaves sendo CPF, pois como o mesmo é único para cada pessoa, não possibilita de duas ou mais pessoas terem o mesmo CPF. Além dessa foi feita mais duas de buscas diferentes, sendo uma para a árvore com chaves sendo o nome e árvores com chave sendo a data de nascimento, onde essas tem ocorrência de existirem mais de um nome e data iguais.

- *Função* `avl_height(Node<Tkey> *no);`

```

template<typename Tkey>
int avl<Tkey>::avl_height(Node<Tkey> *no) {
    if (no == NULL)
        return 0;
    else
        return no->height;
}

```

Complexidade: $O(1)$ apenas um teste e um retorno

Retorna a altura do nó colocado no parâmetro da função. Retorna altura 0 caso a árvore esteja vazia.

- *Função* `avl_intervalo(Node<Tkey> *no, Tkey key1, Tkey key2);`

```

template<typename Tkey> //Percorre toda a arvore e imprime o intervalo entre key1 e key2
Node<Tkey>* avl<Tkey>::avl_intervalo(Node<Tkey> *no, Tkey key1, Tkey key2){ //imprime nós no intervalo

    long int diferenca = abs(key2 - key1); //diferença entre key1 e key2

    if(no != NULL){ //se a chave não for nula

```

```

    avl_intervalo(no ->left, key1, key2);
    if(key2 > key1 && no->key >= key2 - diferenca && no->key <= key2) //se key 2 for
maior que key 1
        no->pes.imprime_csv();
    else if(key1 > key2 && no->key >= key1 - diferenca && no->key <= key1) //se key 1
for maior que key 2
        no->pes.imprime_csv();
    avl_intervalo(no ->right, key1, key2);
}

return NULL;
}

```

Complexidade: $O(n)$ pois acessa toda a árvore

Essa função foi a última a ser criada, foi notado que seria necessário para árvore data. Ele percorre toda a árvore no percurso em ordem analisando as chaves dos nós que estão entre key1 e key2, para isso foi feito o módulo da diferença entre essas duas chaves dos parâmetro é analisado se a chave do nó está entre a maior key e a maior key subtraído pela diferença.

Essa função aceita tanto o key1 ser maior que o key2 quanto o key2 ser maior que o key1.

Função search_repetido(Node<Tkey>* no, Tkey key);

Como existem pessoas que podem ter nomes repetidos, a busca de todas as pessoas que possuem tais nomes iguais nas chaves dos nós vai significar em que elas podem está tanto na subárvore esquerda quanto na subárvore direita da raiz devido às rotações feitas para deixar a árvore balanceada. Nesse caso precisaremos buscar em toda a árvore para pegar todos os dados de uma ou mais pessoa(s) pelo seu nome . O motivo de fazermos isso é devido a testes nos quais fizemos onde inserimos algumas chaves repetidas menores que a raiz em uma árvore avl, e as rotações para manter a árvore

balanceada chegou a mandar chaves repetidas da subárvore esquerda da raiz para a subárvore direita da raiz, mas mantendo o seu balanceamento. Para analisar melhor essa situação, fizemos a visualização da árvore em pré-ordem, pós-ordem e em in-ordem(ordem simétrica) e vimos que as três estavam mostrando as chaves dos nós de forma corretamente, mas como não sabemos se todos determinados nós estão em um lado apenas na árvore, então precisaremos percorrer ela por inteira para achar as que são repetidas.

```
template<typename Tkey>
void avl<Tkey>::search_repetido(Node<Tkey>* no, Tkey key) {
    if (no != nullptr) {
        if (no->key == key) {
            cout << "Chave " << no->key << endl;
            //no->pes.imprime_csv();
            no->pes.imprime_csv();

            search_repetido(no ->left, key);
            search_repetido(no ->right, key);
        }
        else{
            search_repetido(no ->left, key);
            search_repetido(no ->right, key);
        }
    }
}
```

Complexidade: $O(n)$ todos os nós são acessados

→ Funções da Classe pessoa

- Função Pessoa(string cpf, string nome, string sobrenome, string dataNascimento, string cidade_nasc);

```
Pessoa::Pessoa(string cpf, string nome, string sobrenome, string dataNascimento, string
cidade_nasc) {

    this->cpf = cpf;
```

```

    this->nome = nome;
    this->sobrenome = sobrenome;
    this->dataNascimento = dataNascimento;
    this->cidade_nasc = cidade_nasc;
}

```

Complexidade: O(1) apenas atribuições

Construtor da classe pessoa, insere os dados do parâmetro na pessoa.

- Função `imprime_csv()`;

```

void Pessoa::imprime_csv() {

    cout << cpf << ","
         << nome << ","
         << sobrenome << ","
         << dataNascimento << ","
         << cidade_nasc << endl;

}

```

Complexidade: O(1) apenas impressões

Imprime todos os dados da pessoa separado por vírgula

- Função `to_long_int(string cpf)`;

```

long int Pessoa::to_long_int(string cpf) {

    int index = 0;
    string nova;

    for(int i = 0; i < 14; i++){
        if (i == 3 || i == 7 || i == 11)
            continue;
        else{
            nova[index] = cpf[i];
            index++;
        }
    }

    return stol( nova );

}

```

Complexidade: $O(1)$ apenas retorno e atribuições, sempre o laço vai ocorrer constantemente 14 vezes

Recebe uma string cpf e retorna um long int sem ponto e traço, para isso é criada uma string auxiliar nova e um inteiro auxiliar index, todo o vetor é percorrido ignorando a posição 3, 7 e 11 do vetor, pois tem ponto ou traço nessas posições

- Função `converte_dia(string x);`

```
long int Pessoa::converte_dia(string x){

    char nova[2];

    if(x[1] == '/' && x[3] == '/'){    //mes e dia com 1 digito
        nova[0] = x[2];
    }
    else if(x[1] == '/' && x[4] == '/'){    //dia com 1 digito e mes com 2
        nova[0] = x[2];
        nova[1] = x[3];
    }
    else if(x[2] == '/' && x[4] == '/'){    //dia com 2 digitos e mes com 1
        nova[0] = x[3];
    }
    else{                                //dia e mes com 2 digitos
        nova[0] = x[3];
        nova[1] = x[4];
    }

    if(x.size() <= 10 && x.size() >= 8)
        return stol( nova);
    else
        return -1;    //indica erro na quantidade de digitos
}
```

Complexidade: $O(1)$ apenas atribuições e testes

Essa função recebe uma string data com barras separando mês, dia e ano e retorna um inteiro só com o dia como inteiro, para isso ele testa todos os possíveis casos da quantidade de dígitos do dia e mês analisando a posição da barra, depois insere na string auxiliar nova e converte para inteiro.

É compatível com os formatos:

M/D/AAAA - M/DD/AAAA - MM/D/AAAA - MM/DD/AAAA

Caso tenha tamanho fora do padrão ele retorna -1

- Função `converte_mes(string x);`

```
long int Pessoa::converte_mes(string x){
    char nova[2];

    if(x[1] == '/')           //se o segundo caractere for barra
        nova[0] = x[0];
    else{
        nova[0] = x[0];
        nova[1] = x[1];
    }

    if(x.size() <= 10 && x.size() >= 8)
        return stol( nova);
    else
        return -1;           //indica erro na quantidade de digitos
}
```

Complexidade:O(1) apenas atribuições e testes

Essa função recebe uma string data com barras separando mês, dia e ano e retorna um inteiro só com mês como inteiro para isso ele analisa se a barra está na segunda posição do vetor, se estiver, só tem um número no mês, se não tiver tem 2 números, esses números são colocados no vetor auxiliar nova e retornados como inteiros

Caso tenha tamanho fora do padrão ele retorna -1

É compatível com os formatos:

M/D/AAAA - M/DD/AAAA - MM/D/AAAA - MM/DD/AAAA

- Função `converte_ano(string x);`

```
long int Pessoa::converte_ano(string x){
    char nova[4];

    if(x.size() == 10){           //data com tamanho 10
        nova[0] = x[6];
        nova[1] = x[7];
        nova[2] = x[8];
        nova[3] = x[9];
    }
}
```



```

        else if(x.size() == 9){           //data com tamanho 9
            nova[0] = x[5];
            nova[1] = x[6];
            nova[2] = x[7];
            nova[3] = x[8];
        }
        else{
            nova[0] = x[4];               //data com tamanho 8
            nova[1] = x[5];
            nova[2] = x[6];
            nova[3] = x[7];
        }

        if(x.size() <= 10 && x.size() >= 8)
            return stol( nova);
        else
            return -1; //indica erro na quantidade de digitos
    }

```

Complexidade: O(1) apenas atribuições e testes

Essa função recebe uma string data com barras separando mes, dia e ano, tem o objetivo de retornar só o ano como inteiro, para isso ele analisa o tamanho da string inteira para saber a quantidade de dígitos do mês e dia, após isso ele insere na string auxiliar nova e retorna como inteiro.

Caso tenha tamanho fora do padrão ele retorna -1.

É compatível com os formatos:

M/D/AAAA - M/DD/AAAA - MM/D/AAAA - MM/DD/AAAA

- Funções gets;

```

//gets
string Pessoa::get_cpf() {
    return cpf;
}

string Pessoa::get_nome() {
    return nome;
}

string Pessoa::get_sobrenome() {
    return sobrenome;
}

```

```
string Pessoa::get_cidade_nasc() {
    return cidade_nasc;
}
```

Complexidade: O(1) apenas retornos

Para obter dados privados da classe pessoa

- Funções sets;

```
//sets
void Pessoa::set_cpf(string cpf) {this->cpf = cpf;}
void Pessoa::set_nome(string nome) {this->nome = nome;}
void Pessoa::set_sobrenome(string sobrenome) {this->sobrenome = sobrenome;}
void Pessoa::set_cidade_nasc(string cidade_nasc) {this->cidade_nasc = cidade_nasc;}
void Pessoa::set_dataNascimento(string data) {this->dataNascimento = data;}
```

Complexidade: O(1) apenas atribuições

Para modificar dados privados da classe pessoa

→ Classe Node

```
template<typename Tkey>
class Node{
public:
    Tkey key;
    Pessoa pes;
    int height;
    Node<Tkey>* left;
    Node<Tkey>* right;
};
```



Usamos a classe Node para usar seus atributos e criar nós da árvore avl. Criamos 3 tipos de Node, no caso 3 raízes, para atender as características das 3 árvores e serem suas raízes.

Conclusão:

O trabalho começou sendo feito separadamente nos primeiros dias, depois a dupla se juntou para fazer tudo através do live share. A organização de quem iria fazer tal coisa foi feita em uma página no Notion, onde pode ser acessado por neste link [Projeto 1- EDA](#) ou no link presente na bibliografia, na qual fica mais organizado tanto visualmente, quanto pela facilidade de estruturar um planejamento mais adequado e ver o andamento das implementações das coisas do programa.

Tivemos dificuldades em algumas partes do projeto, são elas:

- Separar dado por dado de uma linha do arquivo data.csv. já que era separado por uma vírgula cada um;
- Como a gente ia pegar o CPF e transformá-lo em um long int, para colocar na chave do nó;
- Pegar a data de nascimento de cada pessoa e converter ela para long int e colocar nas chaves dos nós da árvore data;
- Tratar as chaves que são repetidas nas Árvores;
- Tratar os nomes com nomes que possuem acento;
- Problemas em inserir os dados das pessoas no objeto Pessoa de acordo com os nossos planejamentos;

Em geral, o projeto foi bem dividido para ambos. Na maioria do tempo, fizemos a grande das coisas juntos, usando o live share no VSCODE  e conversando via chat voice pelo aplicativo Discord . Sempre que tínhamos algum problema, nós dois nos ajudávamos para tentar resolver o mais eficiente possível.

Bibliografia

Foi utilizado o Notion para organizar o projeto

<https://www.notion.so/Projeto-1-EDA-697f5290a7f34f99b7a9af9b17548a86>

<https://stackoverflow.com/questions/7663709/how-can-i-convert-a-string-to-int>

<https://www.inf.pucrs.br/~pinho/CPP/SobreCargaDeOperadores/SobreCargaOperadores.html#:~:text=A%20sobrecarga%20de%20um%20operador,%22%20a%20fun%C3%A7%C3%A3o%20seria%20operador%3E.>

<http://www.cplusplus.com/reference/string/stol/>

<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

<https://www.ascii-codes.com/cp860.html>

<http://www.cplusplus.com/reference/sstream/stringstream/>