# AVL

**Alunos:**
**G**uilherme Willian Saraiva da Hora - 475980
**S**amuel Evangelista de Aquino Júnior - 485374

## Sumário

# 1 Introdução

## 1.1 Problemática

O problema a ser resolvido, é uma aplicação com a capacidade de carregar um arquivo no formato csv e armazena esses dados na memória principal. A formatação dos dados será encontrada na seguinte forma:

- CPF (string)
  - 388.624.732-57
- Nome (string)
  - Tiago
- Sobrenome (string)
  - Cunha
- Data de Nascimento (MM/DD/AAAA:string)
  - 7/19/1989
- Cidade (string)
  - Rio Branco

A respectiva ordem acima é de como será encontrado o arquivo (.csv), os campos estarão separados por vírgula (,).

Os dados devem ser carregados dentro de objetos Pessoa, após o devido armazenamento dos dados, deverá ser possível realizar algumas consultas sobre os dados/atributos apresentados acima.

- Consultar uma única pessoa pelo seu CPF e exebir todos os seus dados em tela.
- Consultar todas as pessoas cujo nome cujo nome comece com uma string informada pelo usuário e exibir todos os dados em tela, em formato de lista.
- Consultar todas as pessoas cujo a data de nascimento esteja em um intervalo estabelecido pelo usuário e exibir todos os dados em tela, em formato de lista.

O problema princípal se da pelos problemas citados acima, pórem algumas específicações da aplicação trazem consigo alguns problemas secundários, como:

- Os objetos Pessoa armazenados deveram ser indexados pelos dados/atributos citados acima(CPF, nome, data de nascimento), afim de tornar operações como: inserção, remoção e busca, operações rápidas;
- Deverá ser utilizada a estrutura árvore de busca binária balanceada AVL, onde para cada dado/atributo indexado deverá ser criada uma árvore binária de busca AVL.
- Os objetos Pessoa armazenados, não podem ser duplicados na memória. A indexação deve ser realizada apenas com uma chave, dado/atributo citado anteriormente, e o endereço do respectivo registro.

A implementação deve ser realizada em C++ e utilizado o paradigma orientado a objetos.

### 1.2 A aplicação

Explanando de forma sucinta, a aplicação consiste em um sistema para armazenamento de dados de pessoas, e com uma funcionalidade de busca eficiente, que no caso é implementado a partir de uma árvore de busca binária balanceada AVL. Ao usuário será disponibilizado uma interface via Terminal para a realização de consultas, e a respectiva visualização dos dados de retorno da pesquisa. Ao iniciar o programa o usuário vai visualizar um menu com as consultas que podem ser realizadas, que são: CPF, Nome, Data. Após selecionar uma consulta será solicitado o dado de entrada para a realização da pesquisa, em seguida será mostrado em tela o resultado da pesquisa realizada.

## 2 Implementação

### 2.1 Detalhamento

Para a implementação foi utilizado uma árvore de busca binária balanceada, AVL, com isso conseguimos garantir uma eficiência bem satisfatória para o problema. A AVL é uma arvore de busca balanceada onde toda inserção e remoção, que são operações que podem alterar a eficiência da árvore são verificadas e caso seja necessário concertadas para continuar de modo coerente. Todo nó tem um campo altura que serve para calcular o balanceamento dos nós. O balanceamento é o calculo efetuado para manter o custo de acesso (e portanto a altura) na mesma ordem de grandeza de uma árvore ótima, isto é, $O(logn)$. Caso seja necessário, algumas rotações na árvore podem ser realizadas para manter a propriedade de árvore ótima, funções que utilizam tanto o balanceamento para verificar a desrregularidade e as rotações para concertas as desrregularidades são as chamadas $fixup$. Tanto ao final de uma inserção, quanto ao final de uma remoção essa função é chamada para verificar as propriedades e manter a árvore com propriedade de uma árvore ótima.

Foi incentivado a criação de uma árvore de uso genérico, para isso foi adotado a utilização de *templates* que é um recurso do C++ para que classes e funções operem de forma genérica. Isso permite com que uma classe ou função trabalhe com diversos tipos de dados sem ser reescrita. Logo para a solução do problema foi implementado um template encima do conteúdo da chave, para que a árvore podesse ser instanciada com mais de um tipo. Assim poderiamos tratar os dados de Nomes e CPFs como strings e a data como int, para facilitar a resolução.

### 2.2 Classes e Funções

Nesse tópico iremos explanar um pouco mais sobre as estruturas utilizadas, desde funções até as classes que compõem o projeto, como também todas as especificidades necessárias para a resolução do problema como um todo. Abaixo será detalhado todos aspectos importantes da implementação:

#### 2.2.1 Nó

Abaixo é detalhada a classe nó, que é um conjunto de elementos que armazenam informações. Tais nós compõem a estrutura da árvore que é formada pelos nós e suas respectivas conexões entre seus filhos. As especificidades do nó para um comum, é o template encima da chave para que a árvore tenha uso genérico. Após a compreensão completa do problema, foi identificado que poderia haver chaves duplicadas tanto para nomes quanto para datas, logo há a necessidade de tratar esse problema para não desbalancear a árvore. A solução que encontramos foi dentro de nó criar um vetor dinâmico que armazena os dados de pessoas caso tenham as chaves iguais, alocando assim espaço correto tanto para nós com chaves únicas ou duplicadas.

A classe também conta com um método que é responsável por comparar o seu atributo key com um valor passado com parâmetro. Essa comparação se dá pelo tipo do atributo key. Se key for do tipo int, será usada a ordem númerica. Se key for do tipo string, será utilizada a ordem lexicográfica. O método tem os seguintes valores de retorno:

- retorno = 0. Se os dois valores forem iguais;
- retorno = 1. Se o valor passado como parâmetro é maior;
- retorno = -1. Se o atributo key do nó é menor;

Como os atributos são privados utilizamos o recurso de friend class do C++, fazendo assim a classe nó uma friend class de AVLTree, logo na árvore todos os atributos da classe Nó vão estar visiveis para o acesso pela classe AVLTree.

```
template<typename K>
class Node {
private:
K key;
vector<Pessoa*> p;
  int height;
  Node *left;
  Node *right;
public:
    int compareKey( K k ) {
        if ( this->key.size() < k.size() ) {
            for ( int i=0 ; i < (int)this->key.size() ; i++ ) {
                if( this->key[i] < k[i] ) {
                    return (-1);
                }else if( this->key[i] > k[i] ) {
                    return 1;
                }
            }
        return -1;
        }
        if ( this->key.size() >= k.size() ) {
            for ( int i=0 ; i < (int)k.size() ; i++ ) {
                if( this->key[i] < k[i] ) {
                    return (-1);
                }else if( this->key[i] > k[i] ) {
                    return 1;
                }
            }
        return 0;
        }
}
    friend class AVLTree;
};
```

### 2.2.2 Pessoa

Classe responsável por armazenar as informações obtidas no arquivo de entrada do programa.

```
class Pessoa {
private:
    string Cpf;
    string Nome;
    string SNome;
    string Data;
    string Cidade;
public:
    Pessoa(string Cpf, string Nome, string SNome, string Data, string Cidade){
        this->Cpf = Cpf;
        this->Nome = Nome;
        this->SNome = SNome;
        this->Data = Data;
        this->Cidade = Cidade;
    }
};
```

### 2.2.3 AVLTree

A AVL tem todos os métodos similares a uma AVL comum, exceto pelos metodos detalhado abaixo por isso iremos enfatizar neles.

O primeiro método é o *avl_searchPrex*. Método esse que recebe um determinado valor, percorre a árvore e imprime os elementos que possuem o atributo key que comece o valor recebido da função. Se um determinado nó começa com esse determinado valor, é impresso os valores referentes aos objetos do tipo Pessoa contido nele e a função é chamada para ambos os filhos. Caso contrário, a função terá de ser chamada apenas para um de seus filhos, esquerdo ou direito. Essa escolha entre o filho esquerdo e direito, é definido de acordo com o tipo do atributo key. Se o atributo for do tipo *int*, será utilizado a ordem númerica. Se o atributo for do tipo *string*, será utilizado a ordem lexicográfica.

O segundo método é o *avl_searchKey*, esse método faz a busca de uma chave especifica, ou seja, quando informado o CPF de um usuário essa função consegue buscar os dados desse usuário de forma otimizada. Como o C++ consegue fazer comparação lexografica entre strings isso também funciona com os números dentro das strings, logo fazendo comparações de maior e menor conseguimos poldar sempre metade da árvore, caso a árvore esteja cheia. Uma busca simples de comparação indo para o lado correto da busca até encontrar a *key* ou verificar que a *key* não existe.

O terceiro método é o *avl_searchRange*, esse método faz a busca e traz todas as *keys* em um determinado range. Como parâmetro temos *range*1 e *range*2, todas chaves entre esse range, ou seja ($range1 < node < range2$) será necessário mostrar os dados em formato de lista. A função faz a busca na árvore por verificação de extremos, ou seja se $range1 > node$, logo saberemos que todos os valores estarão na parte direita de *node*. Caso o $range1 < node$, logo saberemos que os valores poderão estar tanto na parte direita quanto na esquerda de *node*, com isso também o *node* obdece o primeiro extremo, para verificar o segundo extremo no lado esquerdo temos que verificar se $range2 > node$, caso essa verificação seja obdecida os dois extremos são verificados e os dados de *node* podem ser impressos em tela, podendo haver mais valores a direita de *node* por isso chamamos recursivamente para lá. Após isso, chamamos recursivamente para a esquerda quando $range1 < node$.

A função *avl_insert* é uma insert comum tal como outros em árvores AVLs, entretanto como em geral AVLs não permitem chaves repetidas, quando encontrado chaves repetidas é retornado o Nó daquela chave. Na nossa implementação em vez de retorna o nó, colocamos os dados das chaves subsequentes

repetidas no vetor de pessoas que correspondem a mesma chave, assim mantemos o balanceamento, e conseguimos acessar os dados das pessoas, pois dentro de cada Nó tem a referencia dos dados do objeto Pessoa.

Dentro da *public* temos o constructor e o destructor, como também os métodos externos de cada um dos métodos citados acima, para que os usuários possam acessar.

```
template<typename K>
class   AVLTree {
private:
        Node<K> *root;
protected:
    Node<K> *avl_rightRotation(Node<K> *node);
    Node<K> *avl_leftRotation(Node<K> *node) ;
    Node<K> *avl_fixup_node(Node<K> *node, K key);
    Node<K> *avl_fixup_node_deletion(Node<K> *node);
    Node<K> *avl_delete_pred( Node<K> *root , Node<K> *node );
    int avl_height( Node<K> *node );
    int avl_balance( Node<K> *node );
    Node<K> *avl_clear(Node<K> *node);
    void avl_searchPrex ( Node<K>* node, K key ) {
        if ( node == nullptr ) {
            return;
        }else if ( node->compareKey(key) == 0 ) {
            avl_searchPrex ( node->left, key );
            for ( int i = 0; i < node->p.size() ; i++ ) {
                cout << node->p[i]->Cpf <<" "<< node->p[i]->Nome <<" "<< node->
                   p[i]->SNome <<" "<< node->p[i]->Data <<" "<< node->p[i]->
                   Cidade << endl;
            }
            avl_searchPrex ( node->right, key );
        }else if ( node->compareKey(key) > 0 ) {
            avl_searchPrex ( node->left, key );
        }else {
            avl_searchPrex ( node->right, key );
        }
    }
    void avl_searchKey( Node<K> *node, K key ) {
        if( node == nullptr ) {
            return nullptr;
        }
        if( key < node->key ) {
            return avl_searchKey(node->left, key);
        }else if( key > node->key ) {
            return avl_searchKey(node->right, key);
        }
        else {
            for ( int i = 0; i < node->p.size() ; i++ ) {
                cout << node->p[i]->Cpf << " " << node->p[i]->Nome << " " <<
                   node->p[i]->SNome << " " << node->p[i]->Data << " " << node
                   ->p[i]->Cidade << endl;
            }
            return node;
        }
    }
```

```
45      void avl_searchRange( Node<K> *node, K range1, K range2 ) {
46          if( node == nullptr ) {
47              return;
48          }
49          else if( range1 >= node->key ) {
50              return avl_searchRange( node->right, range1, range2 );
51          }
52          else {
53              if( node->key < range2 ) {
54                  for ( int i = 0; i < node->p.size() ; i++ ) {
55                      cout << node->p[i]->Cpf <<" "<< node->p[i]->Nome <<" "<<
                            node->p[i]->SNome<<" "<< node->p[i]->Data <<" "<< node
                            ->p[i]->Cidade << endl;
56                  }
57                  avl_searchRange( node->right, range1, range2 );
58              }
59              avl_searchRange( node->left, range1, range2 );
60          }
61      }
62      Node<K> *avl_insert(Node<K> *node, K key, Pessoa *p, bool &ans) {
63          if(node == nullptr) {
64              node = new Node<K>();
65              node->key = key;
66              node->p.push_back(p);
67              node->left = nullptr;
68              node->right = nullptr;
69              node->height = 1;
70              return node;
71          }
72          if(key < node->key) {
73              node->left = avl_insert(node->left, key, p, ans);
74          }else if(key > node->key) {
75              node->right = avl_insert(node->right, key, p, ans);
76          }else {
77              node->p.push_back(p);
78              return node;
79          }
80          node->height = 1+ max(avl_height(node->left), avl_height(node->right));
81          node = avl_fixup_node(node, key);
82          ans = true;
83          return node;
84      }
85      Node<K> *avl_delete( Node<K> *node, K key, bool &ans );
86  public:
87      AVLTree(void);
88      ~AVLTree(void);
89      void searchPrex ( K key ) {
90          if ( root == nullptr ) {
91              return;
92          }else if ( root->compareKey(key) == 0 ) {
93              avl_searchPrex ( root->left, key );
94              for ( int i = 0; i < root->p.size() ; i++ ) {
95                  cout << root->p[i]->Cpf <<" "<< root->p[i]->Nome <<" "<< root->
                        p[i]->SNome <<" "<< root->p[i]->Data <<" "<< root->p[i]->
                        Cidade << endl;
96              }
```

```
97              avl_searchPrex ( root->right, key );
98          }else if ( root->compareKey(key) > 0 ) {
99              avl_searchPrex ( root->left, key );
100         }else {
101             avl_searchPrex ( root->right, key );
102         }
103     }
104     bool searchKey(K key) {
105         Node<K> *aux = avl_searchKey(root, key);
106         if(aux != nullptr) {
107             cout << aux->p[0]->Nome << endl;
108             return true;
109         }
110         return false;
111     }
112     void searchRange(K range1, K range2) {
113         avl_searchRange(root, range1, range2);
114     }
115     bool insert(K key, Pessoa *p) {
116         bool ans = false;
117         root = avl_insert(root, key, p, ans);
118         return ans;
119     }
120     bool remove(K key);
121 };
```

### 2.2.4 split

Função que nos ajuda a tratar os registros lidos do arquivo de entrada. Recebe dois parâmetros, uma string e um caracter delimitador. A função irá seperar/quebrar a string recebida de acordo com o caracter delimitador. Cada "pedaço"da string será inserido em um $vector < string >$, e por fim, esse vector com as partes da sttring será retornado.

```
1  vector<string> split( const string& str, const string& delim ) {
2      vector<string> tokens;
3      size_t prev = 0, pos = 0;
4      do {
5          pos = str.find( delim, prev );
6          if ( pos == string::npos ) pos = str.length();
7          string token = str.substr(prev, pos-prev);
8          if ( !token.empty() ) tokens.push_back(token);
9          prev = pos + delim.length();
10     }
11     while ( pos < str.length() && prev < str.length() );
12     return tokens;
13 }
```

### 2.2.5 dataToInt

A função dataToInt é uma forma de transformar o formatado de data (MM/DD/AAAA) para o formato de somente (DD), ou seja, somente dias. Assim podemos tratar a data como um inteiro, sendo mais cômodo e eficiente a comparação entre as datas, pois na arvore de data temos que encontrar por ranges e a comparação por inteiro é bem mais rápido.

```
1 int dataToInt( string data ) {
2     int ans=0;
3     vector<string> ret;
4     ret = split( data, "/" );
5     if( ret.size() == 3 ) {
6         ans = stoi(ret[1]) + stoi(ret[0])*30 + stoi (ret[2])*365;
7     }
8     return ans;
9 }
```

#### 2.2.6 main

Função principal responsável por agrupar cada parte da solução e fornecer o programa para o usuário com todas as funcionalidades requisitadas. A main pode ser caracterizada nas seguintes partes:

- linha 2 - 8
  - Declaração das variáveis que seram utilizadas durante a execução;
  - Instanciação das três árvores binárias de busca balanceda AVL, onde cada árvore é instanciada com um dos atributos de Pessoa(CPF, Data de nascimento, Nome);
  - Criação de um vetor de Pessoas, que será responsável por armazenar os objetos do tipo Pessoa que seram criados durante a leitura dos registros do arquivo de entrada;
  - Criação de uma variável para manipulação do arquivo de entrada.
- linha 10
  - Definindo a path do arquivo de entrada.
- linha 12 - 20
  - Ler os registros do arquivo de entrada;
  - Chamar a função *split* para realizar o split da linha que foi lida do arquivo de entrada;
  - Criar o objeto do tipo Pessoa de acordo com os atributos contidos em *aux*, e adiciona-lo ao vector de pessoas.
- linha 22 - 26
  - Insere os objetos do tipo Pessoa em cada árvore criada anteriormente. É inserido o atributo do objeto no qual a árvore utiliza para indexação de seus elementos, e a referência para os objetos do tipo Pessoa contidos no vector.
- linha 28 - 84
  - Área onde as opções de busca do usuário são recebidas, e também onde são retornadas as suas respectivas respostas;
  - Há três tipos pesquisas disponíveis: busca de um determinado elemento pelo CPF, busca todos os elementos que começam com uma determinada string, busca todos os elementos que estão dentro de um intervalo, possuindo o valor minimo e máximo como entrada.

```
1 int main() {
2     int opc;
3     string cpf, str, data1, data2;
4     AVLTree<string> CPF;
```

```
5       AVLTree<int> DATA;
6       AVLTree<string> NOME;
7       vector<Pessoa> pessoas;
8       ifstream file;
9
10      file.open("../data.csv");
11
12      while(file.good()) {
13          string line;
14          vector<string> aux;
15          getline(file, line);
16          aux = split(line, ",");
17          if(aux.size() == 5) {
18              pessoas.push_back(Pessoa(aux[0], aux[1], aux[2], aux[3],aux[4]));
19          }
20      }
21
22      for(int i=0; i<10 ; i++) {
23          CPF.insert(pessoas[i].Cpf, &pessoas[i]);
24          DATA.insert(dataToInt(pessoas[i].Data), &pessoas[i]);
25          NOME.insert(pessoas[i].Nome, &pessoas[i]);
26      }
27
28      while(true) {
29          cout << "--------------------------------" << endl;
30          cout << "AVL AP2 implementation!" << endl;
31          cout << "--------------------------------" << endl;
32          cout << "1. Search a CPF" << endl;
33          cout << "2. Search a NOME" << endl;
34          cout << "3. Search a DATA" << endl;
35          cout << "4. Exit" << endl;
36          cout << "Get in with your choose: " << endl;
37
38          cin >> opc;
39
40          switch(opc) {
41              case 1:
42                  system("clear");
43                  cout << "Insert the CPF" << endl;
44                  cin >> cpf;
45                  CPF.searchKey(cpf);
46                  cout << "Wait 3 seconds for come back to the menu!!!" << endl;
47                  usleep(3000000);
48                  system("clear");
49                  break;
50
51              case 2:
52                  system("clear");
53                  cout << "Insert the string" << endl;
54                  cin >> str;
55                  NOME.searchPrex(str);
56                  NOME.getKeyRoot();
57                  cout << "Wait 3 seconds for come back to the menu!!!" << endl;
58                  usleep(3000000);
59                  system("clear");
60                  break;
```

```
61
62                  case 3:
63                  system("clear");
64                  cout << "Insert the First Data" << endl;
65                  cin >> data1;
66                  cout << "Insert the Second Data" << endl;
67                  cin >> data2;
68                  DATA.searchRange(dataToInt(data1), dataToInt(data2));
69                  cout << "Wait 3 seconds for come back to the menu!!!" << endl;
70                  usleep(3000000);
71                  system("clear");
72                  break;
73              case 4:
74                  system("clear");
75                  return 0;
76              default:
77                  system("clear");
78                  cout << "Wrong option, Program going to end" << endl;
79                  return 0;
80          }
81      }
82
83  return 0;
84  }
```

## 3 Análise de Complexidade

Aqui na análise iremos analisar a complexidade dos casos especificos que são as funções que foram alteradas. Sabemos que a complexidade de uma AVL implementada de forma correta é $O(logn)$, no nosso programa todas as funções continuam com essa complexidade com a exceção das que citaremos abaixo.

Na *avl_insert* a complexidade em relação ao tempo não se altera, pois só é adicionada uma atribuição de valores que é $O(1)$.

Na *avl_searchKey* a complexidade em tempo também não se altera, levando em consideração que é uma busca em meio a chaves únicas logo a complexidade continua como uma busca normal em AVL $O(logn)$.

Na *avl_searchRange* tratamos por meio de casos improvaveis, caso não tenha o mínimo de possibilidade de ter um valor a um determinado lado de um Nó, podamos aquele lado, logo eliminando a complexidade daquela busca se fosse realizada, uma complexidade exata para esses casos é impossível determinar, pórem a complexidade dessa função podemos determinar o pior caso que é quando o range abrange todo o conjunto de chaves logo sendo necessário buscar em toda árvore, logo a complexidade se torna $O(n)$ pois é necessário detalhar os dados de todos Nós.

Na *avl_searchPrex* o pior caso é desencadeado quando temos todos os nós da árvore, iniciando com a key que foi passada como parâmetro da função. Nesse caso, a pesquisa irá passar por todos os elementos da árvore visto que na busca, quando a atributo key de um nó é igual a key(parâmetro), essa busca deve ser repassada para ambos os seus filhos. Analisando somente esse pior caso, temos que a complexidade se torna $O(n)$. Esse pior caso analisado por nós, só acontece quando todos os nós começam o key(parâmetro). Em outros casos a busca irá sendo "podada", visto que essa busca é repassada somente para um dos filhos.

## 4 Conclusão

O trabalho proposto nos deu uma ideia real de uma aplicação que utiliza árvore binária de busca balanceada AVL para o seu funcionamento pleno e satisfatório. Há uma disponibilidade grande de implementações dessa determinada estrutura na internet, porém mediante as características da aplicação foi necessário o desenvolvimento de algumas soluções específicas.

A criação de um atributo, na classse Node, $vector < Pessoa* >$ nos possibilita resolver duas especificações do problema: os registros(objetos Pessoa) não deveram estar duplicados na memória e na árvore que utiliza o atributo Nome como indexação, poderá haver nomes repetidos. A ideia de criar esse vector de ponteiros para Pessoa nos possibilita armazenar somente a referência dos objetos Pessoa e pelo fato de ser um vector, estrutura em C++ que possibilita o armazenamento de elementos de um mesmo tipo, podemos guardar mais de uma referência para objetos do tipo Pessoa.

O tratamento da data a primorde foi uma "sacada", é bem mais intuitivo tratarmos range de valores por inteiros, em relação a comparação, do que tratar por strings. Logo veio uma dificuldade de achar alguma biblioteca que trabalhasse de forma semelhante ao timestamp em C++, logo depois pensamos em criar nosso próprio padrão de transformação de data, que foi transforma todos os campos para dias, assim teriamos uma comparação entre inteiros e de forma fiel ao correto, a formula se da por (DD + MM*30 + AAAA*365), levando em consideração os valores padrão para mês e ano em relação a dias.

## 5 Bibliografia

CORMEN, Thomas H. et al. Introduction to algorithms. MIT press, 2009.

AVL Tree Insertion, GeeksforGeeks, 2018. Disponível em: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de Dados e seus Algoritmos. Livros Tecnicos e Cientificos, 1994.