REO 6

Relatório de comparação

ABB x AVL

Rafael Porto Vieira de Moura - 201820274

Tales Ribeiro Magalhães - 201620433

09/02/2020

Lavras - MG

1. Introdução

O objetivo desta atividade é, baseado no número de acessos a um nó em uma determinada árvore, quantificar e comparar os resultados para um mesmo conjunto de dados aplicados a outra árvore.

As árvores utilizadas serão a ABB e a AVL. A principal diferença entre elas ocorre que na AVL, a estrutura é balanceada. Em outras palavras, a diferença da altura entre as sub árvores esquerda e direita é de no máximo 1 unidade.

A seguir, veremos o conjunto de dados utilizados para realizar as operações.

2. Conjunto de dados

- 100 números inteiros (esses serão utilizados nas 3 operações [busca, inserção, remoção] e deverão existir nas duas árvores):

659,191,618,956,437,950,16,23,703,636,83,376,331,58,780,597,666,504,476,234,771,975,151,415,707,572,310,458,568,740,687,92,681,360,375,41,842,741,33,923,28,756,798,831,676,628,370,776,383,596,38,297,416,283,910,717,898,277,8,2,266,961,529,584,439,187,352,915,454,913,704,398,494,767,35,68,235,513,250,79,889,127,369,97,626,845,577,163,908,721,774,607,,98,244,24,709,556,428,686,350,515

- 30 números inexistentes nas árvores, para realização da busca:

30,60,1003,994,1005,65,2,11,365,405,333,541,973,111,179,322,31,78,221,555,123,22,567,870,796,341,544,5,29,89

Os números foram separados apenas por vírgula pois estamos fazendo uso de um arquivo CSV para leitura e escrita. O que facilita na importação dos dados para o Excel, Planilhas ou outro software para manipulação de dados .

3. ABB

Para a construção da árvore binária, utilizamos implementação interativa para as três operações (busca, inserção, remoção).

```
void ArvoreABB::insere( Dado d ) {
    Noh* novo = new Noh( d );
    CONTADOR_NO_INSERCAO = 1;
    if ( mRaiz == NULL ) {
        mRaiz = novo;
    }
    else {
        Noh* atual = mRaiz;
        Noh* ant = NULL;

        while ( atual != NULL ) {
            ant = atual;
            if ( atual->valor > d ) {
                atual = atual->mEsquerdo;
            }
            else {
                atual = atual->mDireito;
            }
            CONTADOR_NO_INSERCAO++;
        }
        novo->mPai = ant;
        if ( ant->valor > novo->valor ) {
            ant->mEsquerdo = novo;
        }
        else {
            ant->mDireito = novo;
        }
    }
}
```

Inserção

Na imagem ao lado, além da implementação básica da inserção de um dado em nossa árvore, temos a adição de uma variável que funcionará como contador para o número de acessos aos nós. Inicializamos com o valor 1 pois partiremos da raiz e iremos considerar que, portanto, temos nosso primeiro acesso.

Faremos o incremento da variável, toda vez que nosso ponteiro não encontrar um nó nulo, ou seja, não realize a inserção e assim, temos certeza de que iremos incrementar a cada nó que acessarmos e a verificação de nulo não nos retornar ***true*.**

```
Noh* ArvoreABB::buscaAux( Noh* aux ) {
    Noh* atual = mRaiz;
    CONTADOR_NO_BUSCA = 0;
    while ( atual != NULL ) {
        CONTADOR_NO_BUSCA++;
        if ( atual->valor == aux->valor ) {
            return atual;
        }
        else if ( atual->valor > aux->valor ) {
            atual = atual->mEsquerdo;
        }
        else {
            atual = atual->mDireito;
        }
    }
    return atual;
}

void ArvoreABB::Busca( Dado d ) {
    Noh* aux = new Noh( d );
    Noh* nohComValor = buscaAux( aux );
    if ( nohComValor == NULL ) {
        cout << "NÃO ENCONTRADO\n";
    }
    else {
        cout << "ENCONTRADO\n";
    }
}
```

Busca

Se tratando de busca, temos a mesma lógica usada na inserção. Ou seja, iremos percorrendo nossa árvore em busca do nó folha (que aponta para NULL) e a cada verificação falsa, incrementamos nosso contador de busca.

Reparando neste método, notamos que a inserção e a busca produzirão o mesmo resultado. Isso porque os dois irão percorrer a árvore até um nó folha e após o procedimento irão retornar ou inserir um número de nossa árvore.

```
void ArvoreABB::transplanta( Noh* antigo, Noh* novo ) {
    if ( mRaiz == antigo ) {
        mRaiz = novo;
    }
    else if ( antigo == antigo->mPai->mEsquerdo ) {
        antigo->mPai->mEsquerdo = novo;
    }
    else {
        antigo->mPai->mDireito = novo;
    }
    if ( novo != NULL ) {
        novo->mPai = antigo->mPai;
    }
}

void ArvoreABB::remove( Dado d ) {
    Noh* aux = new Noh( d );
    Noh* remover = buscaAux( aux );
    CONTADOR_NO_REMOCAO = CONTADOR_NO_BUSCA;
    if ( remover == NULL ) {
        cout << "ERRO" << endl;
    }
    else {
        if ( remover->mEsquerdo == NULL ) {
            transplanta( remover, remover->mDireito );
        }
        else if ( remover->mDireito == NULL ) {
            transplanta( remover, remover->mEsquerdo );
        }
        else {
            Noh* sucessor = minimoAux( remover->mDireito );
            if ( sucessor->mPai != remover ) {
                transplanta( sucessor, sucessor->mDireito );
                sucessor->mDireito = remover->mDireito;
                sucessor->mDireito->mPai = sucessor;
            }
            transplanta( remover, sucessor );
            sucessor->mEsquerdo = remover->mEsquerdo;
            sucessor->mEsquerdo->mPai = sucessor;
        }
        delete remover;
    }
}
```

Remoção

Na remoção, as coisas mudam um pouco.

Para fazer a remoção de um elemento, a primeira coisa a se fazer é realizar a busca dele na árvore. Para facilitar, utilizamos o contador de busca para contar os acessos feitos até concluirmos a busca.

Se o nó a ser removido seja folha, nós simplesmente o apagamos da árvore. Caso contrário, faremos a troca de ponteiros para não perdermos o endereço de nossa árvore (ajustando apontamentos do pai e do(s) filho(s)).

4. AVL

Para a construção da árvore AVL, iremos usar métodos de inserção, remoção e busca recursivos. Tornando uma solução mais elegante ainda que em alguns casos possa se tornar menos eficiente.

Com isso, faz-se necessário que o programa armazene o fator de balanceamento que é dado pela subtração da altura da subárvore direita pela altura da subárvore esquerda.

```
void avl::insere( const Dado& umDado ) {
    CONTADOR_NO_INSERCAO = 0;
    isINSERCAO = true;
    raiz = insereAux( raiz, umDado );
    isINSERCAO = false;
}

noh* avl::insereAux( noh* umNoh, const Dado& umDado ) {
    noh* novo;
    novo = new noh( umDado );
    CONTADOR_NO_INSERCAO++;
    if ( umNoh == NULL ) {
        return novo;
    }
    else {
        if ( umDado < umNoh->elemento ) {
            umNoh->esq = insereAux( umNoh->esq, umDado );
        }
        else if ( umDado > umNoh->elemento ) {
            umNoh->dir = insereAux( umNoh->dir, umDado );
        }
        else {
            throw runtime_error( "Erro na insercao: chave repetida!" );
        }
    }
    return arrumaBalanceamento( umNoh );
}
```

Inserção

A inserção ocorre na comparação da chave com um nó existente na árvore. Primeiro fazemos a comparação com a raiz para sabermos se iremos para a subárvore esquerda (caso seja menor) ou subárvore direita (caso seja maior) e assim podemos continuar nossa busca até que encontremos uma posição para a inserção.

Ao final da inserção, chamamos a função de balanceamento que fará com que nossa AVL fique diferente da nossa ABB ao realizar rotações de nós que entrem em uma determinada condição. Isso permitirá que a árvore fique balanceada e reduza a altura da mesma ao ajustar os elementos em uma configuração que otimiza operações dentro da mesma. Nessas rotações ocorrem acessos aos nós e portanto, incrementos na nossa variável.

```
noh* avl::buscaAux( Dado chave ) {
    noh* busca;
    busca = raiz;
    CONTADOR_NO_BUSCA = 0;
    while ( busca != NULL ) {
        CONTADOR_NO_BUSCA++;
        if ( chave == busca->elemento ) {
            return busca;
        }
        else if ( chave < busca->elemento ) {
            busca = busca->esq;
        }
        else {
            busca = busca->dir;
        }
    }
    return busca;
}

Dado avl::busca( Dado chave ) {
    noh* resultado = buscaAux( chave );
    if ( resultado != NULL )
        return resultado->elemento;
    else
        throw runtime_error( "Erro na busca: elemento não encontrado!" );
}
```

Busca

Na busca, ao informarmos o dado a ser procurado, nosso programa irá percorrer toda a nossa árvore. Se o dado for menor que nossa raiz, a busca será feita na subárvore esquerda. Caso contrário, ela será feita na subárvore direita.

Se a busca pelo valor nao retornar ***true*** ao final da checagem das condições, incrementamos nosso contador pois assim, saberemos que o programa irá prosseguir para o próximo nó (contando mais um acesso).

```
void avl::remove( Dado chave ) {
    CONTADOR_NO_REMOCAO = 0;
    isREMOCAO = true;
    raiz = removeAux( raiz, chave );
    isREMOCAO = false;
}

noh* avl::removeAux( noh* umNoh, Dado chave ) {
    if ( umNoh == NULL ) {
        throw runtime_error( "Erro na remoção: elemento não encontrado!" );
    }
    noh* raizSubArvore;
    raizSubArvore = umNoh;
    CONTADOR_NO_REMOCAO++;
    if ( chave < umNoh->elemento ) {
        umNoh->esq = removeAux( umNoh->esq, chave );
    }
    else if ( chave > umNoh->elemento ) {
        umNoh->dir = removeAux( umNoh->dir, chave );
    }
    else {
        if ( umNoh->esq == NULL ) {
            raizSubArvore = umNoh->dir;
        }
        else if ( umNoh->dir == NULL ) {
            raizSubArvore = umNoh->esq;
        }
        else {
            raizSubArvore = encontraMenor( umNoh->dir );
            raizSubArvore->dir = removeMenor( umNoh->dir );
            raizSubArvore->esq = umNoh->esq;
        }
        delete umNoh;
    }
```

Remoção

Ao remover algum dado, o programa percorrerá nossa árvore em busca do elemento da mesma forma que na busca. Porém, diferente da ABB a remoção não pode ser feita diretamente pois é necessário checar se a árvore está balanceada depois de qualquer remoção (da mesma forma que ocorre na inserção).

Portanto, utilizamos da recursividade para percorrer toda a árvore e ao final da remoção, balanceamos se necessário.

```
noh* avl::rotacaoEsquerda( noh* umNoh ) {
    noh* aux;
    aux = umNoh->dir;
    umNoh->dir = aux->esq;
    if ( umNoh == raiz ) raiz = aux;
    aux->esq = umNoh;
    umNoh->atualizaAltura();
    aux->atualizaAltura();
    if (isINSERCAO) CONTADOR_NO_INSERCAO++;
    if (isREMOCAO) CONTADOR_NO_REMOCAO++;
    return aux;
}

noh* avl::rotacaoDireita( noh* umNoh ) {
    noh* aux;
    aux = umNoh->esq;
    umNoh->esq = aux->dir;
    if ( umNoh == raiz ) raiz = aux;
    aux->dir = umNoh;
    umNoh->atualizaAltura();
    aux->atualizaAltura();
    if ( isINSERCAO ) CONTADOR_NO_INSERCAO++;
    if ( isREMOCAO ) CONTADOR_NO_REMOCAO++;
    return aux;
}
```

Rotações

Até então os métodos de incrementos do contador de acessos da AVL eram muito parecidos com os da ABB. Mas o segredo da diferença está nas rotações.

Em toda e qualquer rotação feita em nossa árvore, é necessário que acessem outro nó. Com isso, usamos uma variável para verificar se estamos realizando inserção ou remoção dentro da rotação para podermos incrementar a variável correta.

Dessa forma, conseguimos controlar adequadamente o acesso aos nós de nossa AVL.

5. Estatísticas

Abaixo, estará a tabela completa com os dados de inserção, remoção e busca para os 100 números que existem nas nossas árvores.

E ao final dela, teremos algumas estatísticas produzidas com estes resultados que facilitarão nossa comparação.

Na tabelas temos o campo valor, que é o número que estamos inserindo;
Nos campos inserção, busca e remoção os valores apresentados representam o número de nós acessados para realizar a operação.

ABB				AVL			
Valor	Inserção	Busca	Remoção	Valor	Inserção	Busca	Remoção
659	1	1	1	659	1	6	6
191	2	2	2	191	2	6	6
618	3	3	3	618	5	1	1
956	2	2	2	956	3	7	6
437	4	4	4	437	3	3	5
950	3	3	2	950	6	5	4
16	3	3	3	16	3	6	4
23	4	4	4	23	4	5	7
703	4	4	4	703	4	3	3
636	4	4	3	636	4	5	4
83	5	5	2	83	6	3	2
376	5	5	5	376	4	4	4
331	6	6	6	331	6	2	2
58	6	6	5	58	5	6	6
780	5	5	5	780	7	6	6
597	5	5	5	597	5	7	6
666	5	5	1	666	4	4	5
504	6	6	6	504	8	4	4
476	7	7	7	476	8	6	8
234	7	7	2	234	7	7	6
771	6	6	6	771	5	2	3
975	3	3	3	975	5	7	7
151	6	6	4	151	5	5	2
415	6	6	5	415	6	5	4
707	7	7	7	707	7	5	5
572	7	7	7	572	6	5	4
310	8	8	7	310	6	7	6
458	8	8	4	458	6	5	3
568	8	8	8	568	8	7	7
740	8	8	8	740	8	4	4
687	6	6	1	687	6	6	3
92	7	7	6	92	7	7	7
681	7	7	5	681	9	5	5
360	7	7	7	360	6	5	5

375	8	8	7	375	9	7	6
41	7	7	6	41	8	5	6
842	6	6	6	842	5	3	4
741	9	9	9	741	7	6	8
33	8	8	8	33	8	4	4
923	7	7	7	923	7	7	6
28	9	9	8	28	9	6	7
756	10	10	6	756	6	5	5
798	7	7	3	798	6	4	5
831	8	8	3	831	8	5	2
676	8	8	1	676	9	6	6
628	5	5	5	628	8	6	6
370	9	9	9	370	9	6	6
776	7	7	7	776	8	5	6
383	7	7	7	383	6	6	7
596	8	8	6	596	7	6	4
38	9	9	8	38	6	6	7
297	9	9	5	297	9	6	4
416	7	7	5	416	6	6	5
283	10	10	9	283	7	7	7
910	8	8	8	910	9	4	3
717	9	9	9	717	6	6	6
898	9	9	9	898	8	6	6
277	11	11	2	277	10	5	5
872	10	10	10	872	7	6	6
266	12	12	8	266	9	7	5
961	4	4	3	961	9	6	5
529	9	9	6	529	7	6	5
584	9	9	5	584	9	7	5
439	9	9	4	439	8	6	3
187	7	7	5	187	6	7	5
352	8	8	8	352	7	6	6
915	9	9	3	915	7	6	5
454	10	10	10	454	7	7	7
913	10	10	7	913	9	7	3
704	8	8	3	704	6	6	4
398	8	8	8	398	7	7	7

494	8	8	8	494	7	7	7
767	11	11	8	767	8	6	7
35	10	10	5	35	7	7	3
68	7	7	7	68	7	7	7
235	13	13	2	235	10	4	4
513	10	10	4	513	9	7	5
250	14	14	2	250	10	6	4
759	12	12	6	759	7	7	6
889	11	11	6	889	10	5	5
127	8	8	7	127	9	7	6
369	10	10	10	369	7	7	7
97	9	9	5	97	10	6	2
626	6	6	3	626	7	7	2
845	11	11	1	845	9	7	2
577	10	10	5	577	8	8	6
163	8	8	1	163	8	8	1
908	10	10	10	908	7	7	7
721	10	10	10	721	7	7	7
774	8	8	4	774	8	6	3
607	6	6	5	607	8	8	6
98	10	10	10	98	8	8	8
244	15	15	2	244	9	7	4
24	10	10	5	24	7	7	4
709	10	10	1	709	7	7	2
556	10	10	2	556	8	8	1
428	8	8	7	428	7	7	6
686	8	8	8	686	7	7	7
350	9	9	5	350	9	7	2
515	11	11	10	515	8	8	9
MÉDIA	8	8	5	MÉDIA	7	6	5
DESVIO PADRÃO	2,586210519	2,586210519	2,578778977	DESVIO PADRÃO	1,817173963	1,409061583	1,78916492
MODA	8	8	5	MODA	7	7	6
MÍNIMO	1	1	1	MÍNIMO	1	1	1
MÁXIMO	15	15	10	MÁXIMO	10	8	9

Como podemos notar, a árvore AVL se saiu melhor em todas as operações feitas para este conjunto de dados. Os números foram gerados de forma aleatória mas o conjunto e a ordem de inserção, busca e remoção são as mesmas para ambas as estruturas.

A primeira grande diferença surge no desvio padrão. Desvio padrão é uma medida de dispersão, ou seja, é uma medida que indica o quanto o conjunto de dados é uniforme.

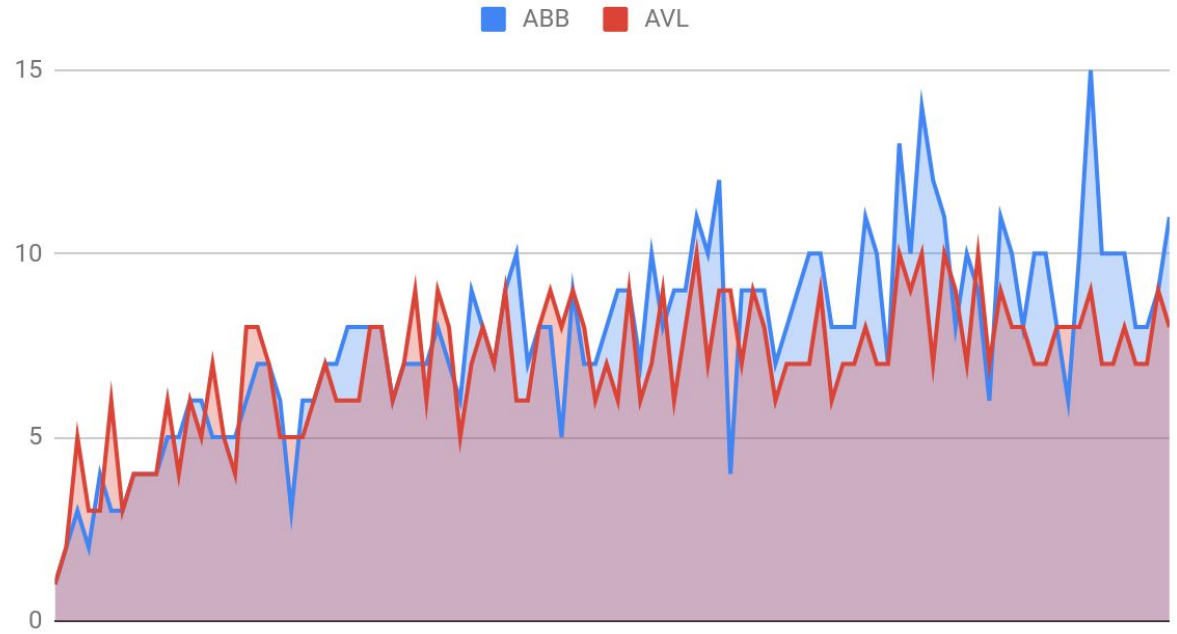
Quando o desvio é baixo quer dizer que os dados do conjunto estão mais próximos da média.

Portanto, podemos concluir que, na AVL, os dados estão organizados de maneira mais uniforme e balanceada. Permitindo que nossas operações tenham mais eficiência ao reduzir o número de acessos (caminho a ser percorrido até atingir o destino).

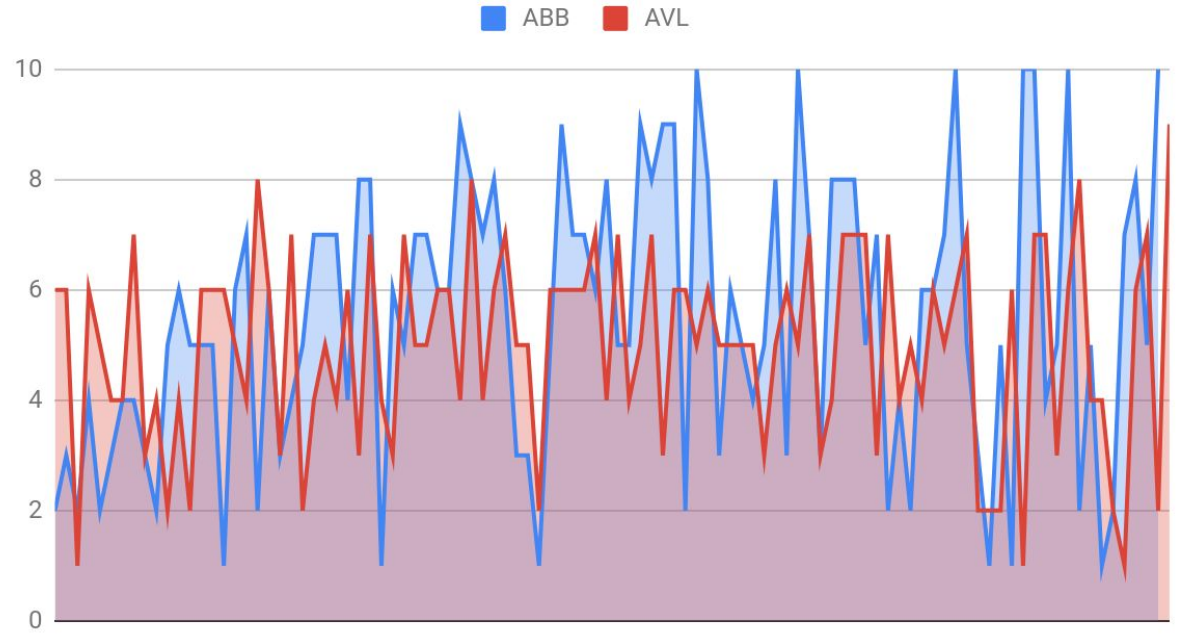
Outro dado que evidencia o que acabamos de constatar é a linha "Máximo" que nos permite ver o máximo de nós acessados para atingirmos o desejado na operação. Notamos que para a busca, temos uma redução de quase 50% de nós acessados. Para um conjunto de dados maior essa diferença é enorme e faz-se extremamente necessária a utilização de uma AVL.

Abaixo, temos alguns gráficos que representam uma comparação de ambas abordagens.

Nós acessados para realizar inserção



Nós acessados para realizar remoção

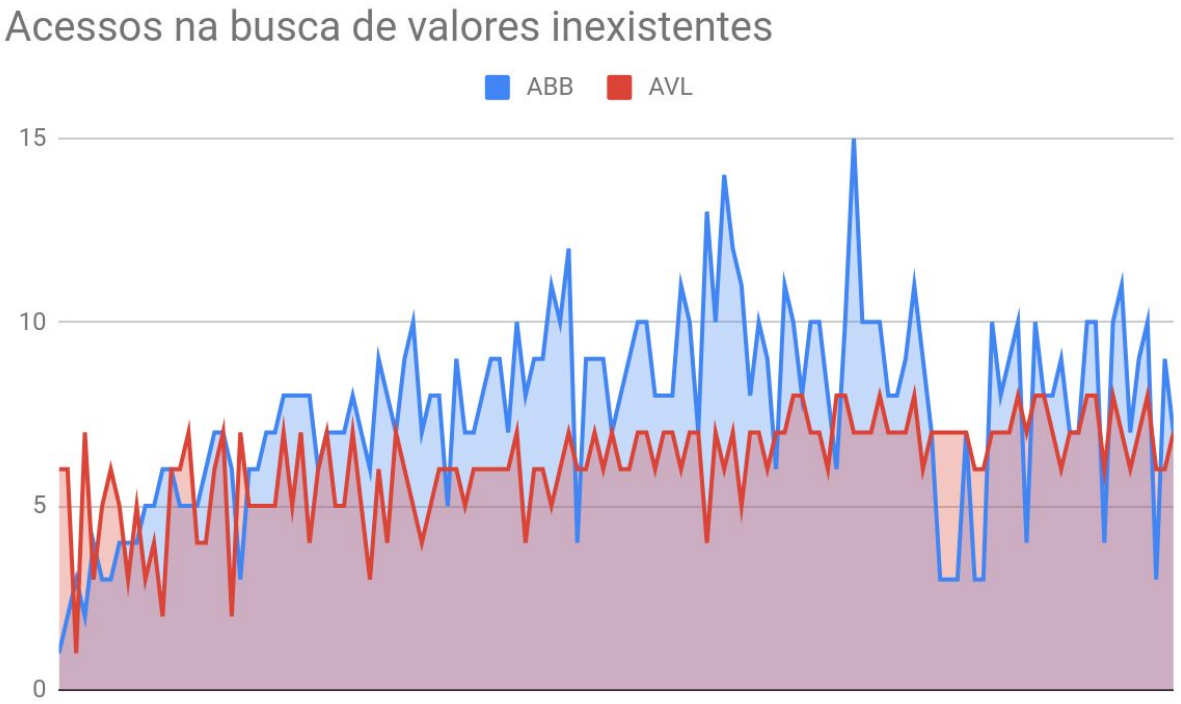


Agora, para separar melhor os dados, organizamos os dados de busca em um outro pedaço da tabela. Para podermos comparar melhor a busca de dados inexistentes.

ABB			AVL	
valor	busca		valor	busca
30	9		30	6
60	7		60	7
1003	3		1003	7
994	3		994	7
1005	3		1005	7
65	7		65	7
2	3		2	6
11	3		11	6
365	10		365	7
405	8		405	7
333	9		333	7
541	10		541	8
973	4		973	7
111	10		111	8
179	8		179	8
322	8		322	7
31	9		31	6
78	7		78	7
221	7		221	7
555	10		555	8
123	10		123	8
22	4		22	6
567	10		567	8
870	11		870	7
796	7		796	6
341	9		341	7
544	10		544	8
5	3		5	6
29	9		29	6
89	7		89	7
DESVIO PADRÃO	**2,715642216**		**DESVIO PADRÃO**	**0,7183954023**

Nesta tabela, que contém apenas o número de nós acessados para realizar a busca de um valor inexistente na árvore, podemos ver o quão grande é a diferença de uma busca feita em uma árvore AVL, e outra na ABB.

O gráfico a seguir nos mostra isso de forma bem clara:



Percebemos que em alguns casos a ABB consegue se sobressair em relação a AVL, mas de maneira geral, a AVL se mantém mais consistente ao se manter balanceada e possui uma organização de dados mais “homogenea”.