# Lista 1 de EDA 2

Arthur Luis Komatsu Aroeira
13/0102750

Pedro Kelvin de Castro M Batista
13/0129674

29 de agosto de 2017

## Perguntas e Respostas

1. **Implemente uma busca sequencial com índice primário. Explique seus algoritmos de inserção e de remoção.**

```
#include<bits/stdc++.h>

using namespace std;
#define ii pair<int, int>

//Busca sequencial com indice Primario
//Usando Vetor

/*
Tabela tera 30 elementos;
O numero de indices desejados serao 5;
30/5 = 6;
0+n/30
primeiro elemento = 0
segundo = (0+30)/5 = 6
terceiro = (0+30*2)/5 = 12
quarto = (0+30*3)/5 = 18
quinto = (0+30*4)/5 = 24
*/

vector<ii> tabela_indices;
int tabela[30] =
    ↪ {3,6,-1,-1,18,20,29,-1,38,-1,-1,59,66,-1,90,-1,129,-1,138,150,-1,-1,-1,210,218,-1,22
    ↪

//Ele busca o valor e retorna o valor do registro onde ele se encontra.
//Caso nao ache, retorna -1
int busca(int valor){

    for(int i = 0; i < tabela_indices.size(); ++i)
    {
        if(valor <= tabela_indices[i].first)
        {
            int novo_indice = tabela_indices[i-1].second;

            while(novo_indice < 30) //enquanto eu ainda varrer a tabela...
            {
                if(valor == tabela[novo_indice])
                    return novo_indice;
```

```
                novo_indice++;
            }
        }
    }

    return -1;
}

//Caso o elemento a ser removido seja valido, ele coloca -1
//em sua posicao, sinalizando que eh uma posicao valida
void remocao(int elemento){

    int indice = busca(elemento);

    if(indice == -1)
        cout << "Elemento nao existe\n";
    else
        tabela[indice] = -1;
}

void inserir(int elemento){

    bool flag = false;

    for(int i = 0; i < tabela_indices.size(); ++i)
    {
        if(elemento < tabela_indices[i].first)
        {
            int novo_indice = tabela_indices[i-1].second;

            while(novo_indice < 30 && elemento > tabela[novo_indice])
            {

                if(tabela[novo_indice] == -1)
                {
                    tabela[novo_indice] = elemento;
                    return;
                }
                novo_indice++;
            }

            cout << "Nao foi possivel achar espaco\n";
            return;
        }
    }
}

void imprime(){

    for(int i = 0; i < 30; ++i)
        cout << "valor: " << tabela[i] << " indice: " << i << endl;

}

int main(){

    tabela_indices.push_back({3,0});
    tabela_indices.push_back({29,6});
    tabela_indices.push_back({66,12});
```

```
    tabela_indices.push_back({138,18});
    tabela_indices.push_back({218,24});

    int num;

    while(cin >> num)
    {
        inserir(num);
        imprime();
    }

    return 0;
}
```

2. **Implemente as três travessias em uma árvore binária: Pré-ordem, Em-ordem e pós-ordem.**

   A seguir, o código implementado em C++. A entrada recebe o número de elementos a serem inseridos na árvore e depois os elemento:

```
#include <bits/stdc++.h>
using namespace std;

typedef struct Arv{
    struct Arv *right, *left;
    int value;
}Tree;

Tree *getTree(int value){

    Tree *New = (Tree *) malloc(sizeof(Tree));
    New->value = value;
    New->left = NULL;
    New->right = NULL;

    return New;
}

Tree *insertTree(Tree *node, int value){

    if(!node)
        return getTree(value);
    else if(value > node->value)
        node->right = insertTree(node->right,value);
    else
        node->left = insertTree(node->left,value);
}

void Visit(Tree *node)
{
    cout << node->value << " ";
}

void *preOrder(Tree *node){

    if(node){
        Visit(node);
        preOrder(node->left);
        preOrder(node->right);
    }
```

```
}

void *inOrder(Tree *node){

    if(node){
        inOrder(node->left);
        Visit(node);
        inOrder(node->right);
    }
}

void *postOrder(Tree *node){

    if(node){
        postOrder(node->left);
        postOrder(node->right);
        Visit(node);
    }
}

int main(){

    Tree *root = (Tree *) malloc(sizeof(Tree)); //Cria raiz
    root = NULL;

    int T, x;
    cin >> T; //Numero de elementos a serem inseridos na arvore

    while(T--)
    {
        cin >> x;
        root = insertTree(root,x); //Insere elementos na arvore
    }
    cout << "Pre Ordem: ";
    preOrder(root);
    cout << endl << "Em ordem: ";
    inOrder(root);
    cout << endl << "Pos ordem: ";
    postOrder(root);
    cout << endl;

    return 0;
}
```

Exemplo de input:

```
10
0 -5 1 6 7 25 15 14 6 5
```

Output:

```
Pre Ordem: 0 -5 1 6 6 5 7 25 15 14
Em ordem: -5 0 1 5 6 6 7 14 15 25
Pos ordem: -5 5 6 14 15 25 7 6 1 0
```

3. **Em uma busca por interpolação, utilize o cálculo dado em sala. Teste o tempo gasto pela busca para encontrar um valor em um vetor de 10, 25, 50, 100, 500, mil, dez mil, cem mil e um milhão de posições preenchidas com números randômicos. A taxa de crescimento é ou não é menor que a ordem de log(n)?**

O método escolhido para analisar a eficiência do algoritmo foi montar gráficos em escala logarítmica e comparar os algoritmos de Busca Binária com Busca por Interpolação. Logo, mudou-se os tamanhos sugeridos pelo enunciado para potências de 2 (tamanhos de 1, 2, 4, 8, 16, ..., 1048576). A seguir, a implementação do código.

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 20 //quantidade de tamanhos de vetores a serem testados

int it;

int BuscaInterpolacao(vector<int> v, int key)
{
        int left = 0, right = v.size() - 1;
        while(v[right] != v[left] && key >= v[left] && key <= v[right])
        {
                int mid = left + (long long)(right - left) * (key - v[left
                    ↪ ])/(v[right] - v[left]);

                if(v[mid] < key)
                        left = mid + 1;
                else if(v[mid] > key)
                        right = mid - 1;
                else
                        return mid;
                it++;

        }
        if(v[left] == key)
                return left;
        return -1;
}

int BinarySearch(vector<int> v, int key)
{
        int l = 0, r = v.size() - 1;
        while(l <= r)
        {
                int m = (l+r)/2;
                if(v[m] < key)
                        l = m + 1;
                else if(v[m] > key)
                        r = m - 1;
                else
                        return m;
                it++;
        }
        return -1;
}

int main()
{
        int tam[MAX+1];//vetor dos tamanhos com tam[x] = 2^x
        tam[0] = 1;
        for(int i = 1 ; i <= MAX ; i++)
                tam[i] = 2 * tam[i-1];

        for(int i = 0 ; i <= MAX ; i++)
        {
                vector<int> v; //vetor gerado aleatoriamente
```

```
                std::default_random_engine generator; //usando gerador de
                    ↪ numeros aleatorios
                std::uniform_int_distribution<int> distribution(0,tam[i]);
                    ↪  //distribuidos aleatoriamente
                for(int j = 0 ; j < tam[i] ; j++)
                        v.push_back(distribution(generator));
                sort(v.begin(), v.end()); //ordenar o vetor para comecar
                    ↪ as buscas

                it = 0; //zera as iteracoes para contar no loop

                auto begin = chrono::high_resolution_clock::now(); //
                    ↪ comeca o clock
                for(int j = 0 ; j < 100000 ; j++)
                        int x = BuscaInterpolacao(v, rand() % tam[i]); //
                            ↪ busca uma chave aleatoria
                auto end = chrono::high_resolution_clock::now();//termina
                    ↪ o clock
                cout << tam[i] << " elementos (Interpolacao): " << (long
                    ↪ double)chrono::duration_cast<chrono::nanoseconds>(
                    ↪ end-begin).count()/1000000000 << " s e " << it << "
                    ↪  iteracoes" << endl;

                it = 0;

                begin = chrono::high_resolution_clock::now();
                for(int j = 0 ; j < 100000 ; j++)
                        int x = BinarySearch(v, rand() % tam[i]);
                end = chrono::high_resolution_clock::now();
                cout << tam[i] << " elementos (Binaria):      " << (long
                    ↪ double)chrono::duration_cast<chrono::nanoseconds>(
                    ↪ end-begin).count()/1000000000 << " s e " << it << "
                    ↪  iteracoes" << endl << endl;
        }

        return 0;
}
```

Foram realizadas 100000 buscas com chaves aleatórias para cada tamanho diferente e calculou-se o tempo que levou no total e a quantidade de iterações realizadas pelos loops. O seguinte resultado foi encontrado:

```
1 elementos (Interpolacao): 0.00442928 s e 0 iteracoes
1 elementos (Binaria):      0.00394725 s e 0 iteracoes

2 elementos (Interpolacao): 0.00429747 s e 0 iteracoes
2 elementos (Binaria):      0.00451561 s e 99974 iteracoes

4 elementos (Interpolacao): 0.00514947 s e 25132 iteracoes
4 elementos (Binaria):      0.00503758 s e 125106 iteracoes

8 elementos (Interpolacao): 0.00624817 s e 62394 iteracoes
8 elementos (Binaria):      0.00465226 s e 212881 iteracoes

16 elementos (Interpolacao): 0.0065939 s e 75156 iteracoes
16 elementos (Binaria):      0.00545123 s e 274602 iteracoes

32 elementos (Interpolacao): 0.011023 s e 121527 iteracoes
32 elementos (Binaria):      0.0081567 s e 371902 iteracoes
```

```
64 elementos (Interpolacao): 0.0091975 s e 126402 iteracoes
64 elementos (Binaria):      0.00915517 s e 456432 iteracoes

128 elementos (Interpolacao): 0.0107872 s e 141767 iteracoes
128 elementos (Binaria):      0.00995709 s e 553700 iteracoes

256 elementos (Interpolacao): 0.0128653 s e 180620 iteracoes
256 elementos (Binaria):      0.0112414 s e 649938 iteracoes

512 elementos (Interpolacao): 0.0134913 s e 206346 iteracoes
512 elementos (Binaria):      0.0140185 s e 746611 iteracoes

1024 elementos (Interpolacao): 0.0159316 s e 253077 iteracoes
1024 elementos (Binaria):      0.0165069 s e 844755 iteracoes

2048 elementos (Interpolacao): 0.0240666 s e 245976 iteracoes
2048 elementos (Binaria):      0.0240465 s e 943195 iteracoes

4096 elementos (Interpolacao): 0.0437012 s e 254252 iteracoes
4096 elementos (Binaria):      0.0441585 s e 1043004 iteracoes

8192 elementos (Interpolacao): 0.092256 s e 259215 iteracoes
8192 elementos (Binaria):      0.0933891 s e 1142853 iteracoes

16384 elementos (Interpolacao): 0.177667 s e 272640 iteracoes
16384 elementos (Binaria):      0.185628 s e 1242818 iteracoes

32768 elementos (Interpolacao): 0.399593 s e 278089 iteracoes
32768 elementos (Binaria):      0.402742 s e 1342551 iteracoes

65536 elementos (Interpolacao): 0.852134 s e 299040 iteracoes
65536 elementos (Binaria):      0.854003 s e 1442179 iteracoes

131072 elementos (Interpolacao): 1.87867 s e 303298 iteracoes
131072 elementos (Binaria):      1.8935 s e 1542261 iteracoes

262144 elementos (Interpolacao): 3.80234 s e 320527 iteracoes
262144 elementos (Binaria):      3.80053 s e 1642408 iteracoes

524288 elementos (Interpolacao): 7.90726 s e 343473 iteracoes
524288 elementos (Binaria):      8.78097 s e 1742282 iteracoes

1048576 elementos (Interpolacao): 40.7713 s e 328877 iteracoes
1048576 elementos (Binaria):      39.229 s e 1841129 iteracoes
```

A seguir, os gráficos (tamanho x tempo) e (tamanho x iterações) em escala logarítmica:

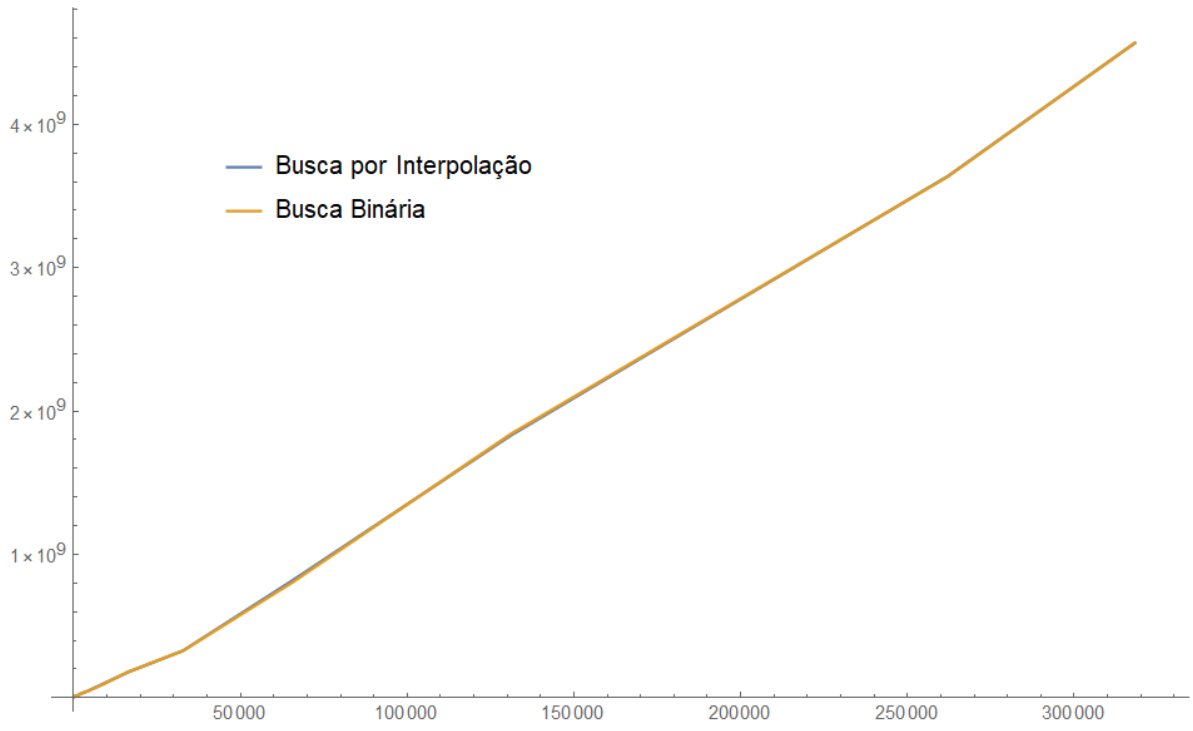


Figura 1 – Gráfico (tamanho x tempo)

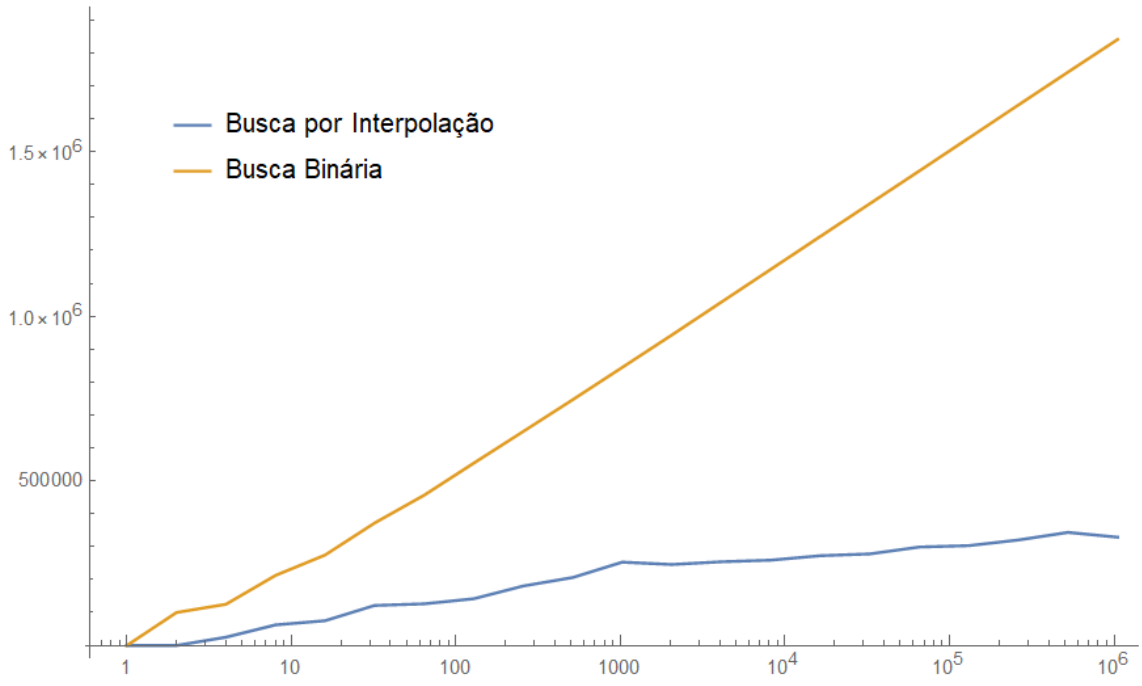


Figura 2 – Gráfico (tamanho x iterações)

Logo, percebe-se que a quantidade de iterações da busca por interpolação é muito menor em relação à binária, porém o tempo gasto entre os dois algoritmos é praticamente a mesma.

Como o número de iterações da busca binária apresentou aproximadamente uma reta e está plotada em um gráfico com escala logarítmica, conclui-se que a taxa de crescimento é $log(n)$. Como a curva da taxa de crescimento da busca por interpolação apresenta um aspecto côncavo, conclui-se que a taxa de crescimento é menor que $log(n)$.