Universidade Federal do Piauí – UFPI
Campus Senador Helvídio Nunes de Barros – CSHNB
Sistemas de Informação – Estruturas de Dados 2 – Prof. Juliana Oliveira

Relatório de trabalho prático sobre árvore binaria e AVL.

Aluno: Marcos Vinicius Batista Sampaio

Resumo:

O conteúdo descrito nesse relatório visa melhorar o entendimento sobre os códigos produzidos pelo autor para atender as atividades propostas e seus resultados, desde as metodologias utilizadas até a forma como o código foi implementado.

Introdução:

Para começar, é necessário entender o conceito de estruturas de dados, que é o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento. As estruturas de dados definem a organização. Métodos de acesso e opções de processamento para a informação manipulada pelo programa.

As estruturas de dados podem ser:

* lineares (ex. arrays) ou não lineares (ex. Grafos);

* homogêneas (todos os dados que compõe a estrutura são do mesmo tipo) ou heterogêneas (podem conter dados de vários tipos);

*estáticas (têm tamanho/capacidade de memória fixa) ou dinâmicas (podem expandir).

A primeira questão tratava da seguinte pergunta:

> Faça um programa em C que gere 1000 números aleatórios, insira-os em uma árvore binária de busca, inicialmente vazia. Os valores para compor a árvore devem ser gerados aleatoriamente utilizando 2 métodos para garantir a aleatoriedade. Depois imprima o nível da folha de maior profundidade e o nível da folha de menor profundidade. Repita o processo 30 vezes. Depois mostre quantas das 30 vezes a diferença entre a profundidade máxima e mínima foram de 0, 1, 2 , 3 e assim por diante. Também contabilize o tempo gasto para inserir todos os elementos na árvore. Além disso, para cada uma das árvores criadas faça experimento para verificar o tempo de busca de elementos. Você pode determinar uma sequência de elementos, em todas as árvores use sempre os mesmo elementos.

E a segunda questão pedia o mesmo, apenas transformando a arvore para o tipo AVL.

O trabalho a seguir consiste em mais duas seções, a especifica onde é apresentado a maneira como o trabalho foi feito, apresentando partes mais técnicas e em seguida a seção de resultados.

Seções Específicas:

No inicio do codigo já é declarado as bibliotecas necessárias para que o condigo funcionem sendo elas:

#include<stdio.h> = padrão para input e output
#include<stdlib.h> = biblioteca padrão que contem o recurso rand, utilizado para preencher
#include<time.h> = será necessaria para utilizar a função para capturar o tempo de insersão
#include<sys/time.h> = auxiliar da biblioteca time já descrita.

A seguir é criado a estrutura da árvore binaria, que para funcionar é necessário ter uma informação seguida das suas referências esquerda e direita para guardar seus próximos itens. Funciona de maneira que alocado na memória, as informações esquerda e direita sejam criadas do tipo árvore também.

```
struct arv{
        int num;
        struct arv *esq;
        struct arv *dir;
};
typedef struct arv Arvore;
```

Na próxima função, é feita a inicialização da arvore, que funciona retornando como NULL, já que se dada como Null, significa que não há uma arvore iniciada ainda e pode seguir com a inserção de um numero na raiz nula.

```
Arvore *inicializar(){
return NULL;
```

A função em seguida na ordem, trata-se da função inserir, que a arvore inicalizada uma folha como NULL visto acima, é feito a alocação de memoria para que possa ser criada e então como null, é feito com uma declaração de arvore auxiliar a inserçao da arvore, caso a raiz não seja nula ao inserir, ele entra em uma das condições a seguir para saber se é maior ou menor que o item na árvore e por fim retorna a raiz.

```
Arvore *inserir(Arvore *raiz, int num){
if(raiz == NULL){
        Arvore *aux = (Arvore*)malloc(sizeof(Arvore));
        aux->num = num;
        aux->esq = NULL;
        aux->dir = NULL;
        return aux;
}else{
        if(num < raiz->num){
                raiz->esq = inserir(raiz->esq, num);
        }else if(num>raiz->num){
                raiz->dir = inserir(raiz->dir,num);
        }
}
return raiz;
}
```

Temos a função para saber a maior profundidade, no geral sabe-se, que quando a raiz é nula, a profundidade se dá por -1, então caso seja raiz == NULL, retorno apenas o -1, caso não seja uma raiz nula e tenha algo é passado por uma variavel tanto esquerda como direita a própria função, retornando +1 caso a esquerda seja maior que a direita ou a direita seja maior que a esquerda, o código dessa parte se encontra dessa forma.

```
int maior_profun(Arvore *raiz){
if(raiz==NULL){
        return -1;
}else{
        int esq = maior_profun(raiz->esq); // esq = ?
        int dir = maior_profun(raiz->dir); // dir = ?
        if(esq > dir)
                return esq + 1;
```

```
            else
                return dir + 1;
        }
}
```

Diferente da função de maior profundidade, a função criada para saber a menor profundidade da árvore é um pouco mais complicada, sempre será executada caso a raiz seja diferente de nula, ou seja, caso há uma raiz ou folha, atendido esses exemplos, passa para o segundo if, se a raiz e sua esquerda e direita forem nulas no caso uma folha, entra em um condição para que a variavel responsavel pela menor profundidade receba o nivel atual, caso não, se n não for folha, o nivel recebe +1.

```
        void menor_profun(Arvore *raiz,int nivel,int *menor){
        if(raiz!=NULL){
            if(raiz->esq==NULL && raiz->dir==NULL) // se a raiz for uma folha
                {
                    if(*menor>nivel)
                        *menor=nivel;
                }
            else{ // se n for folha, nivel recebe +1
                menor_profun(raiz->esq,nivel+1,menor);
                menor_profun(raiz->dir,nivel+1,menor);
            }
        }
}
```

É necessário uma função para que possa ser liberada a memoria, se a raiz for diferente de nula, é feita a liberação, passando primeiro sua esquerda e direita para serem liberadas e depois liberando a própria raiz.

```
        void liberar(Arvore *raiz){
        if(raiz != NULL){
            liberar(raiz->esq);
            liberar(raiz->dir);
            free(raiz);
        }
}
```

Como é uma árvore binária de busca, tem que ter a função busca, passa a própria árvore como parametro junto com o elemento desejavel, e se o elemento for a própria raiz já retorna a raiz, caso não seja, for menor, é passado novamento a mesma função, só que com a esquerda da raiz até ser encontrado, da mesma forma é a execução para a parte direita da raiz.

```
        Arvore* buscar_elemento(Arvore *raiz, int elemento){
        if(raiz){
            if(elemento == raiz->num) // se o elemento for a propria raiz
                return raiz;
            else if(elemento < raiz->num)// se o elemento for menor que a raiz
                return buscar_elemento(raiz->esq, elemento);
            else // se o elemento for maior que a raiz
                return buscar_elemento(raiz->dir, elemento);
        }
        return NULL;
}
```

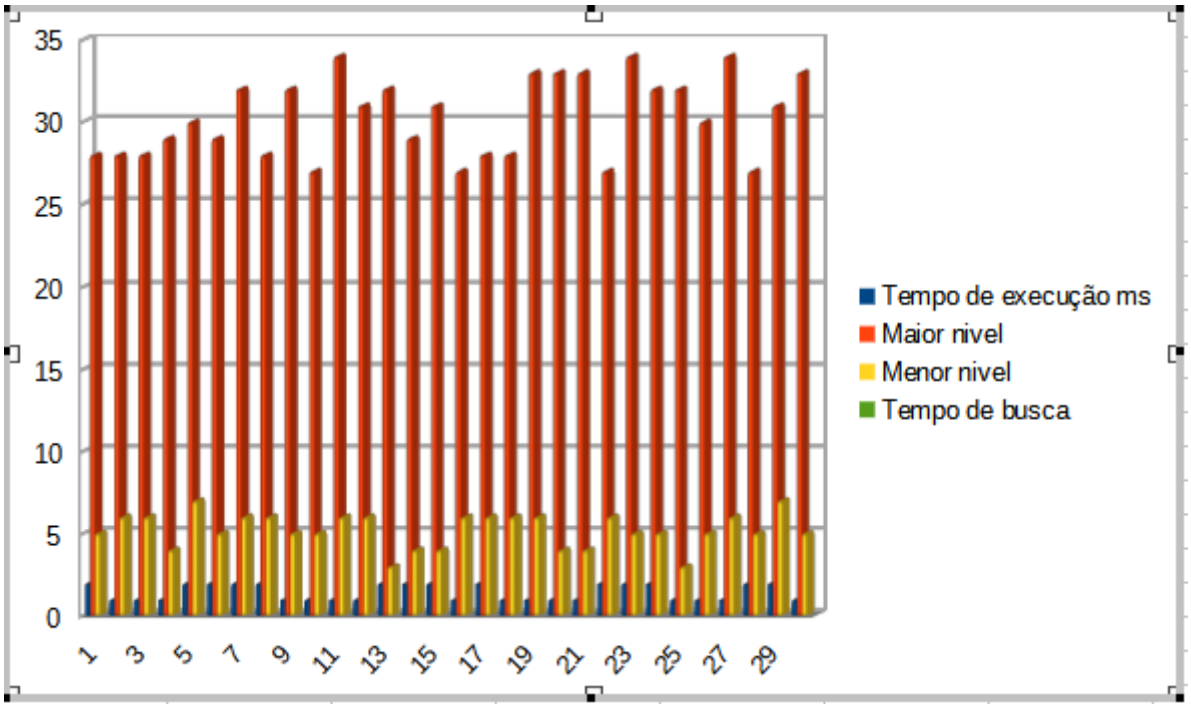
A função main é composta pela chamada das funções, além de fazer as atribuições para armazenar as profundidades maiores e menores, assim como a chamada da busca e o preenchimento da arvore atraves de loops usando rand com o tamanho determinado manualmente com 1000 inserções.

A estrutura da arvore AVL se dá de diferente pelas funções de novo nó, e na estrutura ter a altura e também o balanceamento com as rotações direita e esquerda.

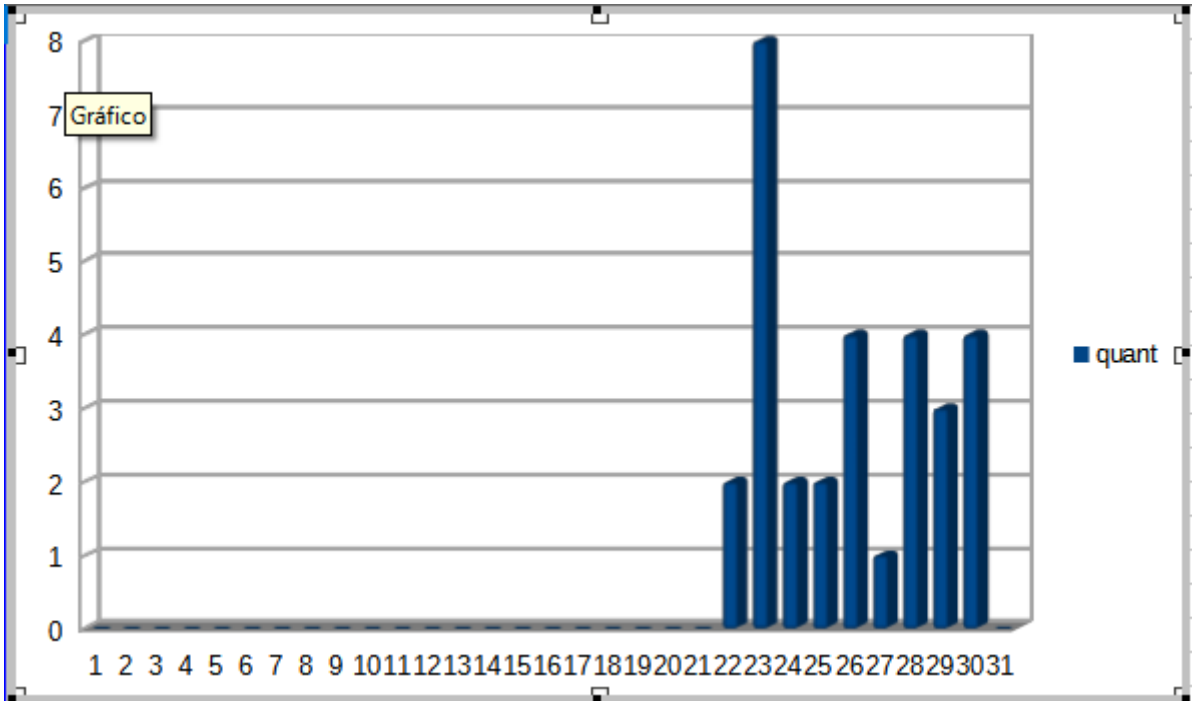
```
	int fatorDeBalanceamento(No *no){// calcula e retorna o fator de balanceamento de um nó
	if(no)
		return (alturaDoNo(no->esquerdo) - alturaDoNo(no->direito));
	else
		return 0;
}
No* rotacaoEsquerda(No *r){
	No *y,*f;

	y = r->direito;
	f = y->esquerdo;

	y->esquerdo = r;
	r->direito = f;

	r->altura = maior(alturaDoNo(r->esquerdo),alturaDoNo(r->direito)) + 1;
	y->altura = maior(alturaDoNo(y->esquerdo),alturaDoNo(y->direito)) + 1;

	return y;
}
No* rotacaoDireita(No *r){
	No *y,*f;

	y = r->esquerdo;
	f = y->direito;

	y->direito = r;
	r->esquerdo = f;

	r->altura = maior(alturaDoNo(r->esquerdo),alturaDoNo(r->direito)) + 1;
	y->altura = maior(alturaDoNo(y->esquerdo),alturaDoNo(y->direito)) + 1;

	return y;
}
No* rotacaoDireitaEsquerda(No *r){
	r->direito = rotacaoDireita(r->direito);
	return rotacaoEsquerda(r);
}
No* rotacaoEsquerdaDireita(No *r){
	r->esquerdo = rotacaoEsquerda(r->esquerdo);
	return rotacaoDireita(r);
}
```

Resultados da Execução do Programa:

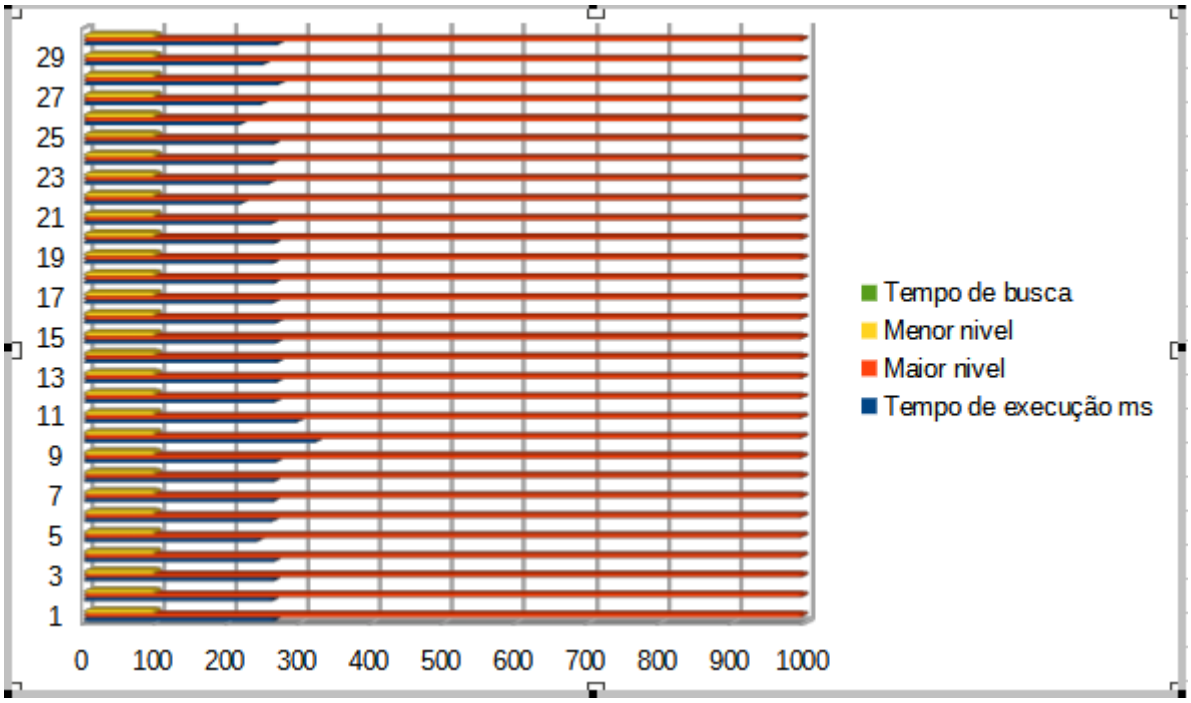
Abaixo tem o resultado em grafico do tempo de inserção, maior nivel, menor nivel e tempo de busca da árvore binaria com numeros aleatorios.



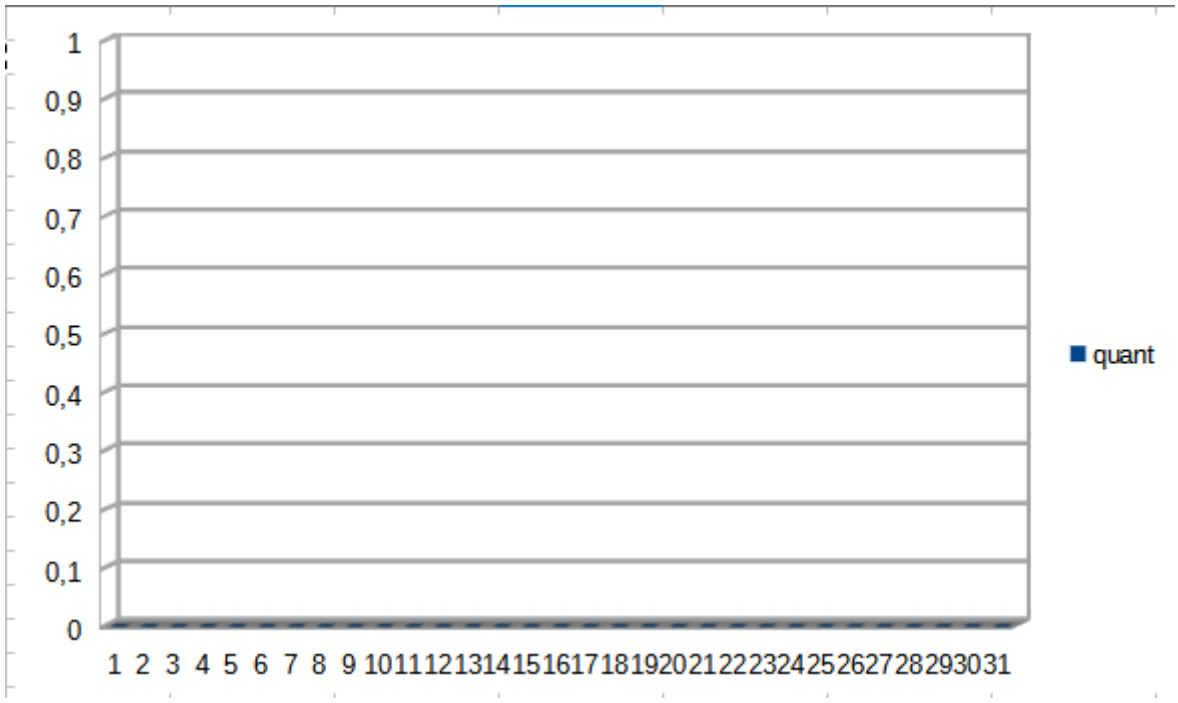
Abaixo a diferença maxima e minima de profundidade:



Abaixo tem o resultado em grafico do tempo de inserção, maior nivel, menor nivel e tempo de busca da árvore binaria com numeros sequenciais.

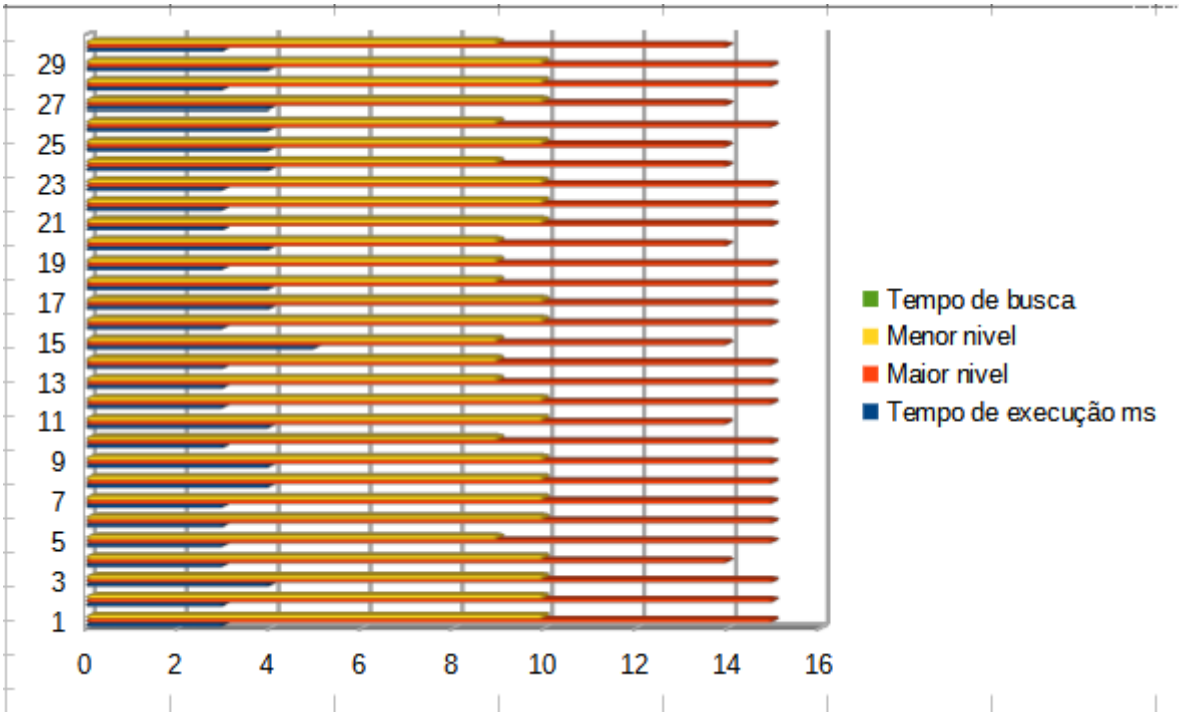


Como é possível ver, com números sequências, não há diferença do mínimo e máximo no intervalo, todos deram valor zerado.

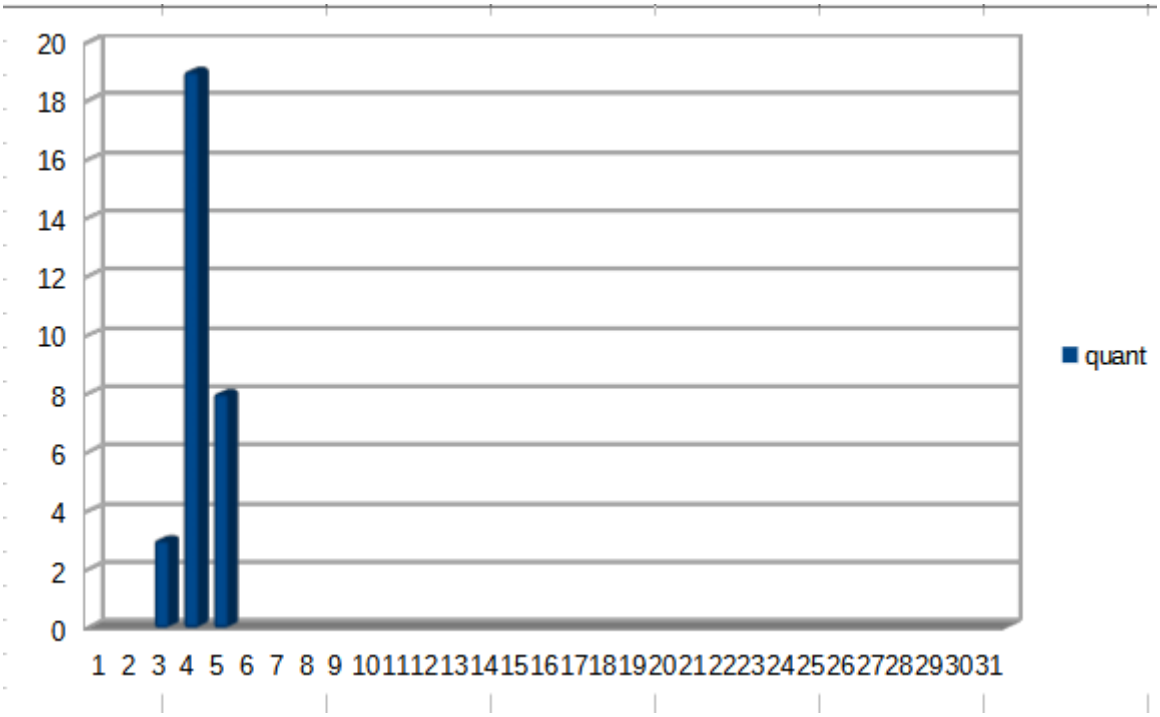


Todos esses gráficos e saídas foram testados com 1000 inserções.

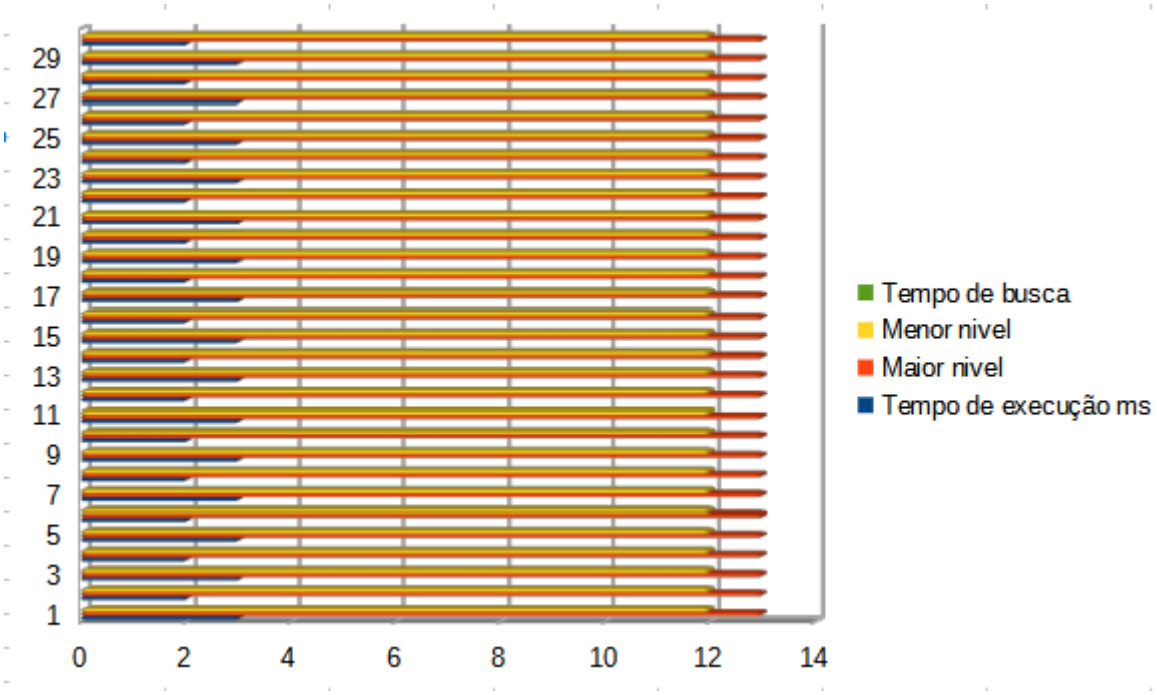
Inserções em árvore AVL:



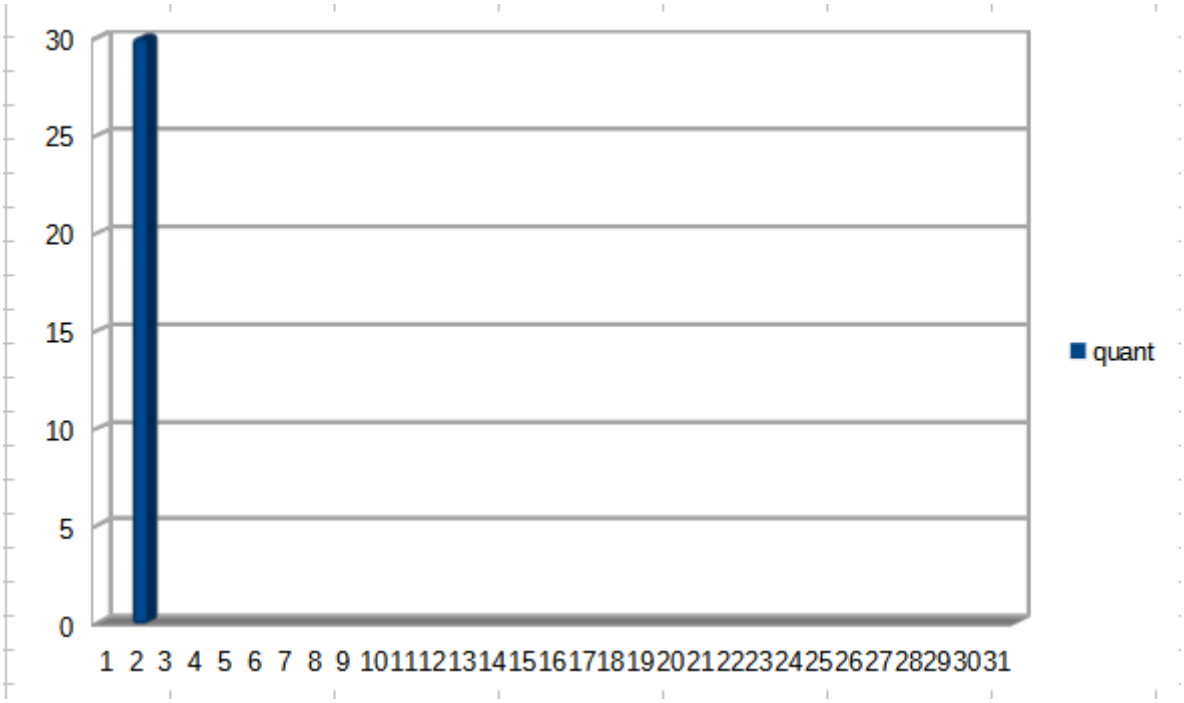
Diferença máxima e mínima de profundidade em AVL:



Com números sequências até 10000 inserções:



Diferença mínima e máxima de profundidade com números sequenciais:



Conclusão:

O experimento teve como propósito realizar a comparação entre os dois métodos de árvore, com o objetivo de descobrir qual estrutura de dados seria melhor utilizando os mesmos parâmetros com números aleatórios e sequenciais, em alguns experimentos foi testado além do propósito, como é possível visualizar, nos números sequenciais, para ter uma noção de tempo de inserção, foi necessário realizar com 10000 mil inserções, já que

com apenas mil não era o suficiente para obter os valores que era almejado, já que a árvore AVL se saía com valores em ms muito melhores. No geral para a inserção, o melhor resultado se deu pela árvore AVL, sendo necessário realizar uma grande massa de inserções para obter resultados parecidos com a binária.

Com isso, é correto afirmar que o propósito da realização das duas questões, foi possível concluir o que se procurava.