Relatório de Estrutura de Dados da Segunda Unidade

Eduardo Victor Nóbrega Fernandes

Julho, 2018

Algoritmos de árvore binária, árvore balanceada e tabela de dispersão

Este relatório busca demonstrar a utilização de busca em algoritmos de árvore binária, árvore balanceada e tabela de dispersão e suas respectivas complexidades. Para atingir tal objetivo utilizamos a linguagem C para implementação e testes dos mesmos.

Árvore binária

A árvore binária ou árvore de pesquisa binária é uma árvore binária onde todos os nós são valores, todo nó a esquerda contêm uma sub-árvore com os valores menores ao nó raiz da sub-árvore e todos os nós da sub-árvore a direita contêm somente valores maiores ao nó raiz.

Termos da árvore:

- Nó: são todos os itens guardados na árvore.
- Raiz é o item do topo da árvore, no exemplo acima é o nó 8.
- Filho são os itens logo abaixo da raiz, no nosso exemplo 3 e 10 são filhos de 8, da mesma forma que 1 e 6 são filhos de 3.
- Parente são os nós do mesmo nível, no exemplo acima 6 e 14 são parentes de 1.
- Folha é um nó que não tem filho, são os últimos itens da árvore, no exemplo acima são os itens 4, 7 e 13.

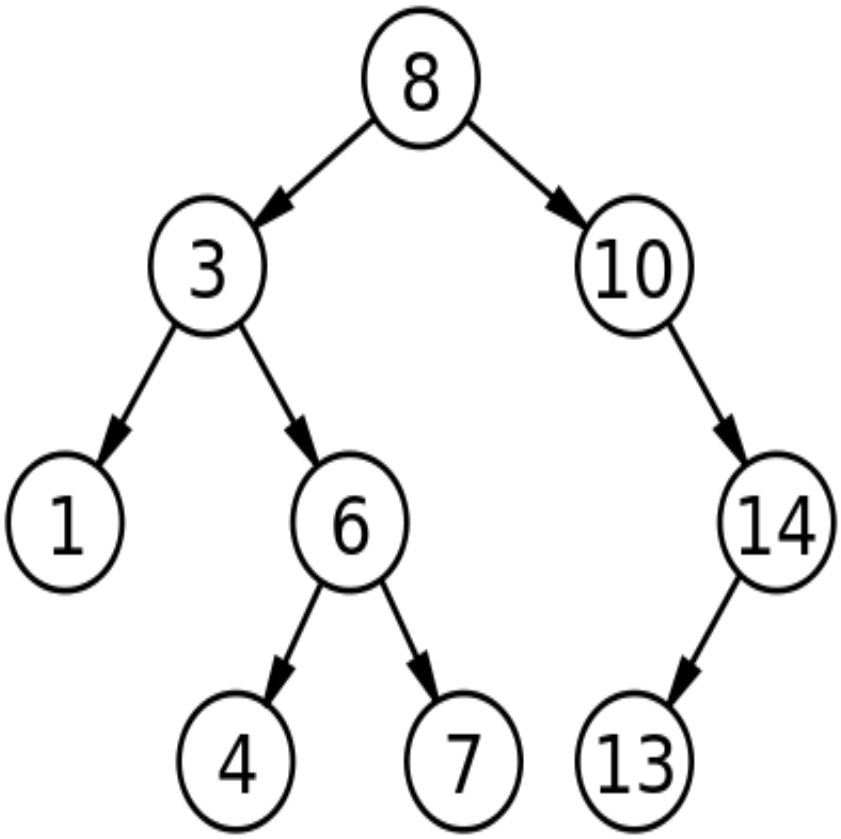

Figure 1: Exemplo de árvore binária

O algoritmo em questão funciona por meio de 3 funções: uma função que cria um novo nó, uma função que insere o nó na árvore e outra que serve para buscar o elemento.

```
struct tnode *newTnode(int v){
    struct tnode *n = (struct tnode *) malloc(sizeof(struct
    n->v = v;
    n->l = NULL;
    n->r = NULL;
    return n;
}
```

Figure 2: Cria um novo nó

```c
void tinsertInterativa(struct tnode **R, int v){
    struct tnode *p, *f;
    if(*R == NULL){
        (*R) = newTnode(v);
    }
    else{
        p = (*R);
        while(p != NULL){
            f = p;
            if(p->v < v){
                p = p->r;
            }
            else{
                p = p->l;
            }
        }
        if(f->v < v){
            f->r = newTnode(v);
        }
        else{
            f->l = newTnode(v);
        }
    }
}
```

Figure 3: Insere o nó na árvore

```
struct tnode *tsearch(struct tnode *R, int v){
    if(R != NULL){
        if(R->v == v){
            return R;
        }
        if(R->v < v){
            return tsearch(R->r , v);
        }
        else{
            return tsearch(R->l, v);
        }
    }
    else{
        return NULL;
    }
}
```

Figure 4: Busca o nó pelo valor

Análise de Complexidade

Melhor Caso

O melhor caso da busca em árvore binária se dar quando independente do número de termos, o elemento a ser buscado é encontrado de primeira. O que geralmente ocorre é a busca na árvore começar pela raiz, logo o elemento que a ser buscado estará logo na raiz. O tempo desse caso é $O(1)$;

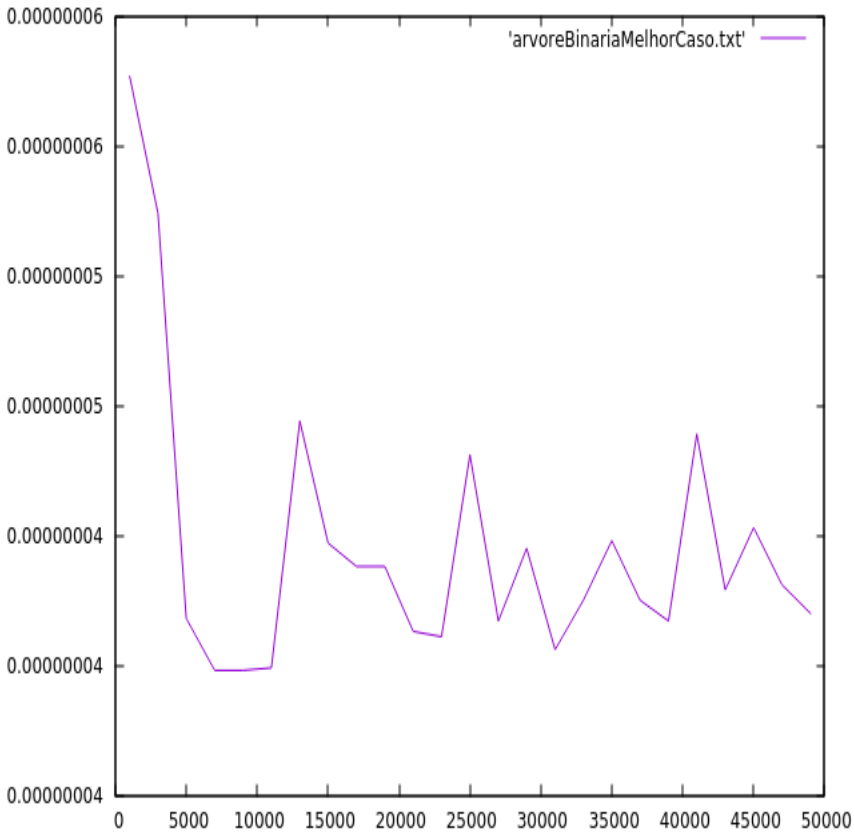

Figure 5: Melhor caso da árvore binária

Pior Caso

O pior caso da busca em árvore binária se dar quando independente do número de termos, o elemento a ser buscado não é encontrado. O tempo desse caso é $O(n)$;

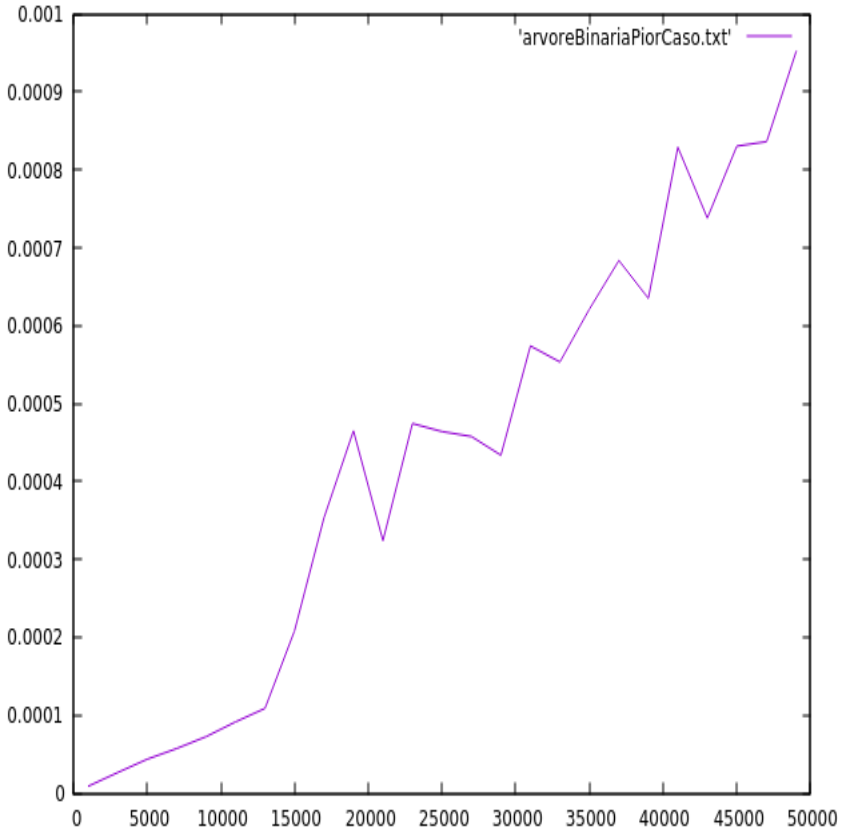

Figure 6: Melhor caso da árvore binária

Caso Médio

O tempo desse caso é $O(\log n)$.

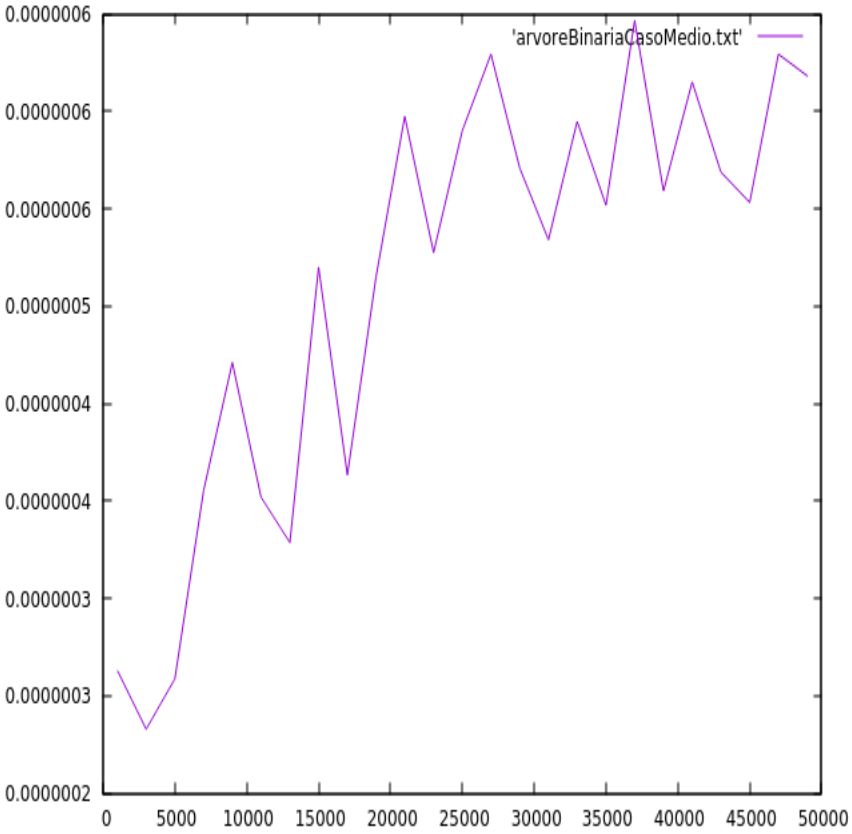

Figure 7: Caso médio da árvore binária

Comparativo dos casos

No primeiro gráfico, olhando a primeira vista, percebe-se apenas dois tempos, mas isso ocorre porque o caso médio é muito mais rápido do que o pior caso. Para deixar claro a diferença entre o caso médio e o melhor caso foi inserido mais um gráfico, que se trata apenas de um zoom nos tempos mais rápidos.

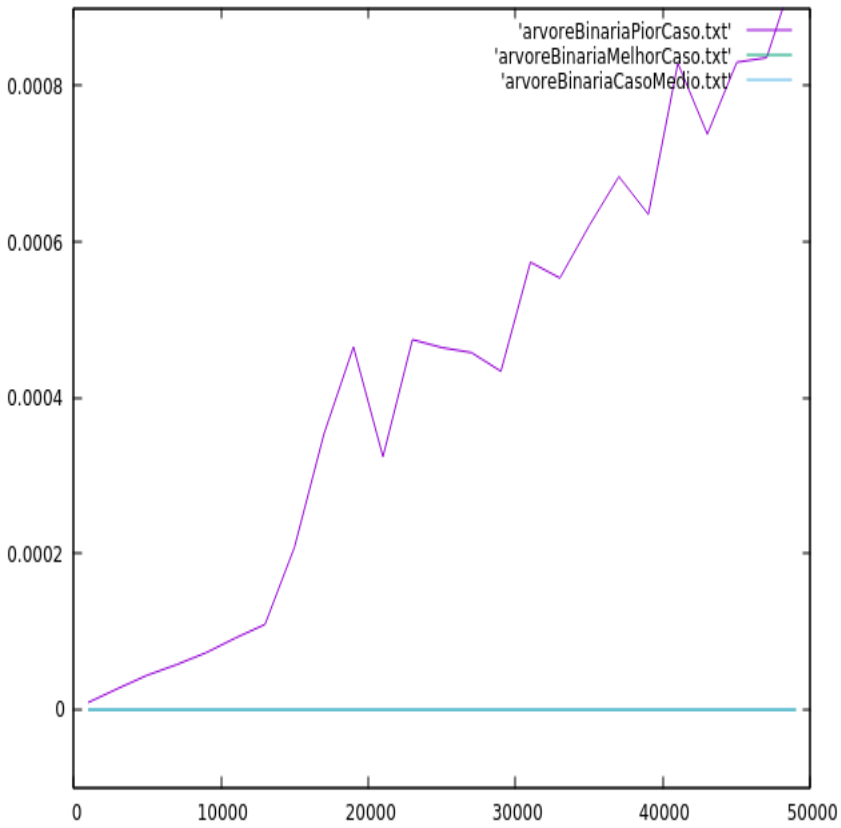

Figure 8: Comparativo dos 3 casos da árvore binária

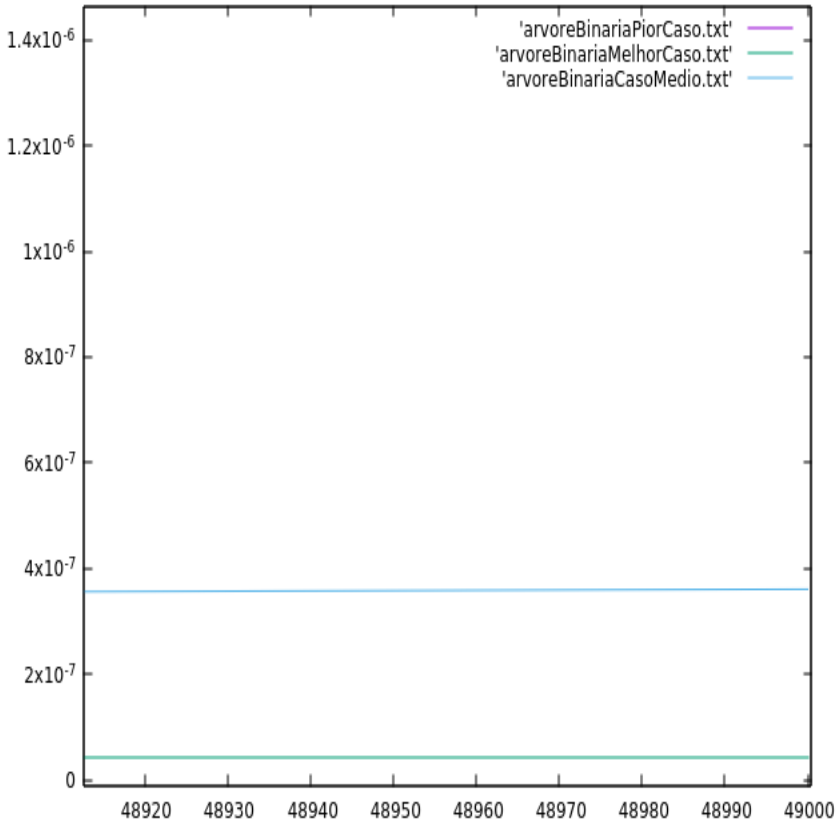

Figure 9: Zoom dos 2 casos mais rápidos da árvore binária

Tabela de Dispersão

A tabela de dispersão (mais conhecida como tabela hash), é uma estrutura de dados especial, que associa chaves de pesquisa a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado.

Hashing é uma maneira de organizar dados que:

- apresenta bons resultados na prática,
- distribui os dados em posições aleatórias de uma tabela.
- Uma tabela hash é construida através de um vetor de tamanho n, no qual se armazenam as informações.
- Nele, a localização de cada informação é dada a partir do cálculo de uma índice através de uma função de indexação, a função de hash.

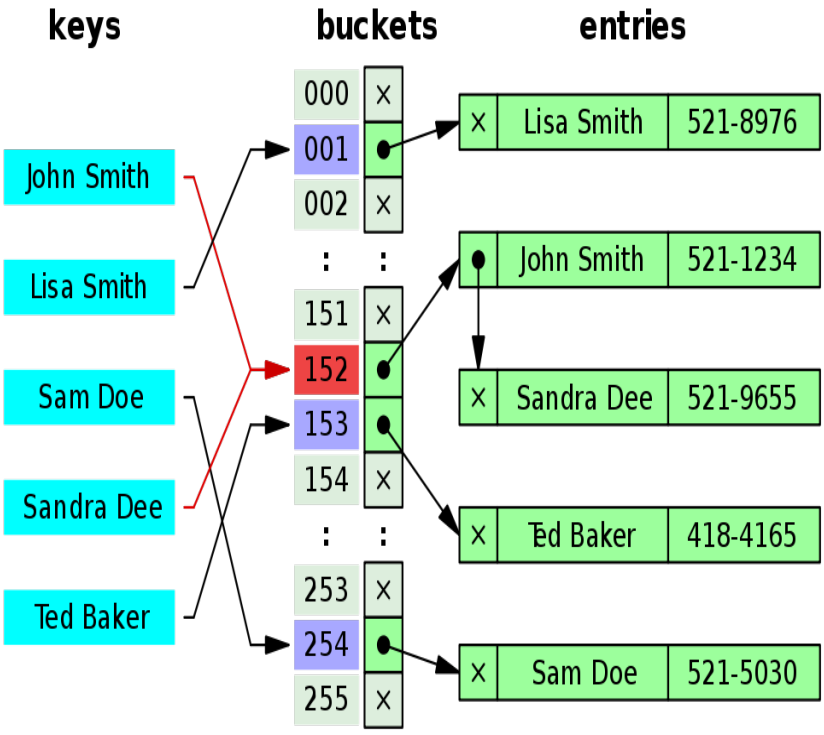

Figure 10: Exemplo de utilização de tabela hash

```
void insertTable(Table* table, Lista* list){
    int position;
    if(table->elements > table->size){
        updateSizeTable(table);
    }
    position = calcPosition(table->size,list->v);
    if(table->l[position]==NULL){
        table->l[position] = list;
    }else{
        list->prox=table->l[position];
        table->l[position]=list;
    }
    table->elements++;

}
```

Figure 11: Insere o valor na tabela

Neste algoritmo o cálculo utilizado foi o resto da divisão do valor pelo número de elementos do vetor.

Afim de manter a velocidade na hora da busca é feito um redimensionamento do vetor caso o número de elementos de cada lista seja maior que o número de elementos do vetor. Esse redimensionamento faz com que todos os elementos passem pra outra lista, que após isso terá o dobro do tamanho.

Este algoritmo é composto de 4 funcões: uma função de inserir na tabela, uma função de calcular o valor do hash (para sabermos qual a posição que deve-se inserir), uma função responsavel por verificar e redimensionar a tabela caso necessário e uma que serve para buscar o valor desejado.

Análise de complexidade

A complexidade deste algoritmo é $O(1)$.

```c
int calcPosition(int tam, int v){
    return v%tam;
}
```

Figure 12: Calcula o valor do hash

```c
void updateSizeTable(Table* table){

    Table aux;
    int size = table->size * 2;
    aux.l = malloc(sizeof(Lista*)*(size));
    aux.size = size;
    aux.elements = 0;
    Lista* list;

    for(int i = 0; i < aux.size; i++){
        aux.l[i] = NULL;
    }

    for(int j = 0; j < table->size; j++){
        list = table->l[j];
        while(list != NULL){
            Lista *lista = newList(list->v);
            int x = calcPosition(aux.size, list->v);
            if(aux.l[x] == NULL){
                aux.l[x] = lista;
            }else{
                lista->prox = aux.l[x];
                aux.l[x] = lista;
            }
            list = list->prox;
            aux.elements++;
        }
    }
    freeTable(table);
    table->size = size;
    table->elements = aux.elements;
    table->l = aux.l;

}
```

Figure 13: Verifica e redimensiona se necessário

```
Lista* searchHash(Table* table, int v){
    Lista* list;
    for(int i = 0; i < table->size; i++){
        list = table->l[i];
        while(list != NULL){
            if(list->v == v){
                return list;
            }
            list = list->prox;
        }
    }
    return NULL;
}
```

Figure 14: Busca pelo valor desejado

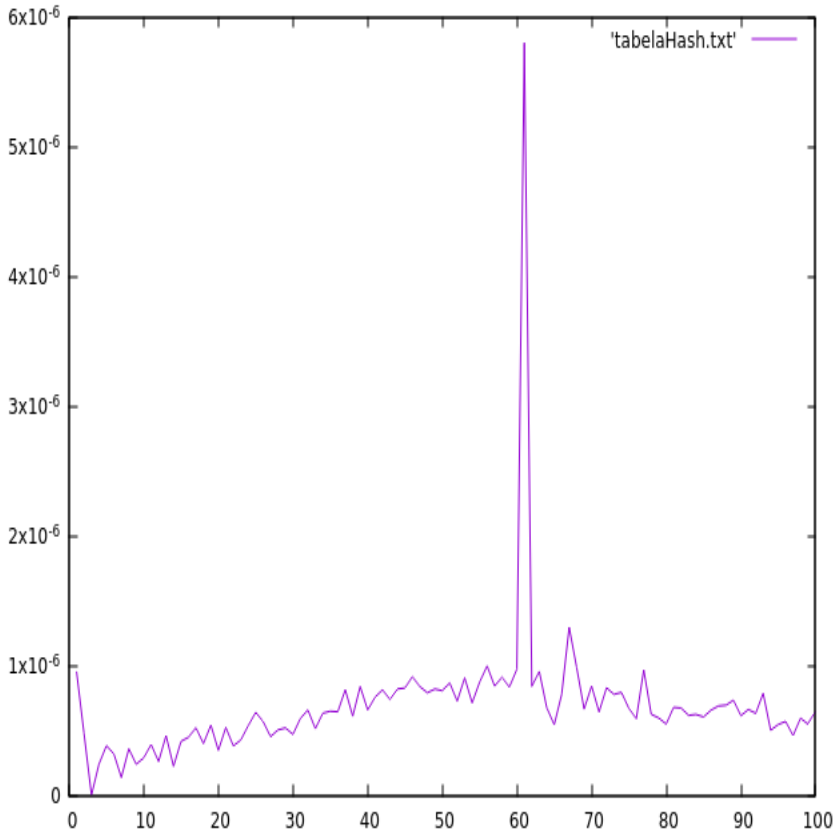

Figure 15: Tabela Hash