Universidade Federal do Rio Grande do Norte – UFRN
Centro de Ensino Superior do Seridó – CERES
Departamento de Computação e Tecnologia – DCT
Bacharelado em Sistemas de Informação – BSI

Atividade Prática II

Emanuel Alves de Medeiros

Orientador: Prof. Dr. João Paulo De Souza Medeiros

Relatório apresentado ao Curso de Bacharelado em Sistemas de Informação como parte avaliativa da matéria de Estrutura de Dados.

Caicó, RN, 27 de agosto de 2024

Sumário

1. Introdução

Algoritmos de busca desempenham um papel fundamental na área da ciência da computação, ajudando a resolver uma variedade de problemas de forma eficiente. Eles são projetados para localizar informações específicas dentro de conjuntos de dados.

Neste relatório, faremos uma análise e comparação dos diferentes algoritmos de busca apresentados pelo professor na matéria de Estrutura de Dados, a fim de mostrar sua complexidade, desempenho e características distintas. Os algoritmos que serão examinados incluem: árvore binária, árvore binária balanceada (AVL) e tabela de dispersão (hash).

Ao longo deste relatório, iremos explorar os princípios básicos desses algoritmos, discutindo suas vantagens e desvantagens em termos de tempo de execução, além de observar seu comportamento em diferentes cenários, considerando casos de melhor e pior desempenho. Para melhor visualização e compreensão, utilizaremos gráficos comparativos.

2. Códigos dos algoritmos

Todos os algoritmos feitos nesse trabalho foram usando a linguagem C, e os gráficos em python. Como são algoritmos rápidos, cada teste de busca foi realizado 1000 vezes, ou seja, o tempo que mostra no gráfico, é 1000 vezes maior que uma execução única.

2.1 Árvore binária

Uma árvore binária é uma estrutura composta por uma coleção de nós interconectados, onde cada nó pai pode ter no máximo dois filhos: um filho à esquerda e um à direita.

Na figura 2.1 temos o código inserção de elementos em uma árvore binária, de uma forma bem simples, o algoritmo vai descendo a arvore indo pra direita se o número a adicionar for maior, caso contrário, para a equerda, até não haver mais elementos, fazendo assim, a inserção.

2.2 Árvore binária balanceada (AVL)

Na figura 2.3 temos o código feito em C. A árvor AVL tenta melhorar a árvore binária, já que ela pode ficar mais extensa de um lado do que de outro, fazendo com que não fique os mais eficiente possível. As principais mudanças entre esse código e o da figura 2.1 é que a AVL tenta se balancear, precisando assim de algumas funções extras, também da pra perceber que o código é igual ao da árvore binária, porém, com algumas adições.

```
1  void btinsert(struct node **r, struct node *n) {
2      if (*r != NULL) {
3          if ((*r)->v > n->v) {
4              btinsert(&(*r)->l, n);
5              (*r)->l->p = *r;
6          } else {
7              btinsert(&(*r)->r, n);
8              (*r)->r->p = *r;
9          }
10     } else {
11         (*r) = n;
12     }
13 }
```

Figura 2.1: Código em C: btinsert.

```
void avlinsert(struct node** r, struct node* n) {
    if (*r == NULL) {
        *r = n;
    } else if (n->v < (*r)->v) {
        avlinsert(&((*r)->l), n);
        (*r)->l->p = *r;
    } else if (n->v > (*r)->v) {
        avlinsert(&((*r)->r), n);
        (*r)->r->p = *r;
    }

    balance(r);
}
```

Figura 2.2: Código em C: avlinsert.

2.3 Tabela de dispersão (hash)

Uma tabela de dispersão, também conhecida como hash table, é uma estrutura de dados baseada em uma função de dispersão (hash function) que mapeia chaves de pesquisa em índices de armazenamento.

Na figura 2.4 temos o código feito em C de como inserir elementos em uma tabela de dispersão, essa função calcula onde que o valor da chave deve ficar amazenado, mas como isso pode gerar colisões (chaves iguais ou diferentes ocupando o mesmo espaço), o dado é uma lista encadeada, se a quantidade de elementos que estiver na tabela for maior ou igual ao tamanho da tabela é necessário atualizar o tamanho do vetor usando a função de atualização 2.5. Na atualização, é necessário aumentar o tamanho da lista, logo, também é necessário reposicionar todos na tabela, já que o tamanho mudou, o valor do cálculo também muda.

```
int get_height(struct node *n) {
    if (n == NULL)
        return 0;
    return n->h;
}

void update_height(struct node* r) {
    int hl = get_height(r->l);
    int hr = get_height(r->r);
    r->h = (hl > hr ? hl : hr) + 1;
}

void rr(struct node** n) {
    struct node* aux = (*n)->l;
    (*n)->l = aux->r;
    if (aux->r != NULL) aux->r->p = (*n);
    aux->p = (*n)->p;
    aux->r = (*n);
    (*n)->p = aux;
    (*n) = aux;
    update_height((*n)->r);
    update_height(*n);
}

void lr(struct node** n) {
    struct node* aux = (*n)->r;
    (*n)->r = aux->l;
    if (aux->l != NULL) aux->l->p = (*n);
    aux->p = (*n)->p;
    aux->l = (*n);
    (*n)->p = aux;
    (*n) = aux;
    update_height((*n)->l);
    update_height(*n);
}

void balance(struct node** r) {
    update_height(*r);
    int hl = get_height((*r)->l);
    int hr = get_height((*r)->r);
    if (abs(hl - hr) > 1) {
        if (hl > hr) {
            if (get_height((*r)->l->l) >= get_height((*r)->l->r)) {
                rr(r);
            } else {
                lr(&((*r)->l));
                rr(r);
            }
        } else {
            if (get_height((*r)->r->r) >= get_height((*r)->r->l)) {
                lr(r);
            } else {
                rr(&((*r)->r));
                lr(r);
            }
        }
    }
}
```

Figura 2.3: Código em C: avlinsert parte 2.

```
void hinsert(struct htable *t, struct block *b) {
    if(t->n >= t->m) {
        hrefresh(t, b);
    } else {
        int i = b->v % t->m;
        b->next = t->l[i];
        t->l[i] = b;
        t->n = t->n + 1;
    }
}
```

Figura 2.4: Código em C: hinsert.

```
void hrefresh(struct htable *t, struct block *b) {
    struct htable nt;
    nt.m = (t->m * 2);
    nt.l = malloc(sizeof(struct block *) * nt.m);
    nt.n = 1;

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

    int i = 0;
    nt.l[i] = b;

    for (int i = 0; i < t->m; i++){
        struct block *nb;
        nb = t->l[i];

        while (nb != NULL) {
            struct block *tab = create_block(nb->v);

            tab->next = nt.l[i];
            nt.l[tab->v % nt.m] = tab;

            nb = nb->next;

            nt.n = nt.n + 1;
        }
    }
    (*t) = nt;
}
```

Figura 2.5: Código em C: hrefresh.

3. Tempo de busca

3.1 Árvore binária

O tempo da árvore binária depende de como ela está organizada, ou seja, da ordem de inserção os elementos, que ditará se está organizada ou não. Veja o gráfico 3.1 com todos os casos.

3.1.1 Melhor caso

Apesar das estruturas serem diferentes, o melhor caso sempre será quando o valor procurado estiver na raiz, fazendo com que a primeira busca, já seja a correta, deixando assim, o tempo de execução constante. 3.2.

3.1.2 Pior caso

O pior caso, ocorrerá quando o elemento não estiver na árvore, e a árvore estiver organizada semelhante a uma lista encadeada, por exemplo, se números em sequência(crescentes ou decrescentes) forem adicionados na árvore binária. Fazendo com que o tempo de execução seja linear. 3.3.

3.1.3 Caso esperado

O caso esperado da árvore binária, é quado números aleatórios são adicionados em ordem aleatória, o que deixa os elementos da árvore mais "espalhados", e como o algoritmo de busca, geralmente divide a árvore ao meio sempre que verifica, o tempo do caso esperado é logarítmico (Veja a figura 3.5). Veja também a comparação com o melhor caso 3.4.

3.2 Árvore binária balanceada (AVL)

O algoritmo de busca da AVL é o mesmo da árvore binária 3.10, a única diferença é como os elementos são inseridos. Veja o gráfico 3.6 com todos os casos.

3.2.1 Melhor caso

O melhor caso da avl será quando o valor procurado estiver na raiz, será da mesma forma do melhor caso da árvore binária, deixando o tempo constante (Veja figura 3.7)

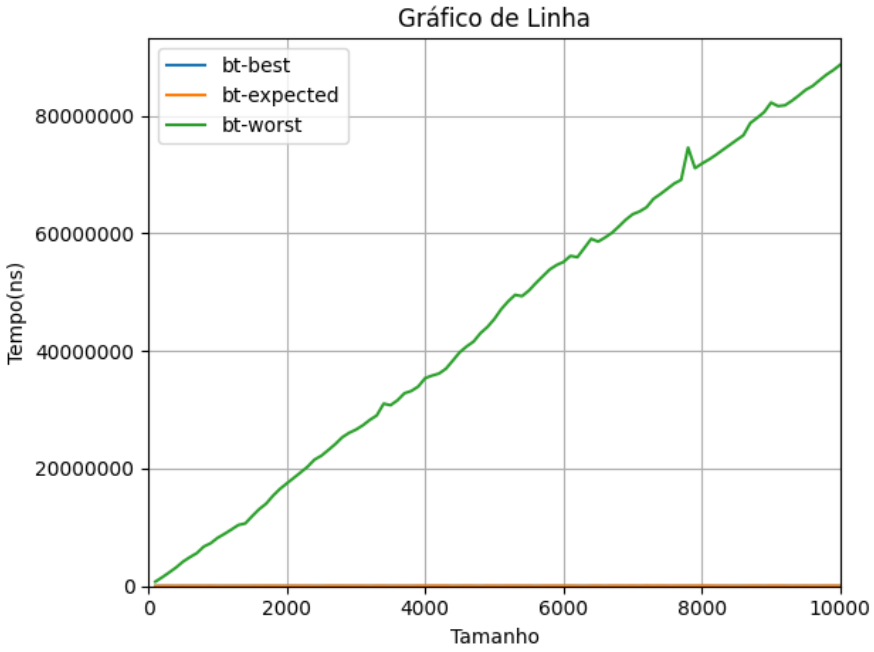

Figura 3.1: Todos os casos árvore binária (tamanho do vetor x tempo).

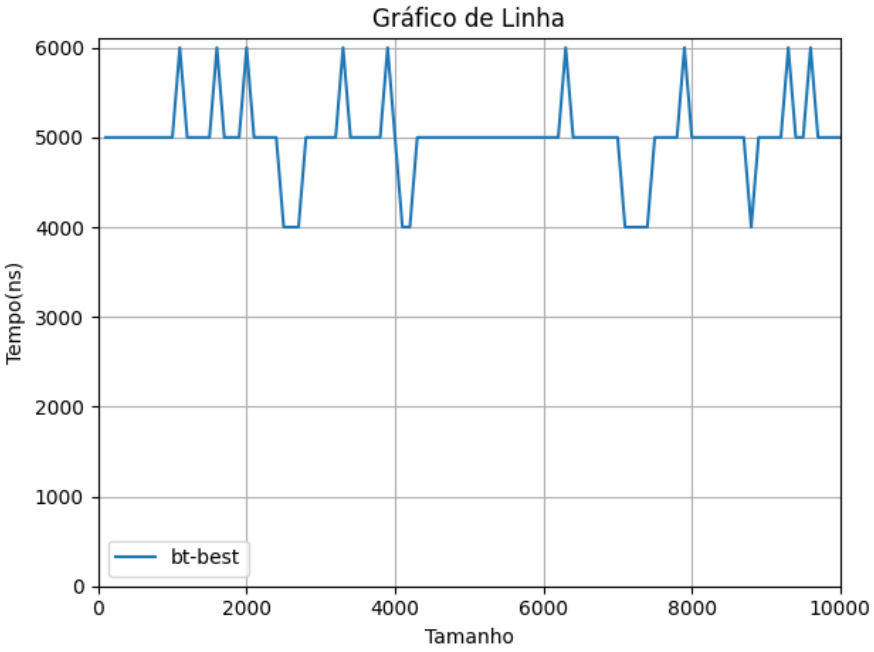

Figura 3.2: Tempo de execução da árvore binária no melhor caso ($\Theta(1)$). (tamanho do vetor x tempo) .

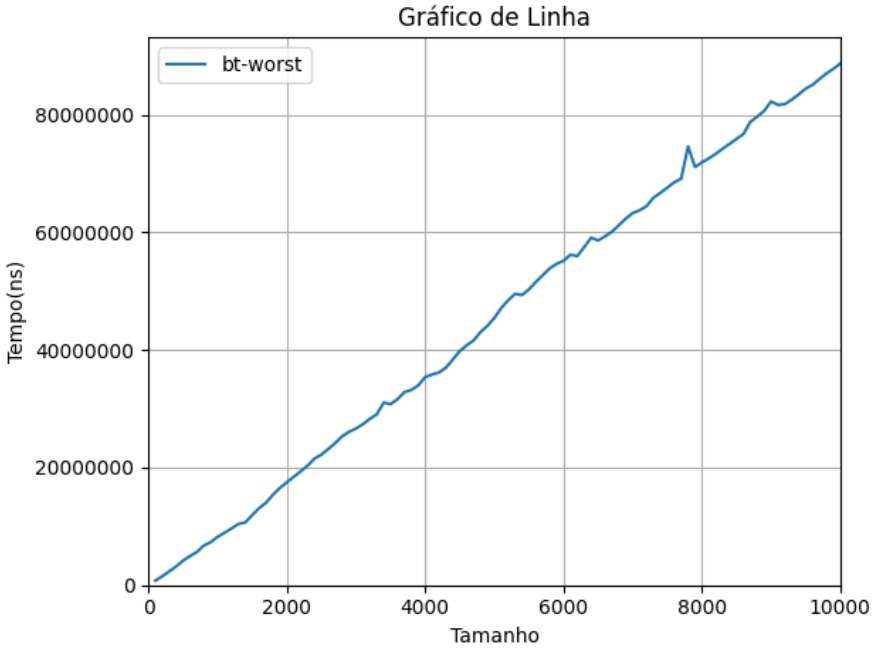

Figura 3.3: Tempo de execução da árvore binária no pior caso ($\Theta(n)$). (tamanho do vetor x tempo) .

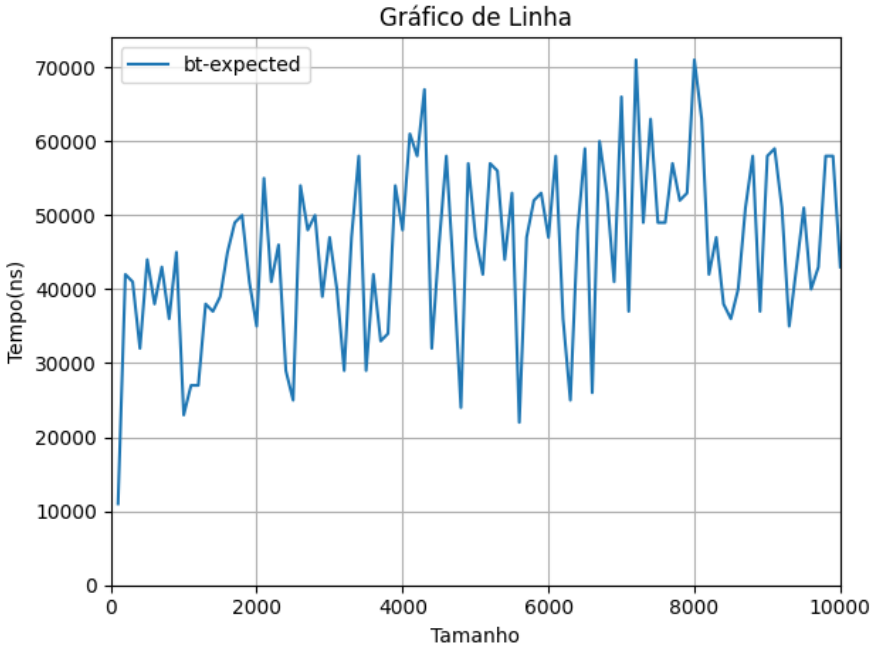

Figura 3.4: Tempo de execução da árvore binária no caso esperado ($\Theta(\log_2 n)$). (tamanho do vetor x tempo) .

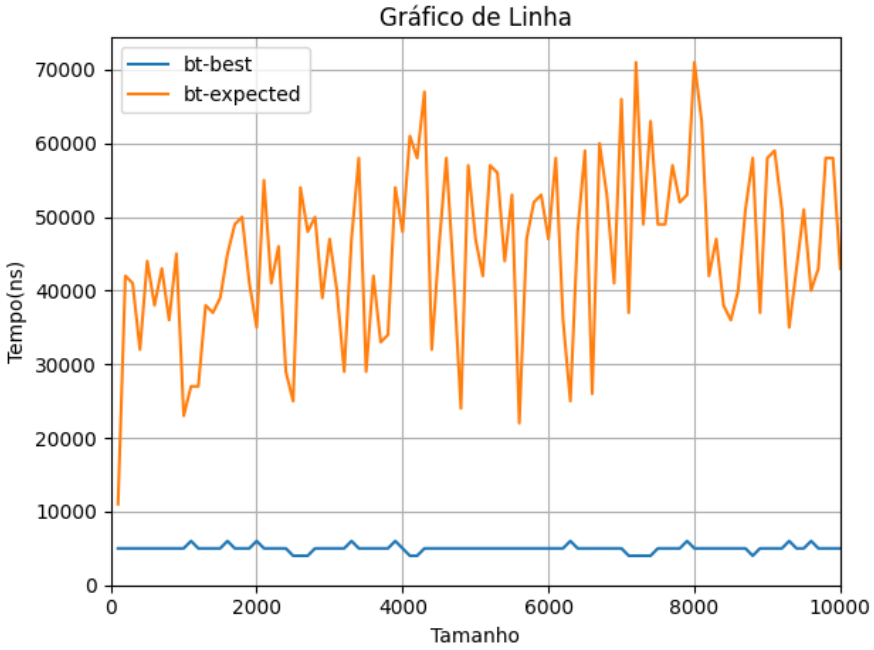

Figura 3.5: Tempo de execução da árvore binária no melhor caso ($\Theta(1)$) e caso esperado ($\Theta(\log_2 n)$). (tamanho do vetor x tempo) .

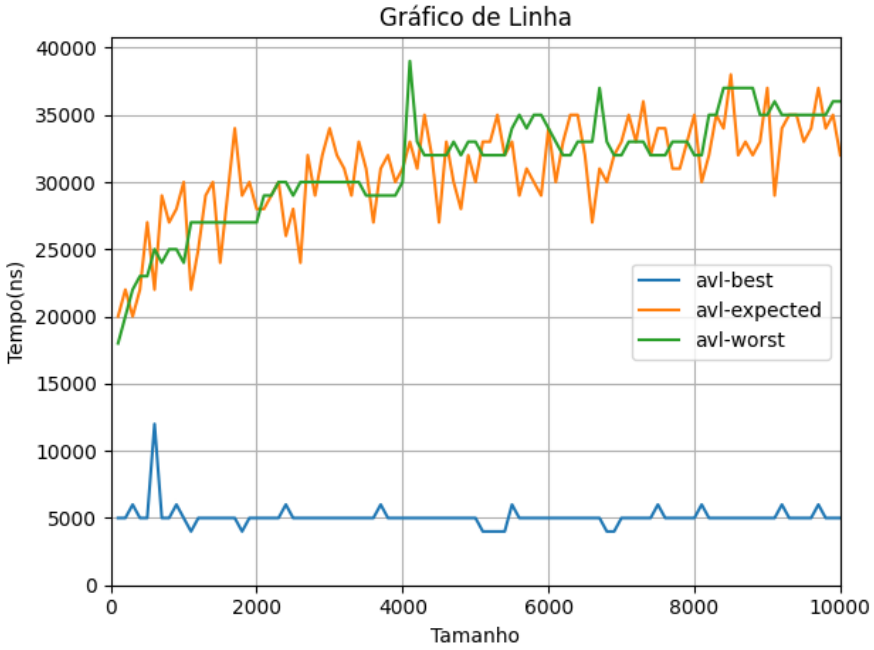

Figura 3.6: Tempo de execução da árvore avl no melhor caso ($\Theta(1)$). (tamanho do vetor x tempo).

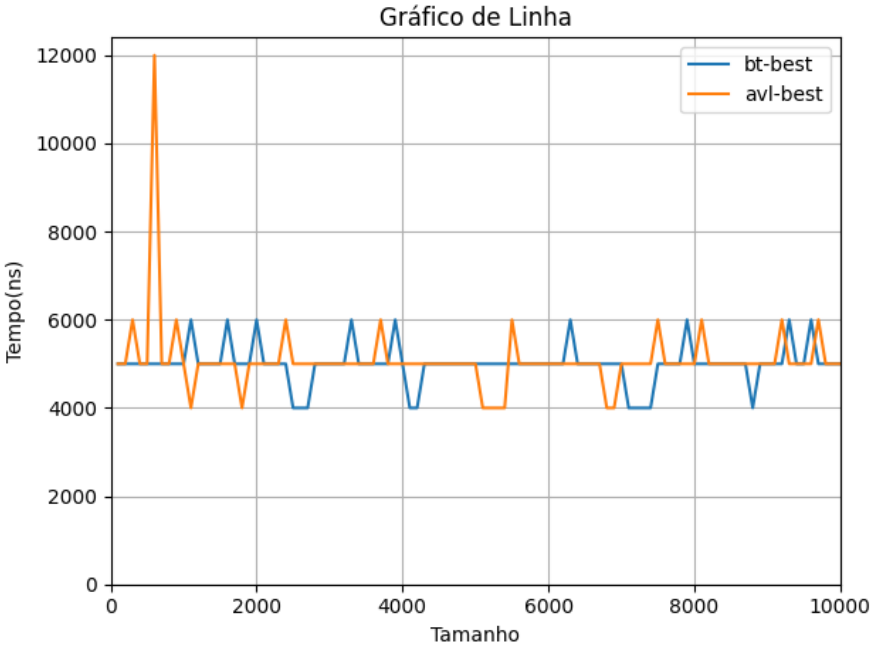

Figura 3.7: Tempo de execução da árvore avl no melhor caso ($\Theta(1)$). (tamanho do vetor x tempo)
.

3.2.2 Pior caso

O pior caso da avl, ocorrerá quando o elemento não estiver na árvore, pois irá fazer o máximo de verificações, sendo assim, de tempo logarítmico, já que ainda sim irá dividir a árvore ao meio sempre (Veja a figura 3.8).

3.2.3 Caso esperado

O caso esperado da avl é o mesmo do pior caso, logarítmico, podendo ser ligeiramente mais rápido quando o elemento estiver na lista (Veja o gráfico 3.9).

3.3 Tabela de dispersão (hash)

A tabela de dispersão (hash) pode estar bem espalhada ou não, se ela tiver bastante espaço alocado, ocorrerão bem menos colisões, deixando a busca mais rápida, caso contrário, ficará mais lenta. Veja o gráfico 3.11 com todos os casos.

3.3.1 Melhor caso

O melhor caso da tabela hash, é quando o elemento buscado está na primeira posição da lista, uma maneira de simular isso, é colocando o tamanho da lista igual ao tamanho de elementos mais um. 3.12

Figura 3.8: Tempo de execução da avl no pior caso ($\Theta(\log_2 n)$). (tamanho do vetor x tempo) .

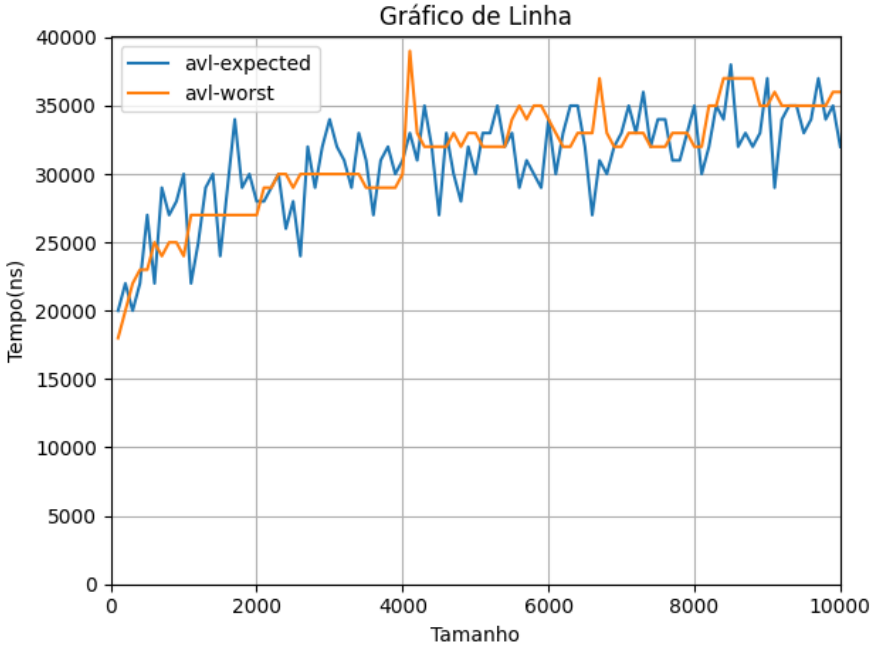

Figura 3.9: Tempo de execução da avl no pior caso ($\Theta(\log_2 n)$) e tempo esperado ($\Theta(\log_2 n)$). (tamanho do vetor x tempo) .

```c
struct node** btsearch(struct node** r, int v) {
    if ((*r) != NULL) {
        if ((*r)->v == v) {
            return r;
        } else if ((*r)->v > v) {
            return btsearch(&((*r)->l), v);
        }
        return btsearch(&((*r)->r), v);
    }
    return NULL;
}
```

Figura 3.10: Código em C: btsearch.

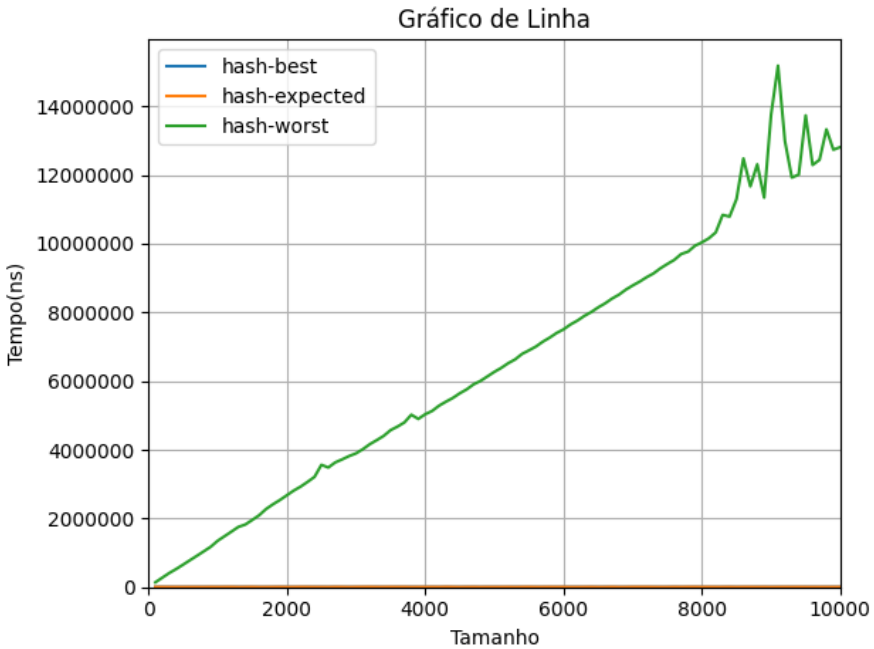

Figura 3.11: Tempo de execução da tabela de dispersão no melhor caso ($\Theta(1)$). (tamanho do vetor x tempo).

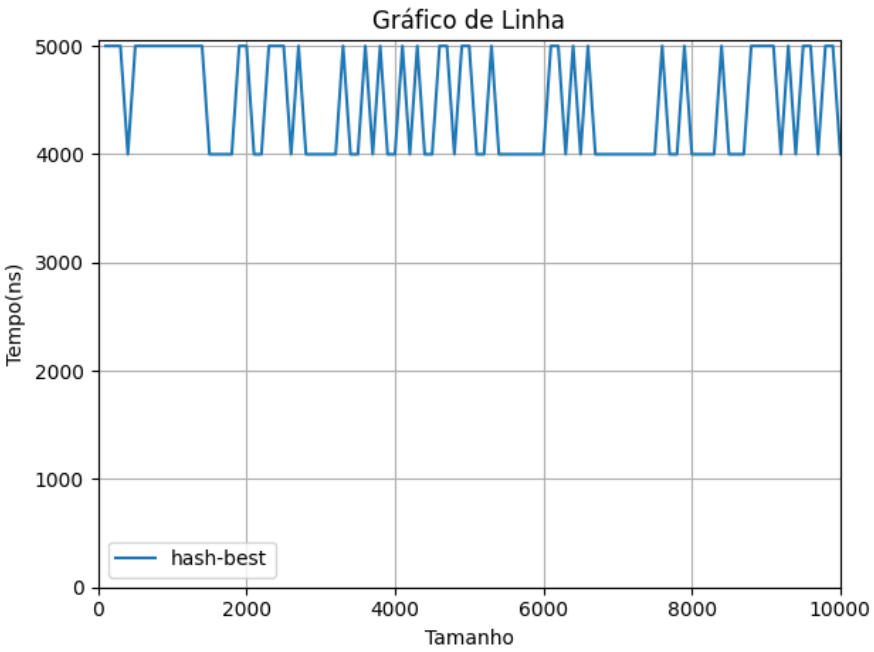

Figura 3.12: Tempo de execução da tabela de dispersão no melhor caso ($\Theta(1)$). (tamanho do vetor x tempo).

3.3.2 Pior caso

O pior caso ocorrerá quando todos os elementos estão em colisão, pos exemplo, todos estarem na posição 0, se transformando em uma lista encadeada, portanto, deixando o tempo linear. 3.12

3.3.3 Caso esperado

O caso esperado será o mesmo do esperado, constante, já que é quando o o tamanho da lista é maior à quantidade de elementos, tornando as colisões pouco prováveis.

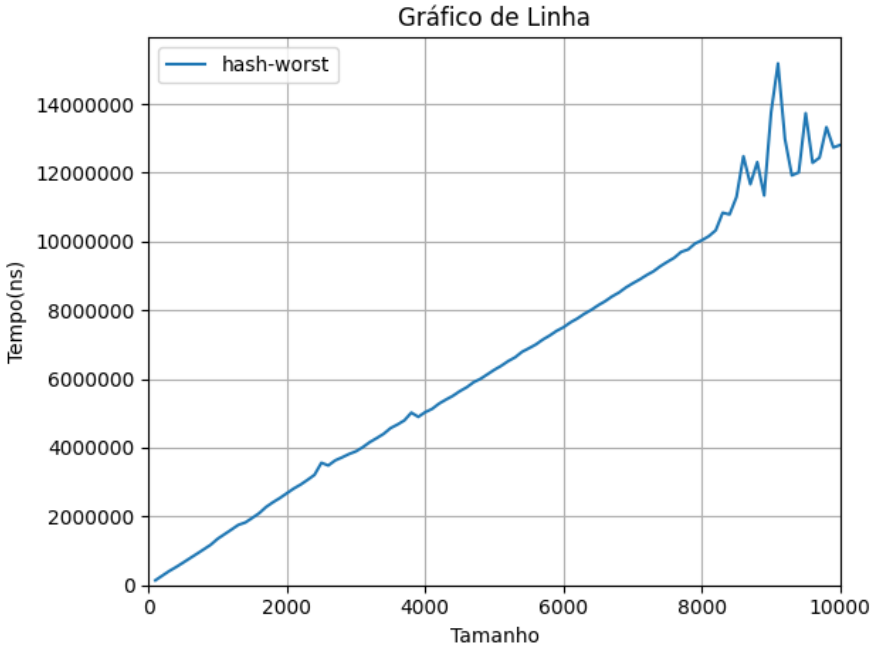

Figura 3.13: Tempo de execução da tabela de dispersão no pior caso ($\Theta(n)$). (tamanho do vetor x tempo) .

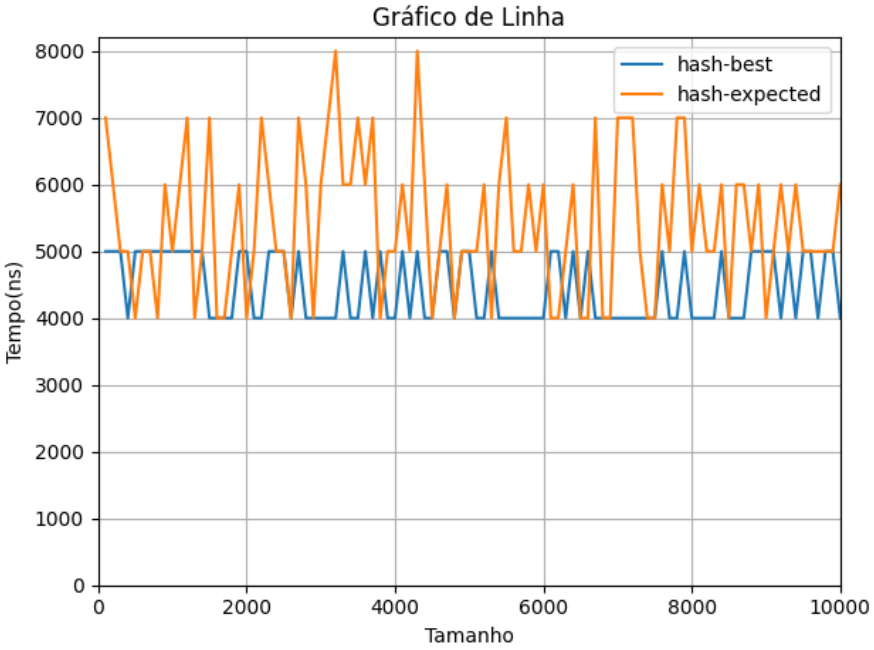

Figura 3.14: Tempo de execução da tabela de dispersão no melhor caso ($\Theta(1)$) e no tempo esperado ($\Theta(1)$). (tamanho do vetor x tempo) .

```
1  int hash_search(struct htable *t, int v) {
2      int i = 0;
3      struct block *b;
4
5      for (b = t->l[i]; b != NULL; b = b->next){
6          if(b->v == v) {
7              return 1;
8          }
9      }
10     return 0;
11 }
```

Figura 3.15: Código em C: hash$_s earch$.

4. Resultados

Aqui serão feitas comparações entre os algoritmos.

4.1 Todos vs Todos

No gráfico 4.1 percebemos com mais clareza a diferença entre todos eles e vemos que os piores casos da árvore binária e da hash, são bem piores que o restante, fazendo inclusive com que sumam.

4.2 Removendo piores casos da árvore binária e da tabela hash

No gráfico 4.2 conseguimos ver com mais clareza os casos com menores tempos e percebemos que o mais estável é a avl, já que os piores casos da árvore binária e da hash são lineares, porém, o melhor e o pior caso da tabela hash são extremamente rápidos, se memória não for um problema.

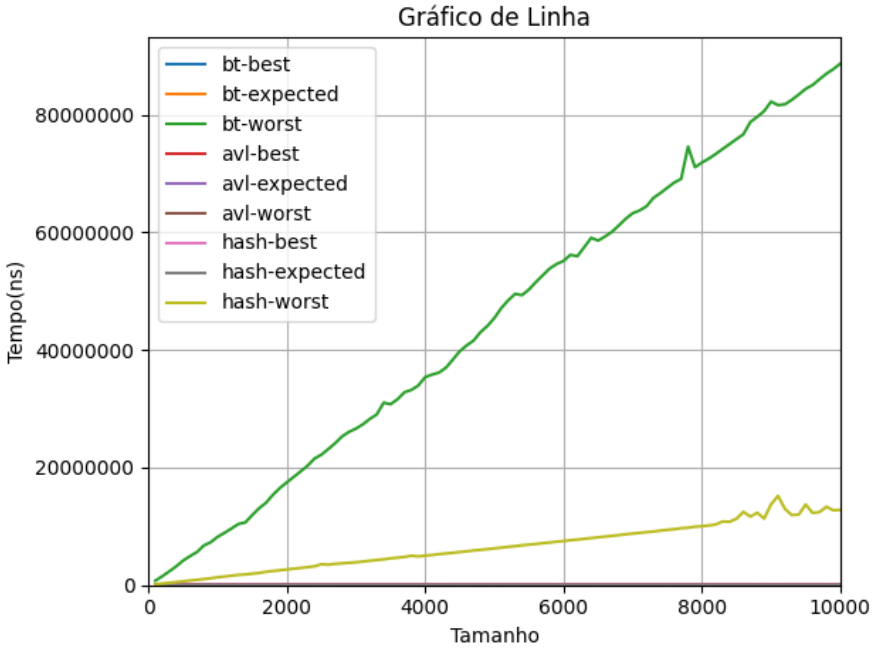

Figura 4.1: vs Todos (tamanho do vetor x tempo) .

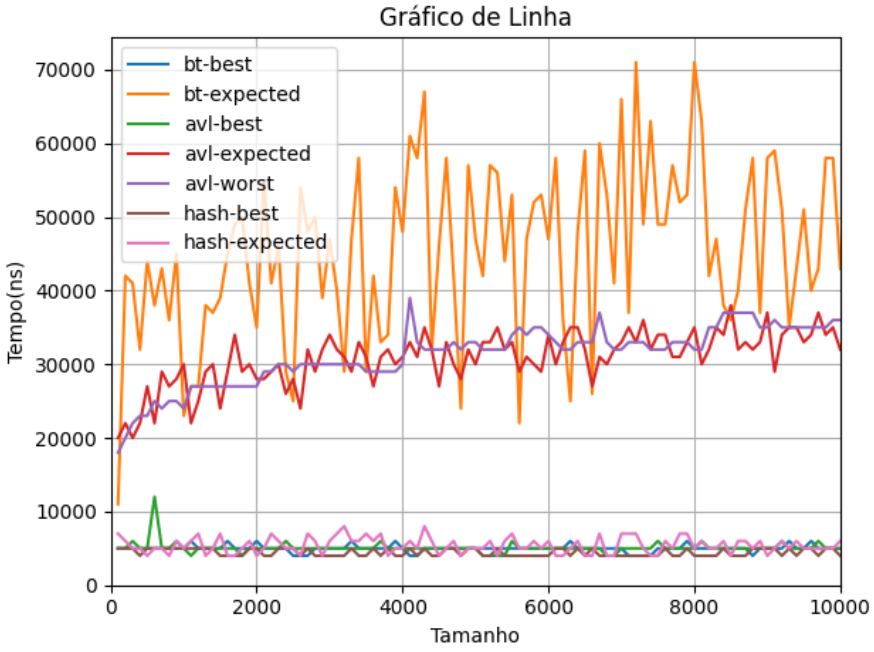

Figura 4.2: vs Todos sem os piores da binária e hash (tamanho do vetor x tempo) .

5. Conclusão

No fim das contas, as diferenças são poucas entre estas estruturas, mas podemos concluir que a tabela hash, quando memória não é uma preocupação, se torna a opção mais rápida, já que seu melhor caso e o caso esperado são constantes. A tabela hash também é mais simples de se implementar do que a avl.

Dito isso, a escolha do algoritmo a ser usado depende das limitações do hardware e do nível de complexidade dos algoritmos, sendo a árvore binária e a tabela hash os mais simples, e a avl mais complexa.

Portanto, é importante dominar essas estruturas para que quando a necessidade de uma delas aparecer, o programador possa escolher a melhor para a situação.