Universidade Federal do Rio Grande do Norte
Centro de Ensino Superior do Seridó
Departmento de Computação e Tecnologia
Curso de Bacharelado em Sistemas de Informação

Trabalho de Estrutura de Dados

Marlison Soares da Silva

Caicó-RN

Agosto de 2024

Marlison Soares da Silva

Avaliação 2

Trabalho apresentado a disciplina Estrutura de Dados do Departamento de Computação e Tecnologia da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção da nota da unidade II.

Orientador:
Prof. Dr. João Paulo de Souza Medeiros

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

Caicó-RN

Agosto de 2024

Avaliação 2

Autor: Marlison Soares da Silva
Orientador(a): Prof. Dr. João Paulo de Souza Medeiros

Resumo

O Trabalho em questão tem como objetivo estudar a busca de valores em árvores binárias quaisquer, ávores binárias balanceadas e tabela de dispersão (hash). Mais precisamente, o tempo de execução de cada uma em diferentes casos.

Palavras-chave: estrutura de dados; árvore binária balanceada; árvore binária; tabela de dispersão; tabela hash; tempo de busca.

Sumário

1 Trabalho Avaliativo II de Estrutura de Dados

1.1 Introdução

Neste trabalho, exploramos a análise de diversas estruturas de dados, com ênfase em árvores binárias, árvores binárias balanceadas e tabelas de dispersão (hash). A escolha dessas estruturas é motivada pela sua relevância e aplicabilidade em uma variedade de problemas práticos enfrentados na computação moderna. Estruturas de dados são fundamentais para a organização e manipulação de informações, e sua eficiência pode impactar significativamente o desempenho de algoritmos em tarefas como busca, inserção e remoção de dados.

A análise proposta busca entender como essas estruturas se comportam em diferentes contextos, levando em consideração tanto os melhores quanto os piores casos de desempenho. Essa abordagem nos permite identificar as vantagens e desvantagens de cada estrutura, proporcionando uma visão mais clara sobre quando e como utilizá-las de maneira eficaz.

Além de apresentar os conceitos teóricos que sustentam essas estruturas, o trabalho também se propõe a realizar uma avaliação prática. Essa avaliação é essencial para que possamos não apenas compreender as características intrínsecas de cada estrutura, mas também as implicações que elas têm em termos de eficiência e complexidade em cenários do mundo real. Através dessa combinação de teoria e prática, buscamos oferecer uma visão abrangente que possa ser útil tanto para estudantes que estão iniciando seus estudos em computação quanto para profissionais que desejam aprofundar seu conhecimento.

Espero que este trabalho sirva como um recurso valioso, contribuindo para o entendimento das estruturas de dados e suas aplicações, e que inspire novas investigações e práticas na área da computação. Acreditamos que uma compreensão sólida dessas estruturas é essencial para o desenvolvimento de soluções eficientes e inovadoras em um campo em constante evolução.

2 Análises dos algoritmos

Capítulo voltado para a análise e comparação da busca em algumas estruturas de dados. As estruturas estudadas a seguir são: árvores binárias (não necessariamente balanceadas), árvores balanceadas (AVL) e tabela de dispersão (hash).

2.1 Árvore Binária

A Árvore Binária é uma estrutura de dados formada por "nós", onde cada nó tem no máximo dois filhos. Nela, todos os nós na subárvore esquerda têm valores menores que o nó e todos os nós na subárvore direita têm valores maiores que o nó.

Essa estrutura por si só é pouco previsível em relação a suas vantagens e desvantagens. Quando balanceada, isto é, tem, em cada nó, uma quantidade semelhante de filhos em ambos os lados, o seu uso é bastante eficiente em relação a várias outras estruturas de dados. No entanto, quando muito desbalanceada, ou seja, tem mais nós de um lado do que de outro, pode ser uma desvantagem usá-la, em relação a outras estruturas, devido seu esquema de busca.

A busca em uma árvore binária acontece da seguinte forma:

Início da busca: Comece na raiz da árvore.

Comparação: Compare o valor procurado com o valor do nó atual.

1. Se o valor procurado é menor, vá para a subárvore esquerda.

2. Se o valor procurado é maior, vá para a subárvore direita.

3. Se o valor procurado é igual ao valor do nó atual, a busca foi bem-sucedida e o nó foi encontrado.

Continuação: Repita o processo na subárvore selecionada (esquerda ou direita) até encontrar o nó ou até atingir uma "folha"(caso em que o valor não está presente na árvore).

Abaixo temos a implementação da árvore binária e da busca binária na linguagem C.

Código Árvore Binária:

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

// estrutura de no
struct node {
    int v;
    struct node* l;
    struct node* r;
};
```

```
11
12 // criacao de um no (sem filhos)
13 struct node* new_node(int v) {
14     struct node* n = (struct node*)malloc(sizeof(struct node));
15     n->v = v;
16     n->l = NULL;
17     n->r = NULL;
18     return n;
19 }
20
21 // insercao de no na arvore
22 void btinsert(struct node** r, struct node* n) {
23     if (*r != NULL) {
24         if ((*r)->v > n->v)
25             return btinsert(&(*r)->l, n);
26         else
27             return btinsert(&(*r)->r, n);
28     }
29     (*r) = n;
30 }
```

Código 2.1: Código Árvore Binária em C

Código Busca Binária:

```
1 struct node* search(struct node* r, int v) {
2     if (r != NULL) {
3         if (v < r->v)
4             return search(r->l, v);
5         if (r->v < v)
6             return search(r->r, v);
7     }
8     return r;
9 }
```

Código 2.2: Código Busca Binária em C

2.1.1 Árvore Binária - Melhor Caso

O melhor caso da Árvore Binária (não-balanceada) acontece quando o elemento buscado é a raiz da árvore, ou seja, o primeiro elemento testado.

Tempo de execução da Árvore Binária no Melhor Caso:

Neste caso, independente da quantide de elementos presentes na árvore, o tempo de busca será sempre o mesmo, que é o tempo de teste de 1 elemento. Fazendo do melhor caso **constante**.

Resposta Assintótica:

$O(1)$

Gráfico Árvore Binária no Melhor Caso:

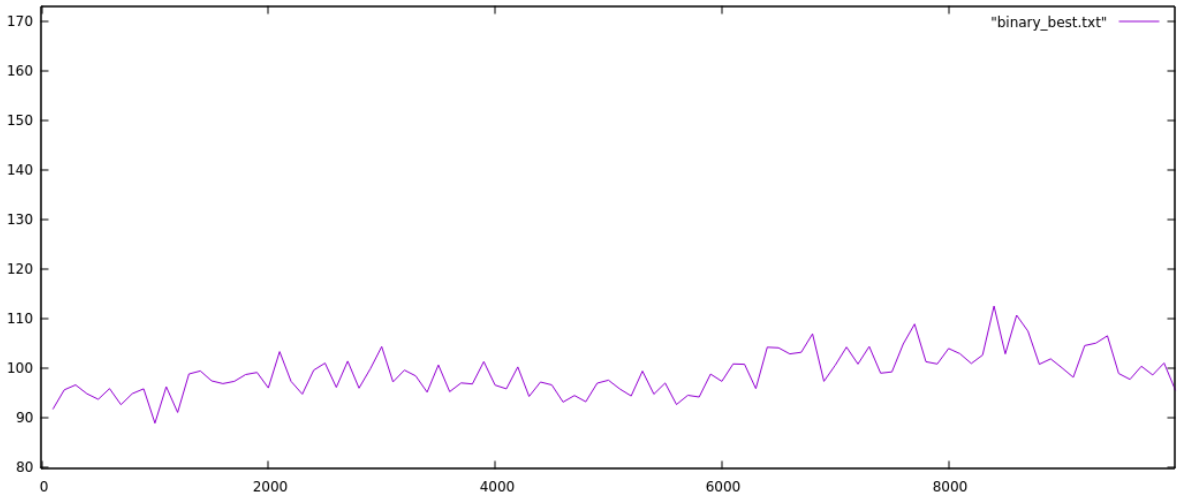

Figura 1: Tempo do melhor caso da busca em árvore binária, em nanossegundos, em função do número de elementos (n) na árvore.

2.1.2 Árvore Binária - Pior Caso

No pior caso da Árvore Binária, os elementos estão inseridos de forma completamente desbalanceada para apenas um lado, desde a raiz. Ou seja, a altura da árvore é proporcional ao número de elementos na árvore. Se há n elementos na árvore, a altura dela também será n. No entanto, para ser realmente o pior caso, depois de percorrida toda a árvore, o elemento não é encontrado.

Tempo de execução da Árvore Binária no Pior Caso:

Neste caso, quanto mais elementos, maior o tempo de busca, de forma proporcional. Essa característica faz o tempo de busca ser linear em relação ao número de elementos n.

Resposta Assintótica:

$O(n)$

Gráfico Árvore Binária no Pior Caso:

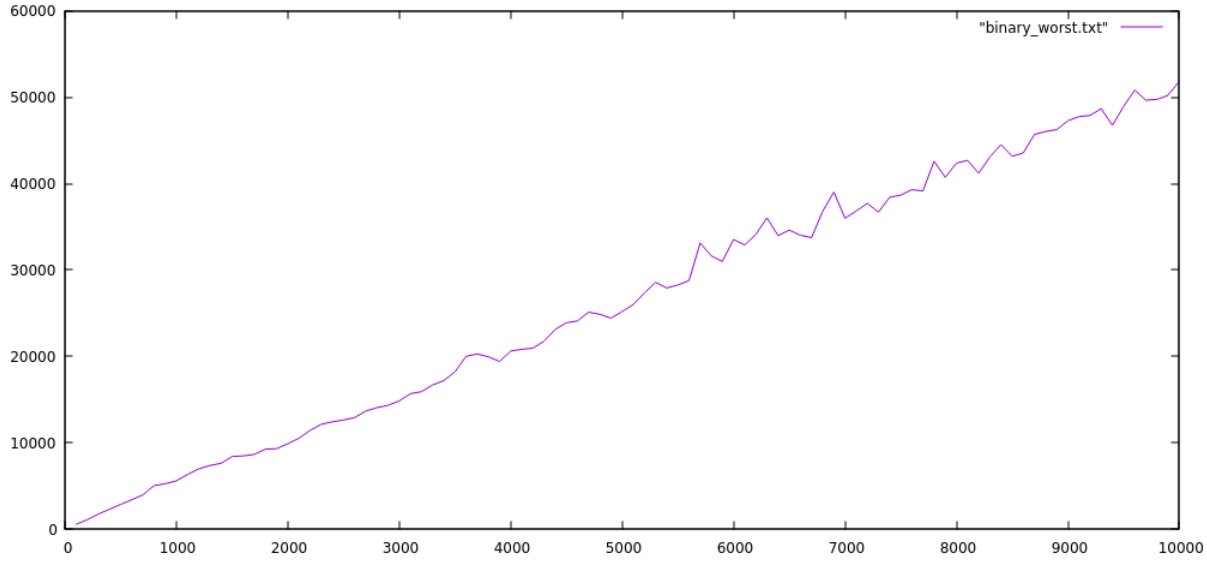

Figura 2: Tempo esperado do pior caso da Árvore Binária, em nanossegundos, em função do número de elementos (n) na lista.

2.1.3 Árvore Binária - Caso Médio

O caso médio da busca em uma árvore binária não-balanceada é a uma variação da busca por elementos quaisquer em diversos estados de balanceamento diversos, desde totalmente desbalanceada até completamente balanceada. Ou seja, é possível buscar um elemento em qualquer altura da árvore e em árvores desde O(log n) a O(n). No entanto,

Tempo de execução da Árvore Binária no Caso Médio:

Neste caso, temos algumas possibilidades para o tempo de execução:

1. constante: quando o elemento escolhido é a raiz;

2. linear: quando os elementos são inseridos em ordem e o elemento buscado é a folha da árvore ou após ela;

3. logarítmico: quando a árvore está balanceada, ou quase.

Como os elementos inseridos e buscado são aleatórios, temos algumas consequências: uma árvore parcialmente balanceada e baixa probabilidade dos elementos serem inseridos em ordem e do buscado ser a primeiro. Desta forma, o caso médio tende a ser logarítmico.

Resposta Assintótica:

$O(log(n))$

Grafico Árvore Binária Caso Médio:

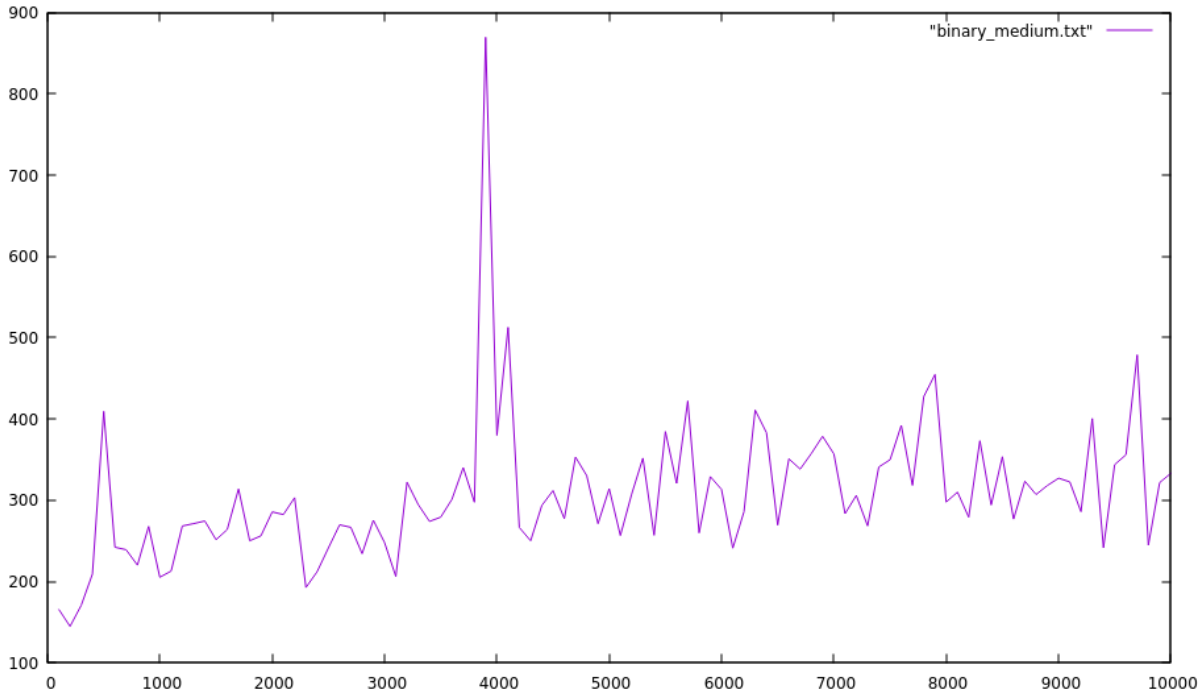

Figura 3: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

Como supracitado, o gráfico tende ao *log* da altura. Observe que ele inicia por volta de 150 nanossegundos e termina acima 300 nanossegundos. Possuindo uma curva de crescimento leve, mesmo que oscilante.

2.1.4 Comparação Árvore Binária

Gráfico de Comparação:

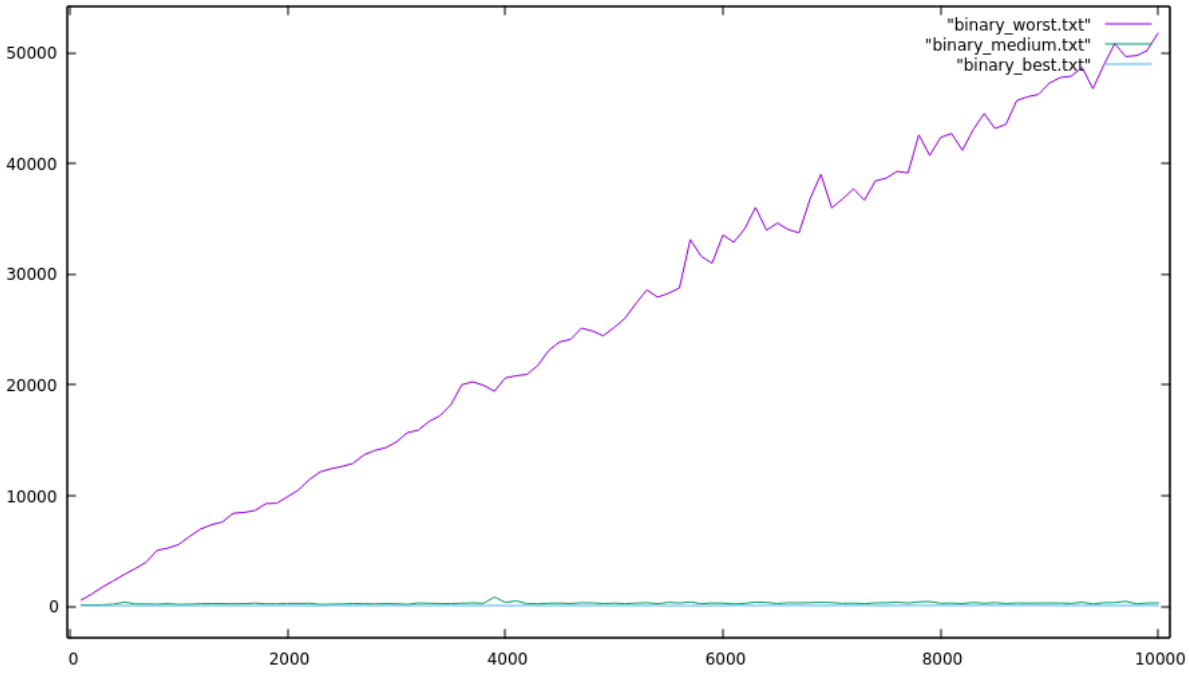

Figura 4: Comparação entre o melhor caso, o pior caso e o caso médio da Árvore Binária

Como esperado, o caso médio está muito distante do pior caso.

Médio e Melhor:

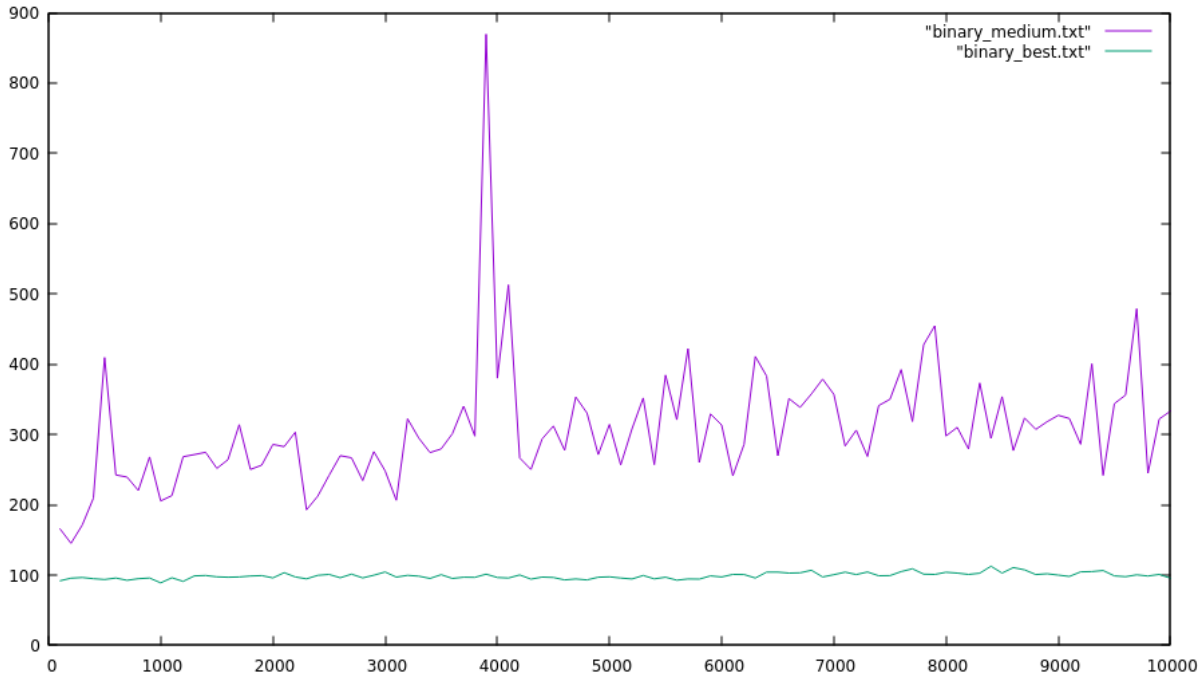

Figura 5: Comparação entre o melhor caso e o caso médio da Árvore Binária

2.2 Árvore Binária Balanceada (AVL)

Uma árvore AVL é um tipo de árvore binária de busca que é auto-balanceada. As árvores AVL mantêm um equilíbrio estrito, o que garante que a altura da árvore seja sempre proporcional ao logaritmo do número de nós. Isso é feito para garantir operações eficientes de busca, inserção e remoção.

A Árvore AVL é uma estrutura de dados formada por "nós", onde cada nó tem no máximo dois filhos. Nela, todos os nós na subárvore esquerda têm valores menores que o nó e todos os nós na subárvore direita têm valores maiores que o nó. Além disso, para ser chamada de AVL, deve satisfazer à condição de balanceamento:

Para cada nó na árvore, a diferença entre as alturas das suas subárvores esquerda e direita (chamada de fator de balanceamento) deve ser no máximo 1.

Ou seja:

$|h_l - h_r| <= 1$

A principal vantagem das árvores AVL é que o balanceamento garante que a altura da árvore seja mantida em O(log n), onde n é o número de nós na árvore. Isso é uma vantagem para a busca binária, fazendo o tempo de execução para a busca ser proporcional à altura da árvore.

A busca em uma árvore AVL funciona de forma semelhante à busca em uma árvore binária de busca:

Início da busca: Comece na raiz da árvore.

Comparação: Compare o valor procurado com o valor do nó atual.

1. Se o valor procurado é menor, vá para a subárvore esquerda.

2. Se o valor procurado é maior, vá para a subárvore direita.

3. Se o valor procurado é igual ao valor do nó atual, a busca foi bem-sucedida e o nó foi encontrado.

Continuação: Repita o processo na subárvore selecionada (esquerda ou direita) até encontrar o nó ou até atingir uma "folha"(caso em que o valor não está presente na árvore).

Abaixo temos a implementação da árvore AVL e da busca binária na linguagem C.

Código Árvore AVL:

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

// estrutura do no
struct node {
    int v;
    unsigned int h;
    struct node* l;
    struct node* r;
    struct node* p;
};

// criacao de um no
struct node* new_node(int v) {
    struct node* n = (struct node*)malloc(sizeof(struct node));
    n->h = 0;
    n->v = v;
    n->l = NULL;
    n->r = NULL;
    n->p = NULL;
    return n;
}

// retorna a altura de um no
int getHeight(struct node *n) {
    if (n == NULL)
        return 0;
    return n->h;
}

// atualiza a altura de um no a partir da de seus filhos
void update_height(struct node** r) {
    int hl = getHeight((*r)->l);
    int hr = getHeight((*r)->r);
    (*r)->h = hl > hr ? hl + 1: hr + 1;
}

// rotacao    direita
void rotateRight(struct node** n) {
```

```
43      struct node* aux = (*n);
44      (*n) = (*n)->l;
45      (*n)->p = aux->p;
46      aux->p = (*n);
47      aux->l = (*n)->r;
48      (*n)->r = aux;
49      update_height(&aux);
50
51  }
52
53  // rotacao    esquerda
54  void rotateLeft(struct node** n) {
55      struct node* aux = (*n);
56      (*n) = (*n)->r;
57      (*n)->p = aux->p;
58      aux->p = (*n);
59      aux->r = (*n)->l;
60      (*n)->l = aux;
61      update_height(&aux);
62      update_height(n);
63  }
64
65  // balanceamento
66  void avl(struct node** r) {
67      int hl = getHeight((*r)->l);
68      int hr = getHeight((*r)->r);
69      if (abs(hl - hr) > 1) { // desbalanceado
70          if (hl > hr) {
71              if (getHeight((*r)->l->l) > getHeight((*r)->l->r)) {
72                  // CASO 2 (esquerda-esquerda)
73                  rotateRight(r);
74              } else {
75                  // CASO 4 (esquerda-direita)
76                  rotateLeft(&((*r)->l));
77                  rotateRight(r);
78              }
79          } else {
80              if (getHeight((*r)->r->r) > getHeight((*r)->r->l)) {
81                  // CASO 1 (direita-direita)
82                  rotateLeft(r);
83              } else {
84                  // CASO 3 (direita-esquerda)
85                  rotateRight(&((*r)->r));
```

```
86                  rotateLeft(r);
87              }
88          }
89      }
90  }
91
92  // insercao de um no na arvore, balanceando
93  void btinsert(struct node** r, struct node* n) {
94      if((*r) != NULL) {
95          if ((*r)->v > n->v) {
96              btinsert(&((*r)->l), n);
97              (*r)->l->p = (*r);
98          } else {
99              btinsert(&((*r)->r), n);
100             (*r)->r->p = (*r);
101         }
102     } else {
103         *r = n;
104     }
105     update_height(r);
106     avl(r);
107 }
```

Código 2.3: Código Árvore AVL em C

Código Busca Binária:

```
1  struct node* search(struct node* r, int v) {
2      if (r != NULL) {
3          if (v < r->v)
4              return search(r->l, v);
5          if (r->v < v)
6              return search(r->r, v);
7      }
8      return r;
9  }
```

Código 2.4: Código Busca Binária em C

2.2.1 Árvore AVL - Melhor Caso

Assim como em uma árvore binária não-balanceada, o melhor caso da Árvore AVL acontece quando o elemento buscado é a raiz da árvore, ou seja, o primeiro elemento

testado.

Tempo de execução do Árvore AVL no Melhor Caso:

Neste caso, independente da quantide de elementos presentes na árvore, o tempo de busca será sempre o mesmo, que é o tempo de teste de 1 elemento. Fazendo do melhor caso **constante**.

Resposta Assintótica:

$O(1)$

Gráfico Árvore AVL no Melhor Caso:

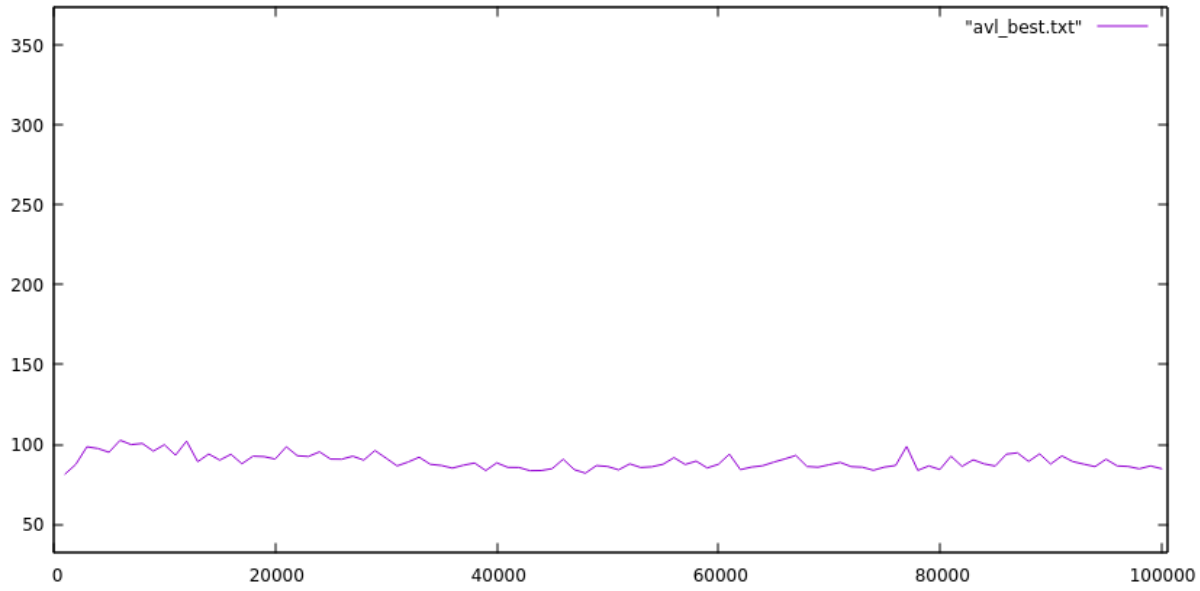

Figura 6: Tempo do melhor caso do Árvore AVL, em nanossegundos, em função do número de elementos (n) na lista.

2.2.2 Árvore AVL - Pior Caso

O pior caso da busca em uma Árvore AVL acontece quando o elemento buscado não está na lista. Passando por toda a árvore, desde a raiz até suas folhas.

Tempo de execução do Árvore AVL no Pior Caso:

Como neste caso toda a árvore é sempre percorrida, o tempo de execução é proporcional à altura da árvore. Ou seja, como a altura cresce de forma logarítmica em função de n, o tempo de execução da busca ocorrerá do mesmo modo.

Resposta Assintótica:

$O(log(n))$

Gráfico Árvore AVL no Pior Caso:

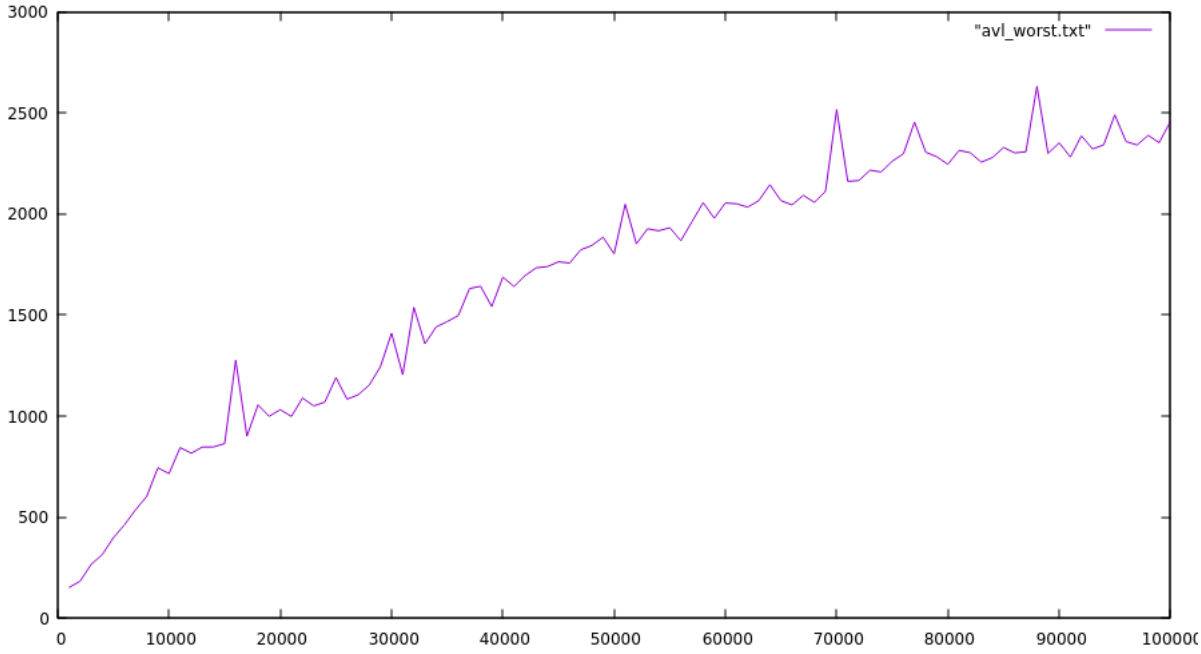

Figura 7: Tempo esperado do pior caso da Árvore AVL, em nanossegundos, em função do número de elementos (n) na lista.

2.2.3 Árvore AVL - Caso Médio

O caso médio da busca em uma árvore AVL ocorre quando é incerta a posição do valor buscado dentro da árvore. Ou seja, o valor pode ser um valor qualquer, desde a raiz até uma folha.

Tempo de execução do Árvore AVL no Caso Médio:

Como a probabilidade do valor buscado ser sempre o primeiro é baixa, a Árvore AVL tende a ser, geralmente, logarítmica.

Resposta Assintótica:

$O(log(n))$

Grafico Árvore AVL Caso Médio:

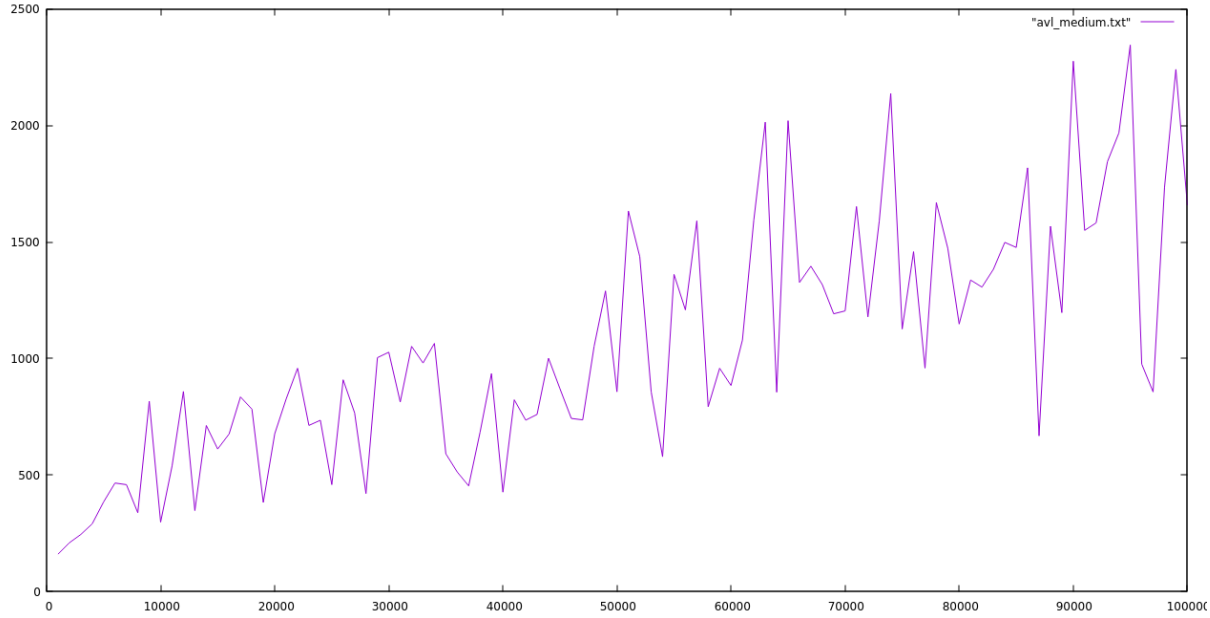

Figura 8: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

Novamente, como supracitado, o crescimento do gráfico tende a ser logarítmico. No entanto, por receber valores quaisquer aleatórios, a variação é enorme.

2.2.4 Comparação Árvore AVL

Gráfico de Comparação:

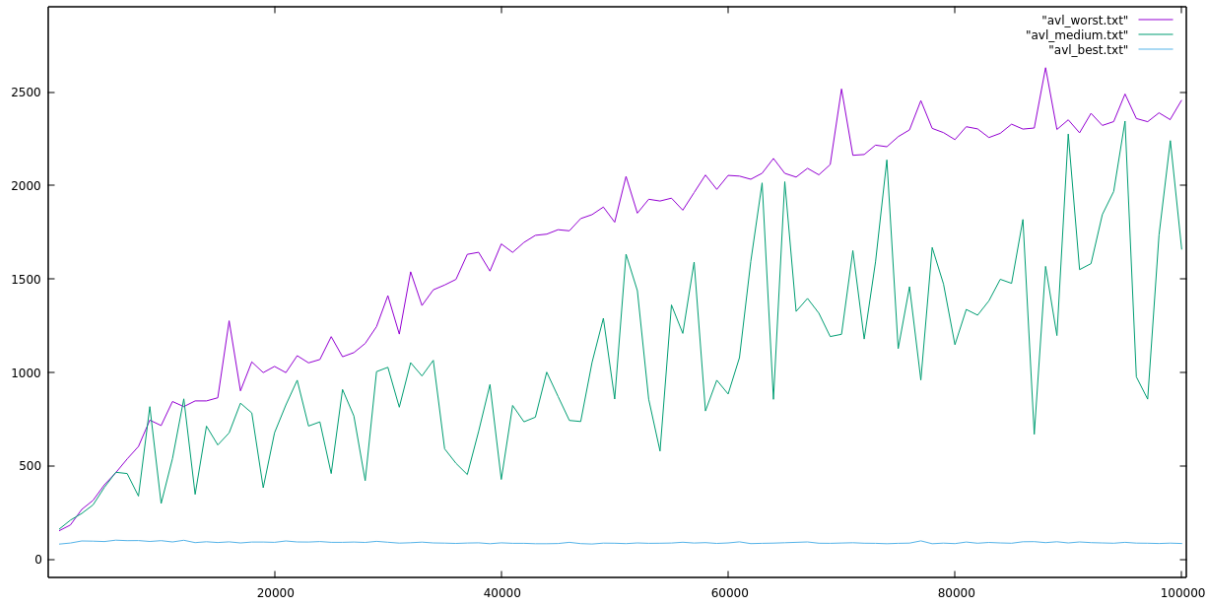

Figura 9: Comparação entre o melhor caso, o pior caso e o caso médio do Árvore AVL

Observando como começa e termina cada gráfico, torna-se atestada cada análise realizada. Nas buscas realizadas, os tempos de execuções de cada busca se mantêm limitados, superiormente, pelo pior caso e, inferiormente, pelo melhor caso.

2.3 Tabela de Dispersão

Uma tabela de dispersão (ou tabela hash) é uma estrutura de dados que associa chaves a valores de forma eficiente. Ela é amplamente utilizada quando precisamos de operações rápidas de inserção, busca e remoção.

Neste documento, ela segue algumas regras:

1. cada valor adicionado à tabela de dispersão segue à função hash para decidir seu índice;

2. a tabela não pode ter 1 ou mais itens por índice - fator de carga -, ou seja, se houver 4 índices, o limite de valores deve ser 3;

3. sempre que o fator de carga for 1 ou maior, a tabela deve ser reconstruída (re-hash), nesta situação, o número de índices deve dobrar.

Para a busca de um valor na tabela de dispersão, o valor buscado deve passar pela função hash, a qual retorna seu índice. Em seguida, é buscado o valor dentro da lista de valores (itens) com aquele índice.

Código Tabela de Dispersão:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// estrutura de um item da tabela
struct item {
    int value;
    struct item* next;
};

// estrutura da tabela
struct hashTable {
    unsigned int m; // Tamanho da tabela
    unsigned int n; // Quantidade de Elementos
    struct item** items;
};

// funcao hash
unsigned int func_hash(int value, int m) {
    return abs(value % m);
}
```

```
22
23 void insert(struct hashTable* hashTable, int value);
24 void reHash(struct hashTable* hashTable);
25
26 // Funcao Re-hash - Para aumentar a quantidade de espacos na tabela e
       evitar que o tempo nao seja constante
27 void reHash(struct hashTable* hashTable) {
28     unsigned int old_m = hashTable->m;
29     struct item** old_items = hashTable->items;
30
31     hashTable->m *= 2;
32     hashTable->n = 0;
33     hashTable->items = (struct item**)malloc(sizeof(struct item*) *
           hashTable->m);
34
35     for(int i = 0; i < hashTable->m; i++) {
36         hashTable->items[i] = NULL;
37     }
38
39     for(int i = 0;i < old_m; i++) {
40         struct item* aux = old_items[i];
41         while(aux != NULL) {
42             // Insercao dos elementos
43             insert(hashTable, aux->value);
44
45             // Liberar memoria alocada
46             struct item* toFree = aux;
47             aux = aux->next;
48             free(toFree);
49         }
50     }
51     free(old_items);
52 }
53
54 // Calcula o fator de carga
55 float loadFactor(struct hashTable* hashTable) {
56     return (float)hashTable->n / (float)hashTable->m;
57 }
58
59 // Criando um novo item
60 struct item* create_item(int value) {
61     struct item* item = (struct item*)malloc(sizeof(struct item));
62     item->value = value;
```

```
63      item->next = NULL;
64      return item;
65  }
66
67  // Inicializa a Tabela Hash
68  struct hashTable* create_table(unsigned int m) {
69      struct hashTable* hashTable = (struct hashTable*)malloc(sizeof(
            struct hashTable));
70      hashTable->m = m;
71      hashTable->n = 0;
72      hashTable->items = (struct item**)malloc(sizeof(struct item*) * m);
73      for(int i = 0; i < m; i++) {
74          hashTable->items[i] = NULL;
75      }
76      return hashTable;
77  }
78
79  // Insere um item em um indice da tabela
80  void insert_item(struct item* item, int value) {
81      if (item->next != NULL)
82          insert_item(item->next, value);
83      else
84          item->next = create_item(value);
85  }
86
87  // Insere um item na tabela
88  void insert(struct hashTable* hashTable, int value) {
89      unsigned int key = func_hash(value, hashTable->m);
90
91      if (loadFactor(hashTable) < 1) {
92          if(hashTable->items[key] == NULL) {
93              hashTable->items[key] = create_item(value);
94          } else {
95              insert_item(hashTable->items[key], value);
96          }
97          hashTable->n++;
98      } else {
99          reHash(hashTable);
100         insert(hashTable, value);
101     }
102 }
```

Código 2.5: Código Tabela de Dispersão em C

Código Busca na Tabela de Dispersão:

```
// Procurando um elemento em um indice da tabela
struct item* search_item(struct item* item, int value) {
    if (item != NULL) {
        if (item->value == value) {
            return item;
        }
        search_item(item->next, value);
    }
    return NULL;
}

// Procurando elementos na tabela Hash
struct item* search(struct hashTable* hashTable, int value) {
    unsigned int key = func_hash(value, hashTable->m);
    return search_item(hashTable->items[key], value);
}
```

Código 2.6: Código Busca na Tabela de Dispersão em C

2.3.1 Tabela de Dispersão - Melhor Caso

O melhor caso para busca da tabela de dispersão é buscar por um elemento onde seu índice está vazio, ou buscar por qualquer valor de uma tabela perfeitamente dispersa (máximo de 1 item em cada índice).

Tempo de execução da Tabela de Dispersão no Melhor Caso:

Neste caso, como a função hash possui tempo de execução constante e a busca entre os itens de um índice verifica 0 ou 1 valores, a busca completa possui tempo **constante**.

Resposta Assintótica:

$O(1)$

Gráfico Tabela de Dispersão no Melhor Caso:

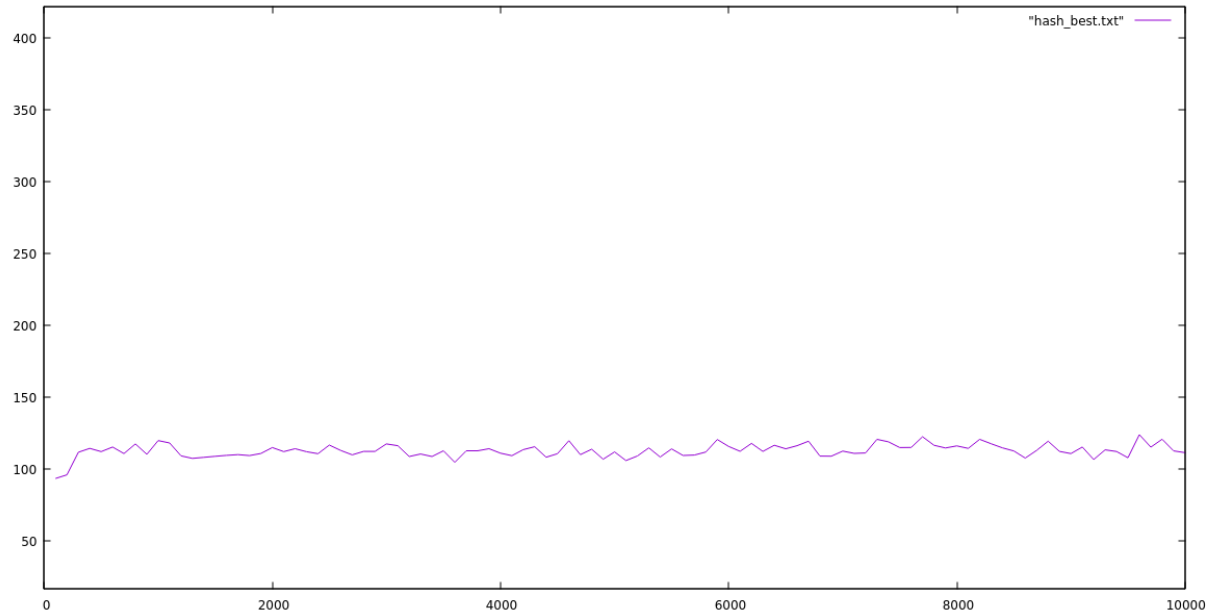

Figura 10: Tempo do melhor caso da Tabela de Dispersão, em nanossegundos, em função do número de elementos (n) na lista.

2.3.2 Tabela de Dispersão - Pior Caso

No pior caso da busca em uma tabela de dispersão, todos os valores da tabela estão listados em um único índice e o valor buscado se encaixaria neste índice, no entanto, não está nele. Desta forma, se a tabela tiver n índices e todos os valores estiverem todos em um único índice, o número de itens verificados é proporcional ao número de valores contidos na tabela.

Tempo de execução da Tabela de Dispersão no Pior Caso:

O tempo de execução da busca na tabela de dispersão, no pior caso, é proporcional ao número de valores na tabela. Ou seja, o tempo é de ordem **linear**.

Resposta Assintótica:

$O(n)$

Gráfico Tabela de Dispersão no Pior Caso:

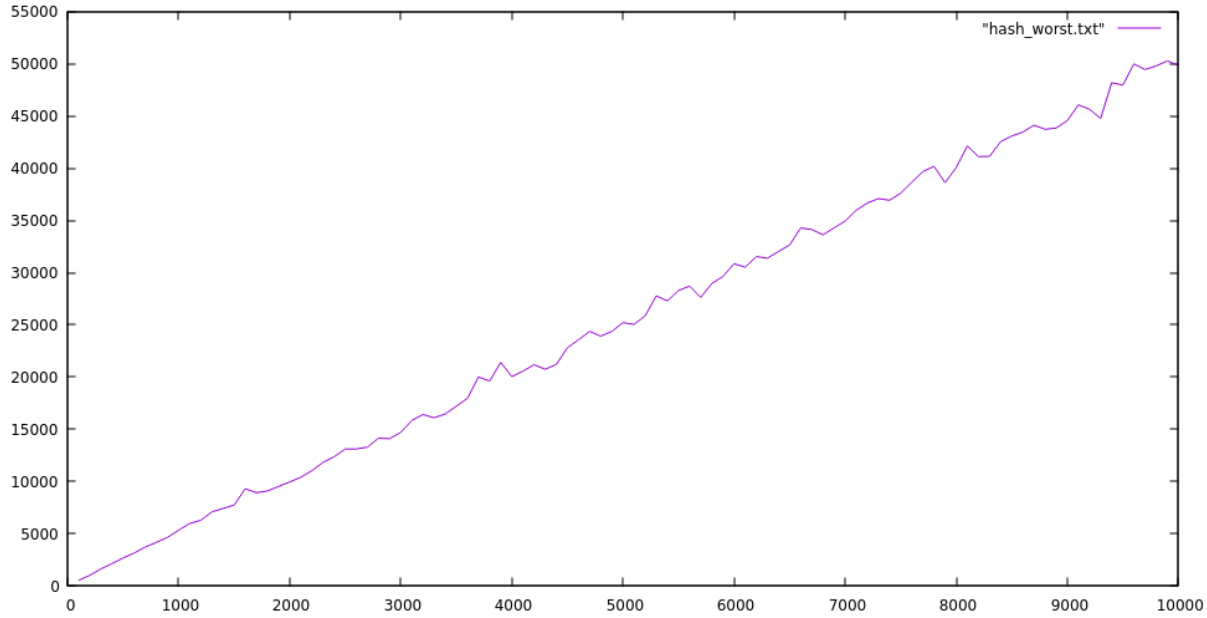

Figura 11: Tempo esperado do pior caso da Tabela de Dispersão, em nanossegundos, em função do número de elementos (n) na lista.

2.3.3 Tabela de Dispersão - Caso Médio

A formação de uma tabela de dispersão pode variar bastante a depender do tipo de valor inserido e da função hash. No nosso caso, a função hash atribui os valores aos índices a partir do resto da divisão pelo número total de índices na tabela. Ou seja, se houver 8 índices, a função hash deve pegar o valor e atribuir ao índice que corresponda ao resto da divisão do valor pelo número 8. Logo, algo entre 0 e 7.

Nesta situação, se os números inseridos forem sempre certos múltiplos, por exemplo: 10, 20, 30, 40 e 50, inseridos com a função hash em questão, irão todos serem inseridos no índice 2. No entanto, geralmente, inserindo números fora de um certo padrão determinado para gerar um pior caso para a função hash específica, o número de valores em um mesmo índice não é sempre máximo.

Tempo de execução da Tabela de Dispersão no Caso Médio:

Com os valores inseridos e buscado na tabela de dispersão sendo aleatórios e não seguindo um padrão contraditório à função hash, o tempo de busca na tabela deve ser, no geral, **constante**.

Resposta Assintótica:

$O(n)$

Grafico Tabela de Dispersão Caso Médio:

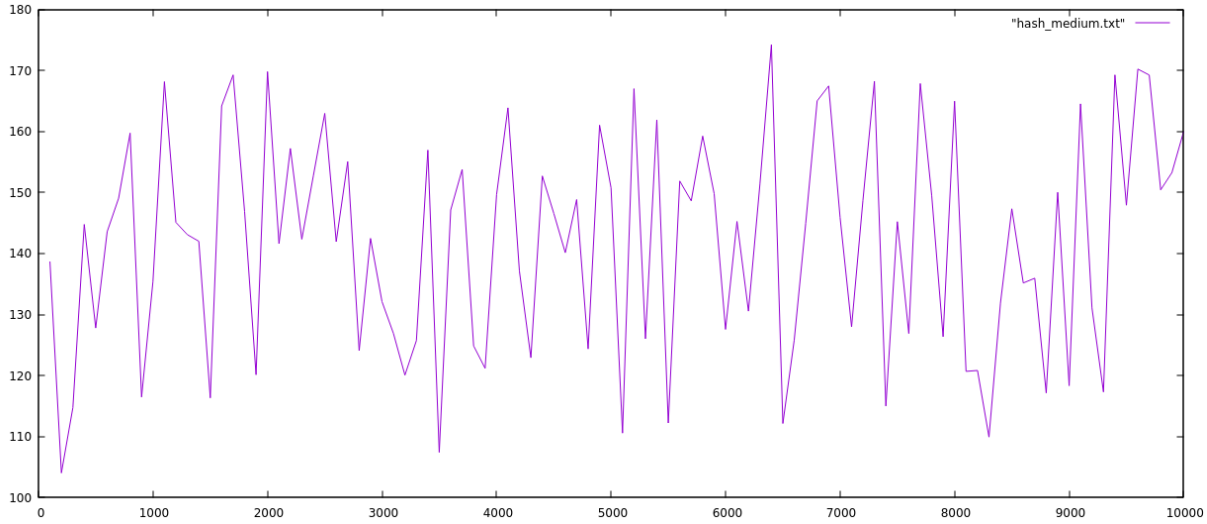

Figura 12: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

2.3.4 Comparação Tabela de Dispersão

Gráfico de Comparação:

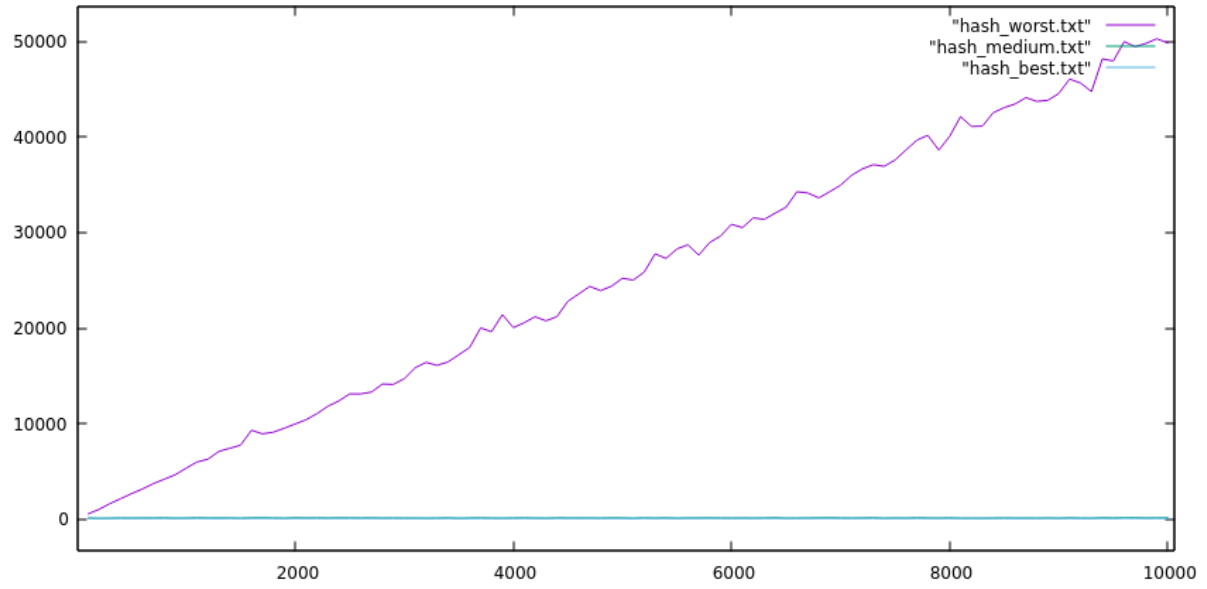

Figura 13: Comparação entre o melhor caso, o pior caso e o caso médio da Tabela de Dispersão

Como esperado, o caso médio está bem mais próximo do melhor caso do que do pior.

Médio e melhor

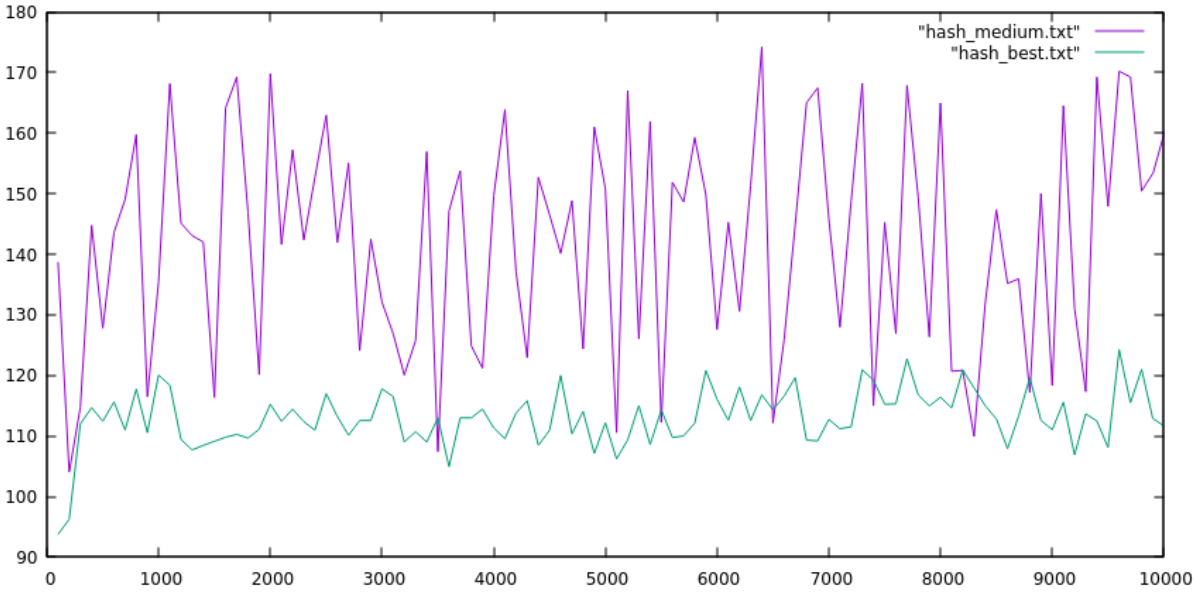

Figura 14: Comparação entre o melhor caso e o caso médio da Tabela de Dispersão

3 Comparações Gráficas

Após análises individuais de cada estrutura, iremos cruzar as três estruturas estudadas em seus casos.

Piores casos

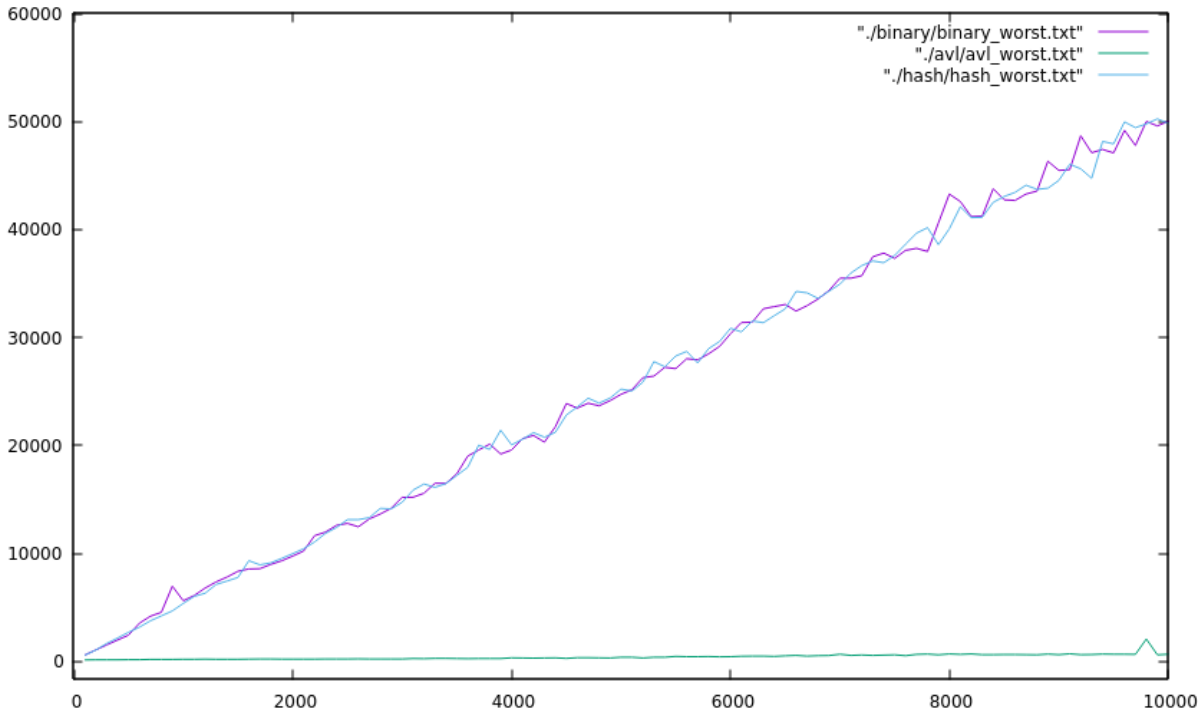

Figura 15: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

Se a chance de cair no pior caso é alta, ou recorrente, a estrutura recomendada é, sem dúvidas, a Árvore AVL.

Melhores casos

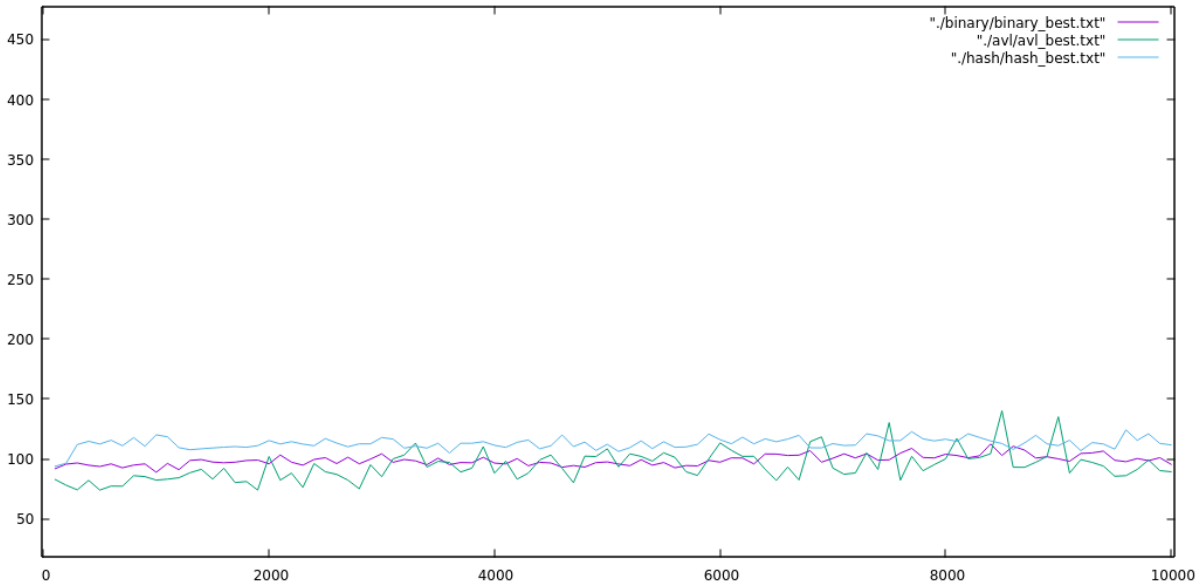

Figura 16: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

Se a chance de cair no melhor caso é alta, ou recorrente, a estrutura recomendada é irrelevante, apesar de visualmente algumas linhas do gráfico parecerem estar majoritariamente acima ou abaixo das outras. Isso porque o tempo é de ordem constante para as três estruturas, ou seja, nenhuma vai aumentar ao longo do tempo.

Casos médio

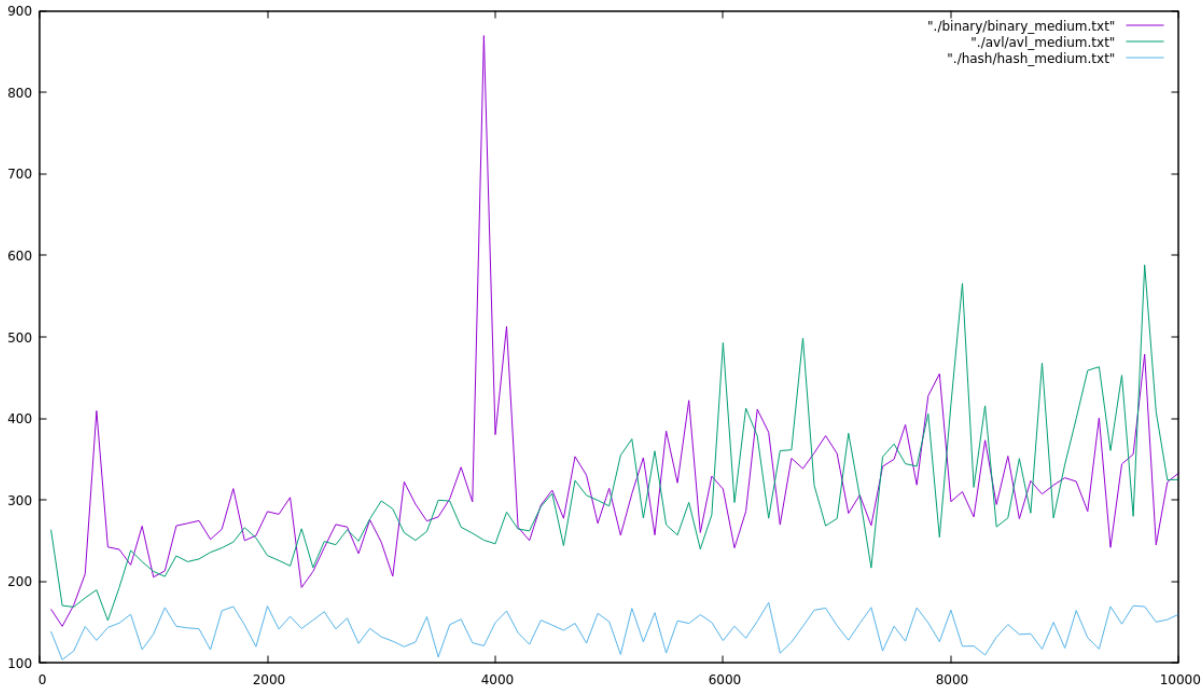

Figura 17: Tempo esperado, em nanossegundos, em função do número de elementos (n) na lista.

Analisando os casos médios, percebe-se que a AVL e a binária comum não se diferenciaram muito nestas execuções de código, mas a tabela hash continuou constante. Isto indica que, se a chance da busca em uma tabela de dispersão cair no pior caso for baixa, ou seja, houver uma boa função de dispersão para os dados inseridos, então a tabela de dispersão é a indicada.

4 Conclusão

O presente trabalho avaliativo sobre Estruturas de Dados proporcionou uma análise abrangente e prática das diferentes abordagens utilizadas na busca de valores em diversas estruturas, como árvores binárias, árvores binárias balanceadas e tabelas de dispersão. Através da investigação dos melhores e piores casos de desempenho, foi possível identificar as características intrínsecas de cada estrutura, bem como suas vantagens e desvantagens em cenários do mundo real.

Os resultados obtidos demonstraram que, embora cada estrutura tenha seu próprio conjunto de benefícios, a escolha da estrutura mais adequada depende do contexto específico em que será aplicada. Em termos de desempenho, as árvores binárias balanceadas, como as árvores AVL, mostraram-se superiores em situações onde a eficiência na busca é crucial, especialmente em cenários que exigem operações frequentes de inserção e remoção, pois mantêm um equilíbrio que garante tempos de busca logarítmicos.

Por outro lado, as tabelas de dispersão (hash) se destacaram em casos onde a rapidez na busca é prioritária, oferecendo tempos de acesso constantes em média, desde que a função de hash seja bem projetada e a tabela não esteja excessivamente carregada. No entanto, é importante considerar que, em situações de colisões frequentes, o desempenho pode ser comprometido.

Além disso, a análise dos tempos de execução revelou que, em situações onde a eficiência é crucial, a compreensão das complexidades envolvidas é fundamental para a implementação de soluções eficazes. A árvore binária simples, embora menos eficiente em termos de tempo de busca em comparação com as estruturas balanceadas e hash, pode ser adequada em cenários onde a simplicidade e a facilidade de implementação são mais valorizadas.

A combinação de teoria e prática ao longo do trabalho não apenas reforçou os conceitos abordados, mas também destacou a importância de uma formação sólida em estruturas de dados para estudantes e profissionais da área de computação. Esperamos que este estudo sirva como um recurso valioso, incentivando novas investigações e práticas que contribuam para o avanço do conhecimento em computação e para o desenvolvimento de soluções inovadoras em um campo em constante evolução. A compreensão profunda

das estruturas de dados é, sem dúvida, um pilar essencial para a construção de sistemas eficientes e robustos.