



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

Análise de tempo de execução de algoritmos de árvore binária e tabela de dispersão

Lucas Mateus da Silva

Orientador: Prof. Dr. João Paulo de Souza Medeiros

Relatório Técnico apresentado ao Curso de Bacharelado em Sistemas de Informação como parte dos requisitos para aprovação na disciplina de Estrutura de dados.

Caicó, RN, 29 de junho de 2023

Resumo

Trabalho desenvolvido com o objetivo de averiguar os tempos de execução dos algoritmos: árvore binária, árvore balanceada(AVL) e tabela de dispersão(Hash). Também fazendo a análise assintótica e visualização gráfica. Todos os algoritmos mencionados neste documento podem ser encontrados no repositório mencionado ao final deste documento.

Keywords: Algoritmos; Árvore; Complexidade.

Sumário

Lista de Figuras	3
1 Árvore Binária	4
1.1 Complexidade	4
1.1.1 pior caso	4
1.1.2 médio Caso	4
1.1.3 Melhor caso	4
1.2 Implementação	5
1.2.1 Resultados	5
2 Árvore Balanceada (AVL)	12
2.1 Complexidade	12
2.1.1 Tempo de execução esperado	12
2.1.2 Melhor caso	12
2.2 Implementação	12
2.2.1 Resultados	13
3 Tabela de dispersão(Hash)	19
3.1 Complexidade	19
3.1.1 melhor caso	19
3.1.2 pior caso	19
3.1.3 Médio caso	20
3.2 Implementação	20
3.2.1 Resultados	20
4 Conclusões	25
4.1 Links Externos	25

Lista de Figuras

1.1	Estrutura de uma árvore binária desbalanceada no pior caso	6
1.2	Estrutura de uma árvore binária desbalanceada	7
1.3	Gráfico do tempo de execução melhor caso na busca na árvore binária . . .	7
1.4	Classe nó implementada em python	8
1.5	Classe árvore implementada em python	8
1.6	Método de busca em arvore binária implementada em python	9
1.7	Tempo de execução do caso médio na busca na árvore binária	10
1.8	Tempo de execução do pior caso na busca na árvore binária	11
2.1	Estrutura de uma árvore balanceada	13
2.2	Método que atualiza a altura de um nó	13
2.3	Método de balanceamento	14
2.4	Métodos de rotação	15
2.5	MClasse Node para árvore balanceada	16
2.6	Classe Tree para árvore balanceada	17
2.7	Tempo de execução nas busca em uma árvore balanceada	18
2.8	Tempo de execução nas busca em uma árvore balanceada, no pior caso . . .	18
3.1	Tabela de dispersão adpatada em python	21
3.2	Método Search para tabela de dispersão	22
3.3	Gráfico do tempo de execução do melhor caso na busca na tabela hash . . .	22
3.4	Gráfico do tempo de execução do pior caso na busca na tabela hash	23
3.5	Gráfico do tempo de execução do caso médio na busca na tabela hash . . .	24
3.6	Gráfico do tempo de execução de todos os casos na busca na tabela hash . .	24

1. Árvore Binária

Uma árvore binária é uma estrutura de dados computacional que consiste em nós interligados, onde cada nó pode ter no máximo dois filhos: um filho à esquerda e um filho à direita, sendo os filhos a direita, valores maiores do que o nó pai, e os filhos a esquerda são menores. Essa característica cria uma hierarquia entre os nós, permitindo a organização eficiente de dados.

Uma das principais vantagens das árvores binárias é sua eficiência na busca, inserção e remoção de elementos. Devido à sua estrutura hierárquica e ao uso de comparações ordenadas, as operações em uma árvore binária têm uma complexidade média de $O(\log n)$, onde n é o número de elementos na árvore.

1.1 Complexidade

1.1.1 pior caso

O pior caso de busca ocorre quando a árvore não está balanceada (visto no capítulo 2), ou seja, possui uma estrutura linear. Por exemplo, se os elementos forem inseridos em ordem crescente ou decrescente na árvore, ela se tornará uma lista encadeada, com uma estrutura parecida com a vista na figura 1.1. Nesse cenário, a complexidade de busca será $O(n)$, onde n é o número de elementos na árvore. A busca terá que percorrer todos os nós da árvore até encontrar o elemento desejado, pois não haverá ramificações para otimizar a busca.

1.1.2 médio Caso

A busca em uma árvore binária, segue a mesma lógica de inserção, onde irá comparar o valor buscado e ir para a direita ou esquerda do nó, e tem uma complexidade média de $O(\log n)$, onde n é o número de elementos na árvore. Isso ocorre porque, apesar de poderem haver casos onde a árvore não esteja totalmente balanceada, a cada comparação, o espaço de busca é dividido pela metade, reduzindo a quantidade de nós que precisam ser percorridos.

1.1.3 Melhor caso

O melhor caso na busca na árvore binária ocorre quando o item buscado está na raiz da árvore, assim não entra em recursão e o tempo é constante.

1.2 Implementação

O algoritmo foi implementado usando a linguagem python, usando orientação a objetos, como visto nas figuras 1.4 e 1.5, e aplicando o método de busca recursiva nos nós da árvore, visto em 1.6. Onde ao inserir um valor, é criado um novo nó e ele ira ser passado recursivamente para a árvore até achar o local correto dele na estrutura. O método de busca seguirá a mesma lógica do método **insert**, onde irá percorrer os nós de forma recursiva até achar o local onde deveria estar e retornando verdadeiro caso esteja. Para o teste de tempo de busca foram inseridos valores na árvore e logo após foi sorteado um número aleatório para ser buscado na estrutura.

1.2.1 Resultados

Implementado o algoritmo, os resultados tiveram resultados próximos de um crescimento logarítmico no caso médio, com alguns ruídos vistos em 1.7, que podem estar sendo gerados pela altura do nó onde o item procurado estava, caso estivesse na estrutura. No pior caso, onde a árvore estava se comportando como lista linear a busca teve uma demora consideravel comparado aos demais casos, tendo tempo de execução praticamente linear, visto em 1.8, como foi esperado. No melhor caso o tempo foi constante, sempre buscando pelo valor que estivesse na raiz da árvore, visto em 1.3.



Figura 1.1: Estrutura de uma árvore binária desbalanceada no pior caso

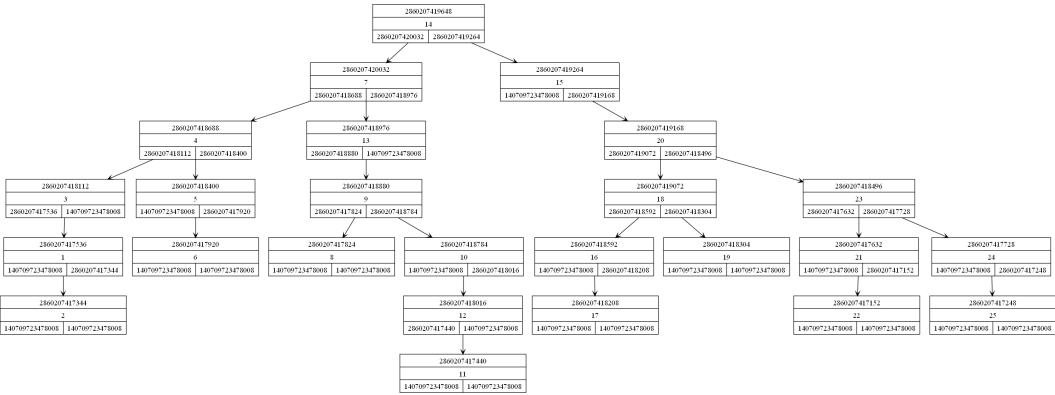


Figura 1.2: Estrutura de uma árvore binária desbalanceada

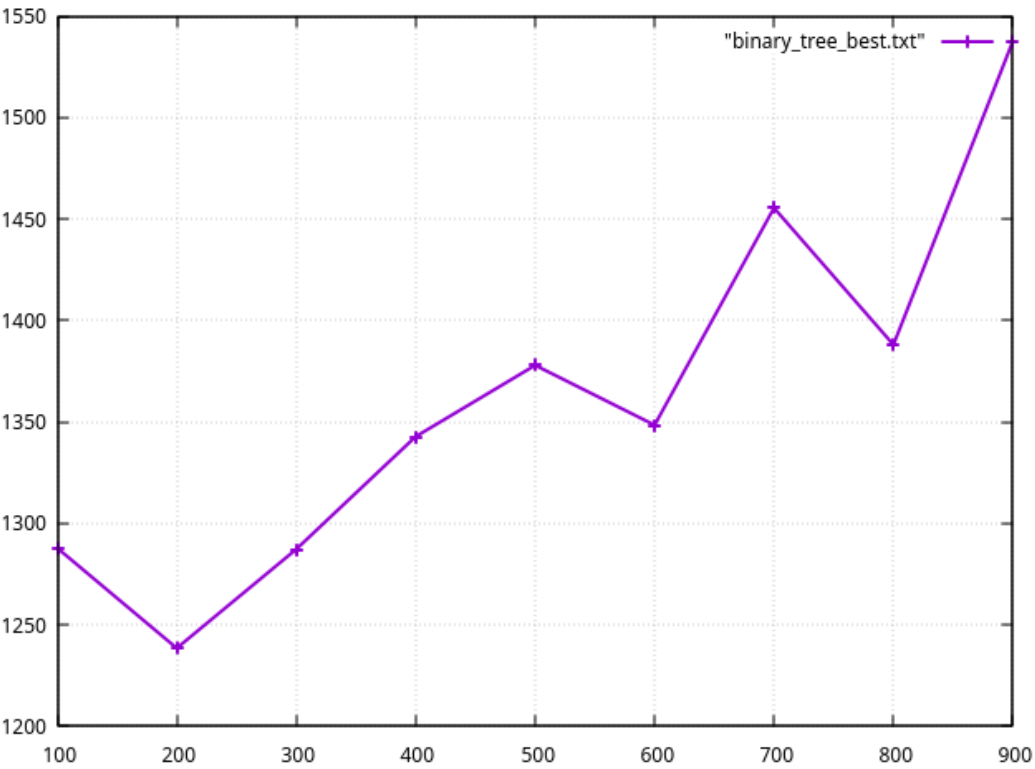
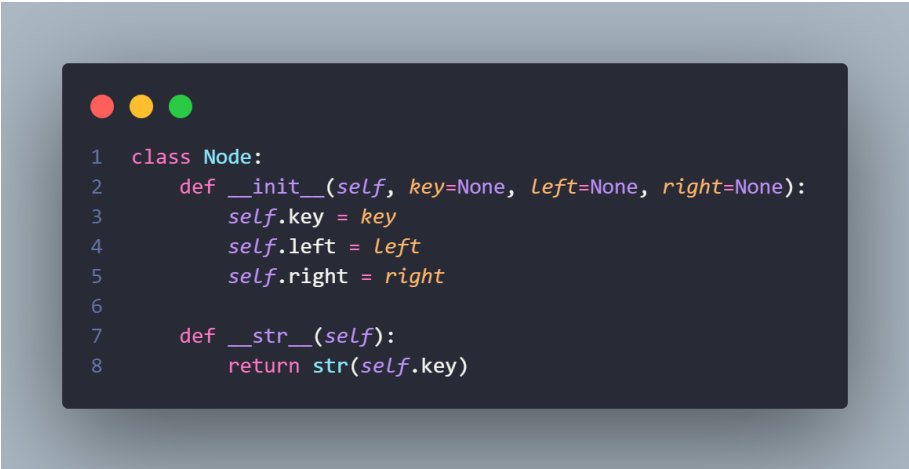


Figura 1.3: Gráfico do tempo de execução melhor caso na busca na árvore binária



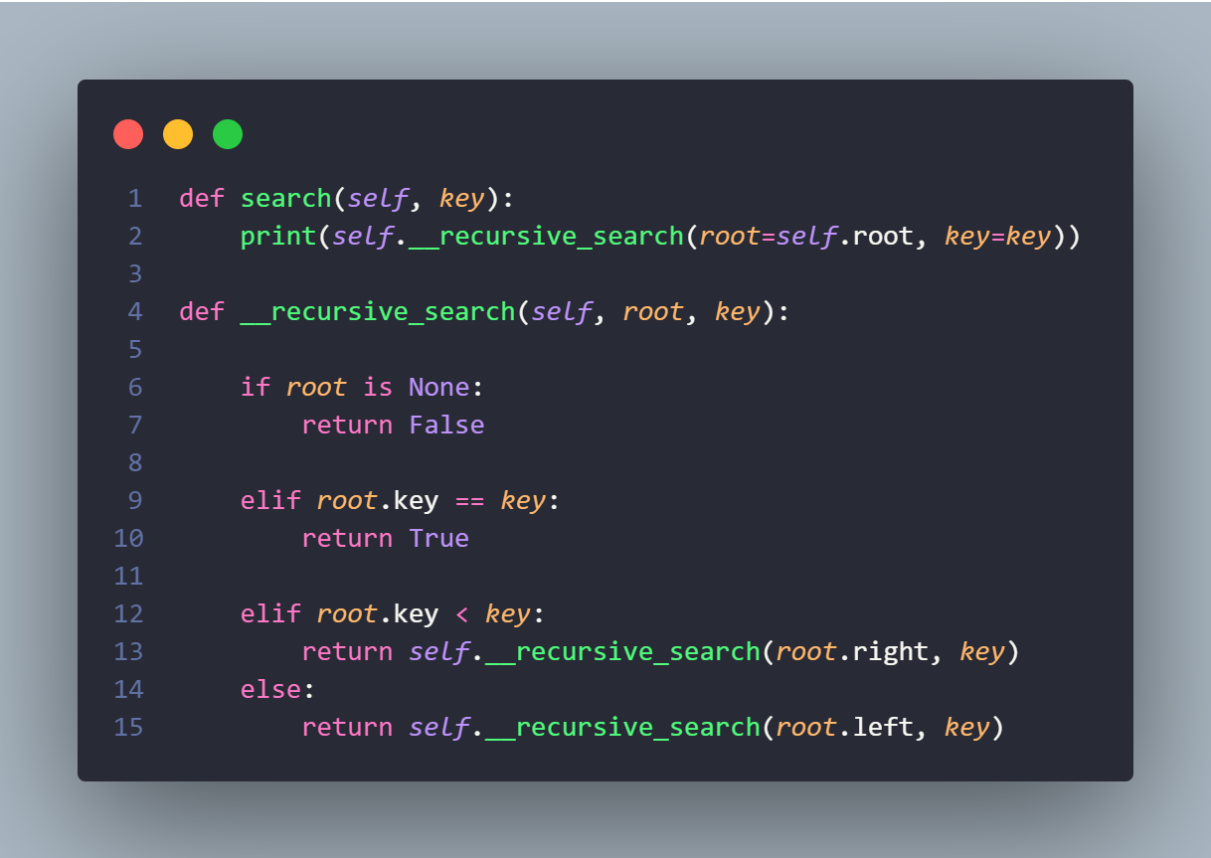
```
1 class Node:
2     def __init__(self, key=None, left=None, right=None):
3         self.key = key
4         self.left = left
5         self.right = right
6
7     def __str__(self):
8         return str(self.key)
```

Figura 1.4: Classe nó implementada em python



```
1 class Tree:
2     def __init__(self):
3         self.root = None
4
5     def insert(self, value):
6         newnode = Node(key=value)
7         self.__recursive_insert(root=self.root, value=newnode)
8
9     def __recursive_insert(self, root, value):
10        if self.root is None:
11            self.root = value
12        else:
13            if root.key < value.key:
14                if root.right is None:
15                    root.right = value
16                else:
17                    self.__recursive_insert(root=root.right, value=value)
18            else:
19                if root.left is None:
20                    root.left = value
21                else:
22                    self.__recursive_insert(root=root.left, value=value)
```

Figura 1.5: Classe árvore implementada em python



```
1  def search(self, key):
2      print(self.__recursive_search(root=self.root, key=key))
3
4  def __recursive_search(self, root, key):
5
6      if root is None:
7          return False
8
9      elif root.key == key:
10         return True
11
12     elif root.key < key:
13         return self.__recursive_search(root.right, key)
14     else:
15         return self.__recursive_search(root.left, key)
```

Figura 1.6: Metodo de busca em arvore binária implementada em python

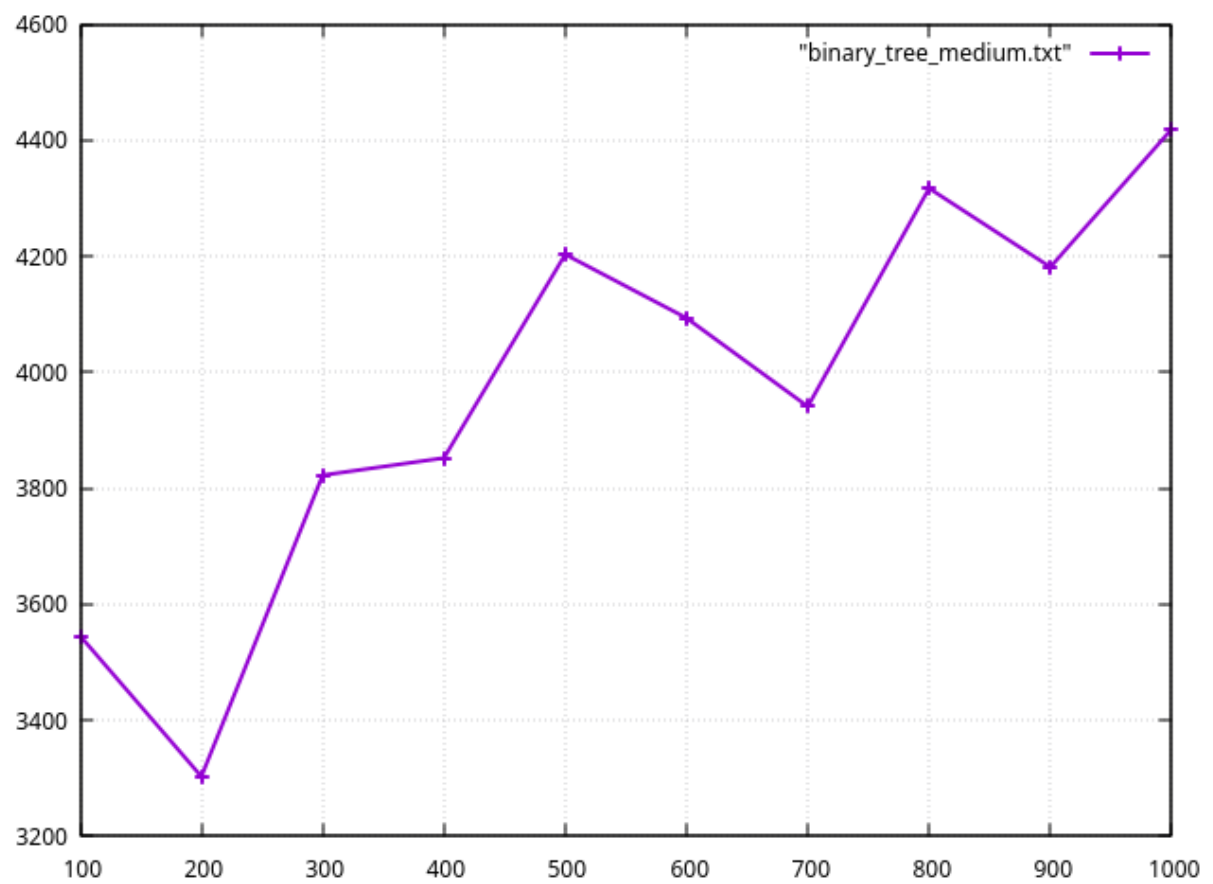


Figura 1.7: Tempo de execução do caso médio na busca na árvore binária

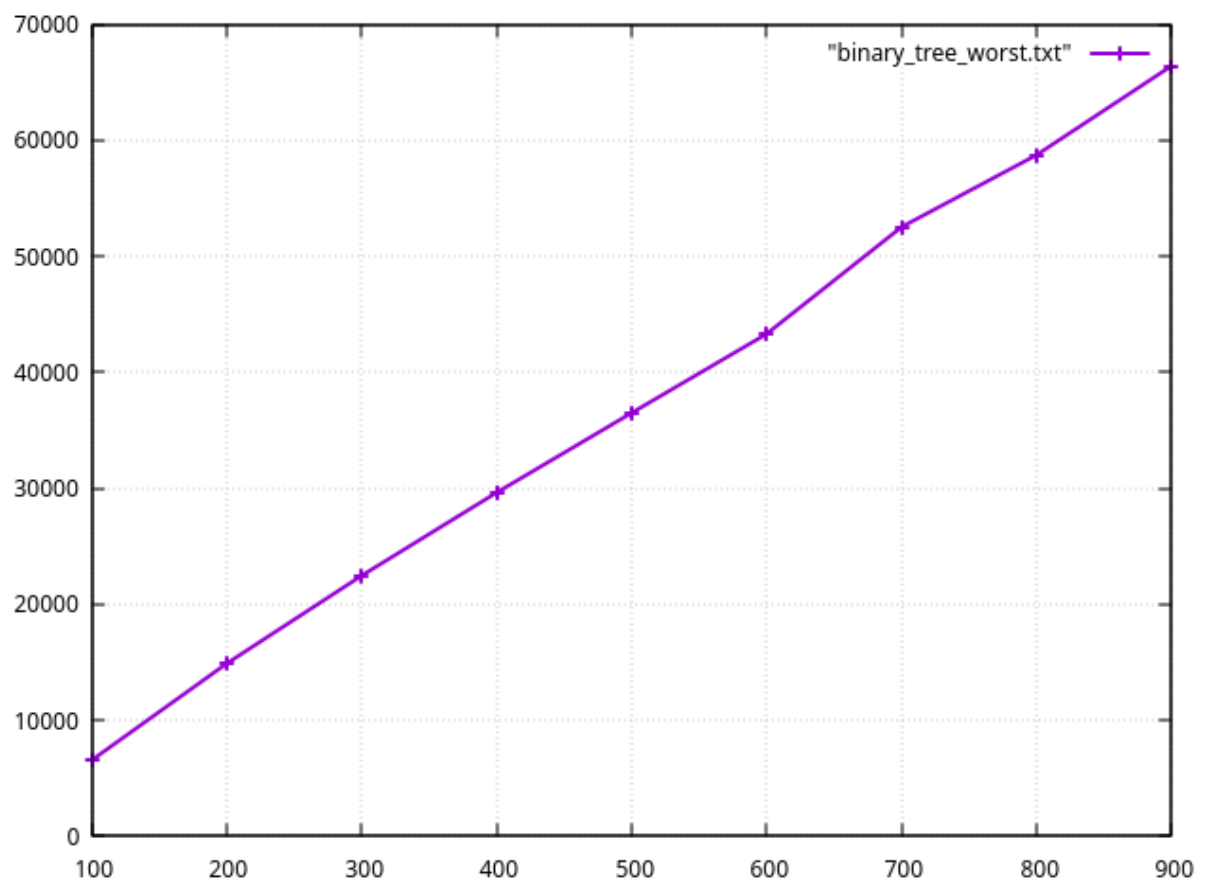


Figura 1.8: Tempo de execução do pior caso na busca na árvore binária

2. Árvore Balanceada (AVL)

As árvores binárias balanceadas seguem a mesma estrutura que as árvores binárias normais vistas no capítulo 1 deste documento. Porém em cada remoção ou inserção de um novo item na árvore, irá ser executado um algoritmo que mudará os itens de lugar, para garantir que uma árvore AVL permaneça balanceada, são realizadas rotações nos nós conforme necessário. As rotações são operações de reorganização que mantêm a estrutura balanceada da árvore, ajustando a posição dos nós.

Em uma árvore AVL, a diferença entre as alturas das subárvores esquerda e direita de qualquer nó (chamada de fator de balanceamento) é mantida dentro de um limite específico. Normalmente, esse limite é de 1, o que significa que a diferença entre as alturas das subárvores de qualquer nó não pode exceder 1. Embora as árvores AVL garantam um desempenho eficiente em termos de complexidade de tempo, elas podem exigir um pouco mais de tempo e esforço para a manutenção do balanceamento durante as operações de inserção e remoção.

2.1 Complexidade

2.1.1 Tempo de execução esperado

As árvores AVL oferecem uma complexidade de tempo de $O(\log n)$ para a operação de busca, onde n é o número de elementos na árvore. Isso ocorre porque, com o balanceamento adequado, a altura da árvore é mantida mínima possível, fazendo com que a cada comparação, o espaço de busca seja dividido pela metade, reduzindo a quantidade de nós que precisam ser percorridos. Ao contrario do visto em 1.2 conforme os itens forem sendo inseridos aleatoriamente, os nós irão ser balanceados, então com valores bem parecidos da primeira figura, teremos uma estrutura parecida com 2.1

2.1.2 Melhor caso

O melhor caso possível para árvore balanceada é quando o item procurado está na raiz, o que tornar o tempo de busca constante, assim como na árvore desbalanceada. A diferença é que na árvore desbalanceada, esse item é o primeiro a ser inserido na estrutura, e na balanceada esse item é o que provavelmente é o item que divide os valores da árvore ao meio.

2.2 Implementação

O algoritmo foi implementado usando a linguagem python, com base no visto nas imagens 1.4 e 1.5, e ainda seguindo o mesmo método de busca visto em 1.6, porém aplicando

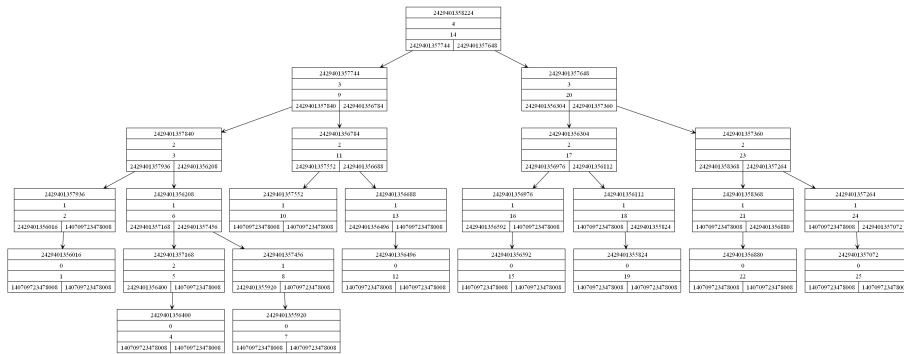


Figura 2.1: Estrutura de uma árvore balanceada

as mudanças necessárias para balancear a árvore, sempre atualizando a altura dos nós através do método visto na figura 2.2, e logo após aplicando o balanceamento (visto em 2.3) que efetua as rotações vistas em 2.4. Todos esses métodos adicionais são chamados quando cada novo nó é inserido, sendo essas as mudanças feitas em comparação a uma árvore binária não balanceada, sendo vistas em 2.5 e 2.6.

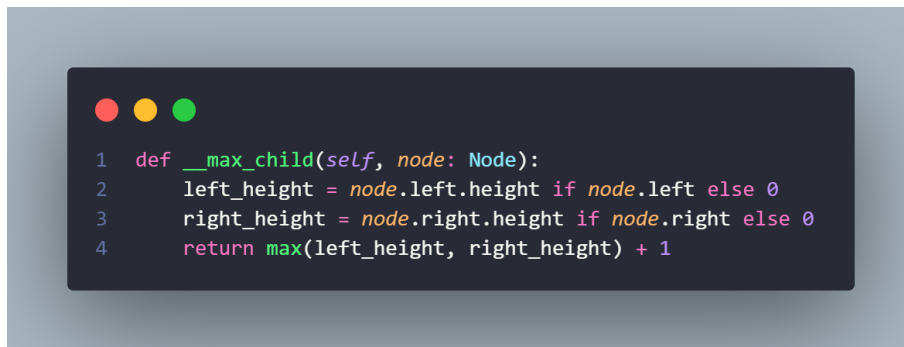



Figura 2.2: Método que atualiza a altura de um nó

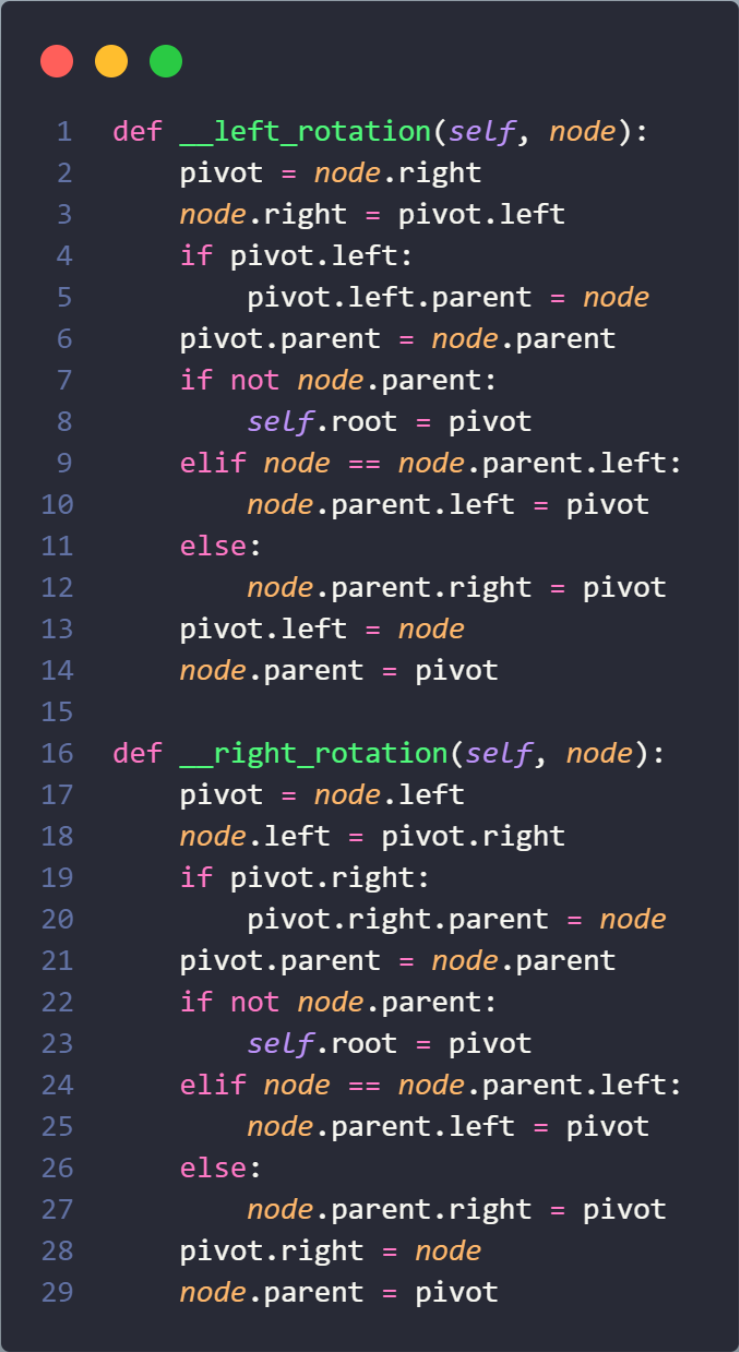
2.2.1 Resultados

O algoritmo foram implementados e assim como visto em 1.3, na árvore binaria normal, o tempo foi próximo ao constante. Por algum motivo desconhecido o pior caso, visto em 2.8, se saiu melhor do que no caso médio, visto em 2.7, e até mesmo pior que o caso médio da arvore desbalanceada, vista 1.7 porém ainda teve um crescimento logarítmico.



```
1 def __balance_factor(self, node):
2
3     left_height = node.left.height if node.left else 0
4     right_height = node.right.height if node.right else 0
5     return left_height - right_height
6
7 def __balance(self, node):
8     if node is None:
9         return
10
11     balance_factor = self.__balance_factor(node)
12     if balance_factor > 1:
13         if self.__balance_factor(node.left) < 0:
14             self.__left_rotation(node.left)
15             self.__right_rotation(node)
16     elif balance_factor < -1:
17         if self.__balance_factor(node.right) > 0:
18             self.__right_rotation(node.right)
19             self.__left_rotation(node)
```

Figura 2.3: Método de balanceamento



```
1  def __left_rotation(self, node):
2      pivot = node.right
3      node.right = pivot.left
4      if pivot.left:
5          pivot.left.parent = node
6      pivot.parent = node.parent
7      if not node.parent:
8          self.root = pivot
9      elif node == node.parent.left:
10         node.parent.left = pivot
11     else:
12         node.parent.right = pivot
13     pivot.left = node
14     node.parent = pivot
15
16 def __right_rotation(self, node):
17     pivot = node.left
18     node.left = pivot.right
19     if pivot.right:
20         pivot.right.parent = node
21     pivot.parent = node.parent
22     if not node.parent:
23         self.root = pivot
24     elif node == node.parent.left:
25         node.parent.left = pivot
26     else:
27         node.parent.right = pivot
28     pivot.right = node
29     node.parent = pivot
```

Figura 2.4: Métodos de rotação



```
1 class Node:
2     def __init__(self, key=None,
3                   left=None,
4                   right=None,
5                   parent=None,
6                   height=1) -> None:
7
8         self.key = key
9         self.right = right
10        self.left = left
11        self.parent = parent
12        self.height = height
13
14    def __str__(self):
15        return f'{self.key}'
```

Figura 2.5: MClasse Node para árvore balanceada

```
1 class Tree:
2     def __init__(self) -> None:
3         self.root = None
4
5     def insert(self, value):
6         newnode = Node(key=value)
7         self.__recursive_insert(root=self.root, value=newnode)
8
9     def __recursive_insert(self, root, value):
10
11         if self.root is None:
12
13             self.root = value
14
15         else:
16             value.parent = root
17             if root.key < value.key:
18                 if root.right is None:
19                     root.right = value
20                 else:
21                     self.__recursive_insert(root.right, value)
22                     self.__balance(root)
23
24             else:
25                 if root.left is None:
26                     root.left = value
27                 else:
28                     self.__recursive_insert(root.left, value)
29
30                 self.__balance(root)
31                 root.height = self.__max_child(root)
```

Figura 2.6: Classe Tree para árvore balanceada

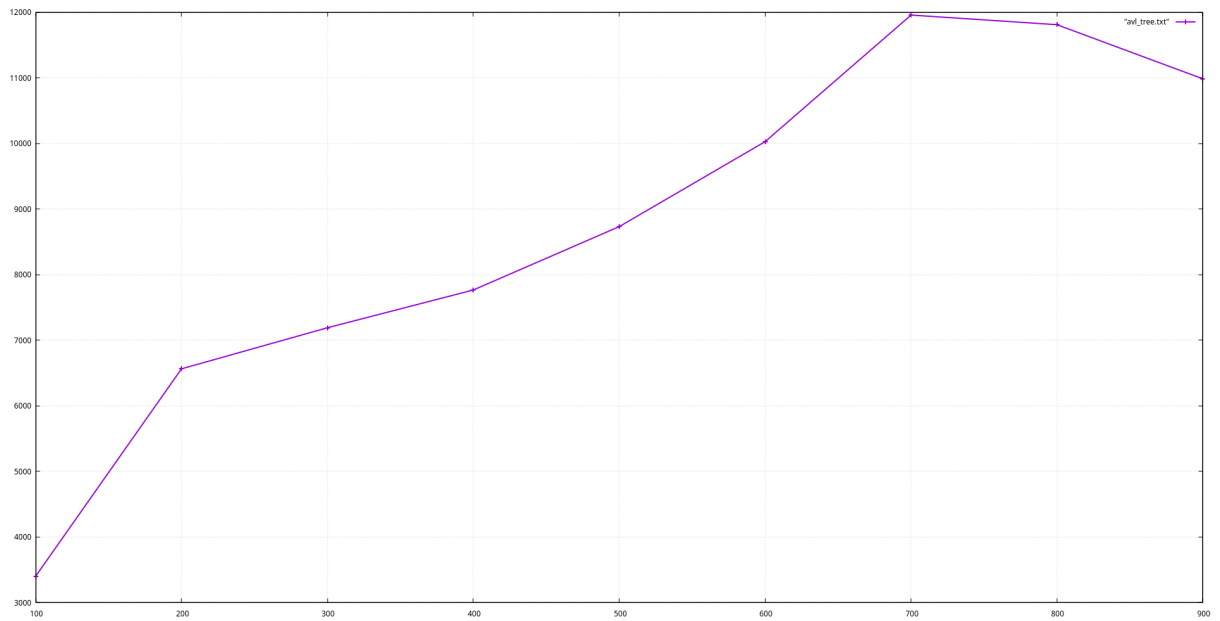


Figura 2.7: Tempo de execução nas busca em uma árvore balanceada

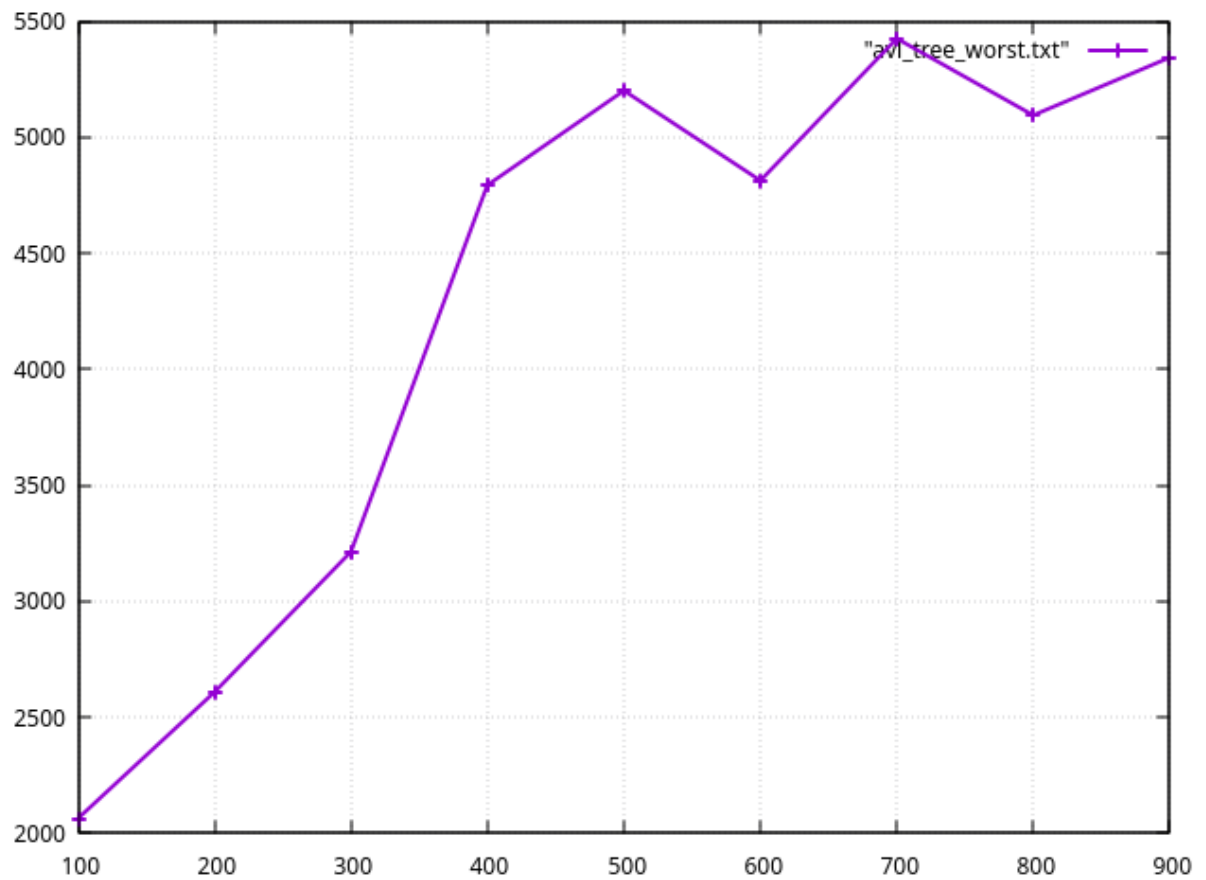


Figura 2.8: Tempo de execução nas busca em uma árvore balanceada, no pior caso

3. Tabela de dispersão(Hash)

Uma tabela de dispersão, também conhecida como hash table, é uma estrutura de dados que permite o armazenamento e recuperação eficiente de elementos. Ela é composta por um array de tamanho fixo, no qual os elementos são armazenados de acordo com suas chaves. Para mapear uma chave a um índice na tabela, é aplicada uma função de dispersão (hash function) que calcula um valor único para cada chave. Esse valor é então utilizado como índice para acessar a posição correspondente no array. Para o algoritmo citado neste documento, foi adotado como calculo de hash, o resto da divisão pelo tamanho da tabela.

O objetivo dessa estrutura de dados é ter tempo de busca sempre próximo ao constante, já que deve buscar diretamente na chave que aquele valor estaria, mas em certos casos pode haver vários valores em uma única chave. Há diversas maneiras de contornar esse caso, uma delas é armazenar os itens em forma de lista encadeada, porém nesses casos a lista pode crescer desenfreadamente, tornando o tempo de busca linear. No algoritmo apresentado nesse documento, para impedir casos como esse, a cada novo item que for inserido na lista encadeada é executado um algoritmo que calcula o estouro do fator de carga, que é dado por $n/m \geq 1$ onde m é o tamanho da tabela, e n o número de itens na lista encadeada, ou seja, o tamanho da lista encadeada nunca pode ser maior do que o tamanho da tabela.

Quando ocorre o caso em que o fator de carga é maior ou igual a 1, é executado um algoritmo de redispersão (re-hash) que irá dobrar o o tamanho da tabela, e realocar cada item que já estava na tabela, para uma nova posição de acordo com o novo calculo de hash.

3.1 Complexidade

3.1.1 melhor caso

O melhor caso na busca da tabela de dispersão é constante $O(1)$, já que a partir do valor inserido para busca é calculado o hash e é buscado diretamente na posição que ele estaria caso estivesse dentro da estrutura, e quando tem apenas um item ou até mesmo nenhuma, naquela posição.

3.1.2 pior caso

O pior caso na busca na tabela de dispersão é linear $O(n)$, e ocorre quando o item buscado não está na tabela, mas quando aplicado a função hash, essa posição existe na tabela porém tem vários itens nessa posição em forma de lista encadeada, logo será linear.

3.1.3 Médio caso

O tempo de execução esperado ocorre quando os valores estão devidamente distribuídos na tabela, podendo estar ordenados em listas encadeadas ou assumindo uma posição, única. Neste caso apesar pode ter algumas variações esse tempo tende a ser mais próximo do melhor caso, que seria constante $O(1)$.

3.2 Implementação

O algoritmo foi implementado usando a linguagem python e orientação a objetos, onde cada espaço na tabela é uma lista vazia, e quando um novo item é inserido, essa lista vai sendo preenchida, de acordo com o cálculo de hash visto em 3.1, e na operação de **re-hash** os itens atuais da tabela são salvos para serem realocados, e uma nova tabela vazia é criada para ser preenchida com os valores antigos. No algoritmo de busca, o **hash** é calculado e a busca é feita primeiramente na tabela e caso haja uma lista, ela será percorrida, visto em 3.2.

3.2.1 Resultados

Após implementar o algoritmo, o tempo de execução do melhor caso sempre esteve próximo de constante tendo ruídos muito pequenos, vistos em 3.3, se mantendo próximos à 2500 nano-segundos. O pior caso seguiu exatamente como esperado, quando há um acúmulo de itens em uma única chave o tempo teve seu crescimento de forma linear, visto em 3.4. No tempo de execução esperado, houve um crescimento que aparenta ser linear a primeira vista, porém se manteve sempre entre 2000 e 3000 nano-segundos, como visto em 3.5, então ainda está próximo de constante, sendo melhor visualizado na comparação com os demais na figura 3.6.



```
1 class HashTable:
2     def __init__(self):
3         self.size = 1
4         self.table = [[]]
5
6     def _hash_function(self, key):
7         return key % self.size
8
9     def _re_hash(self):
10        self.size = self.size * 2
11        old_table = self.table
12        self.table = [[] for _ in range(self.size)]
13        for item in old_table:
14            if item:
15                if len(item) > 1:
16                    for i in item:
17                        self.insert(i)
18                else:
19                    self.insert(item[0])
20
21    def insert(self, value):
22        if len(self.table[self._hash_function(value)]) >= self.size:
23            self._re_hash()
24
25        self.table[self._hash_function(value)].append(value)
```

Figura 3.1: Tabela de dispersão adaptada em python

```
1 def search(self, value):  
2     index = self._hash_function(value)  
3     if self.table[index]:  
4         for item in self.table[index]:  
5             if item == value:  
6                 return True  
7     return False  
8
```

Figura 3.2: Método Search para tabela de dispersão

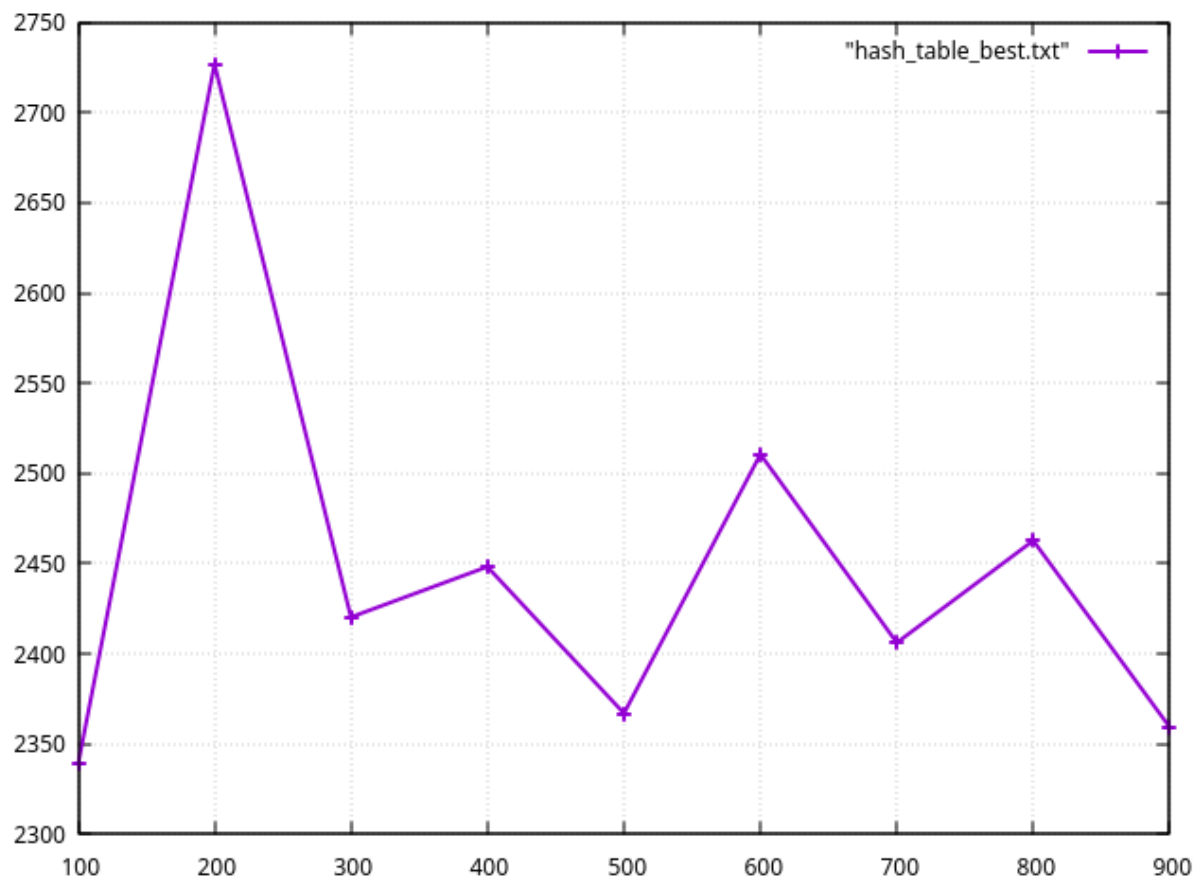


Figura 3.3: Gráfico do tempo de execução do melhor caso na busca na tabela hash

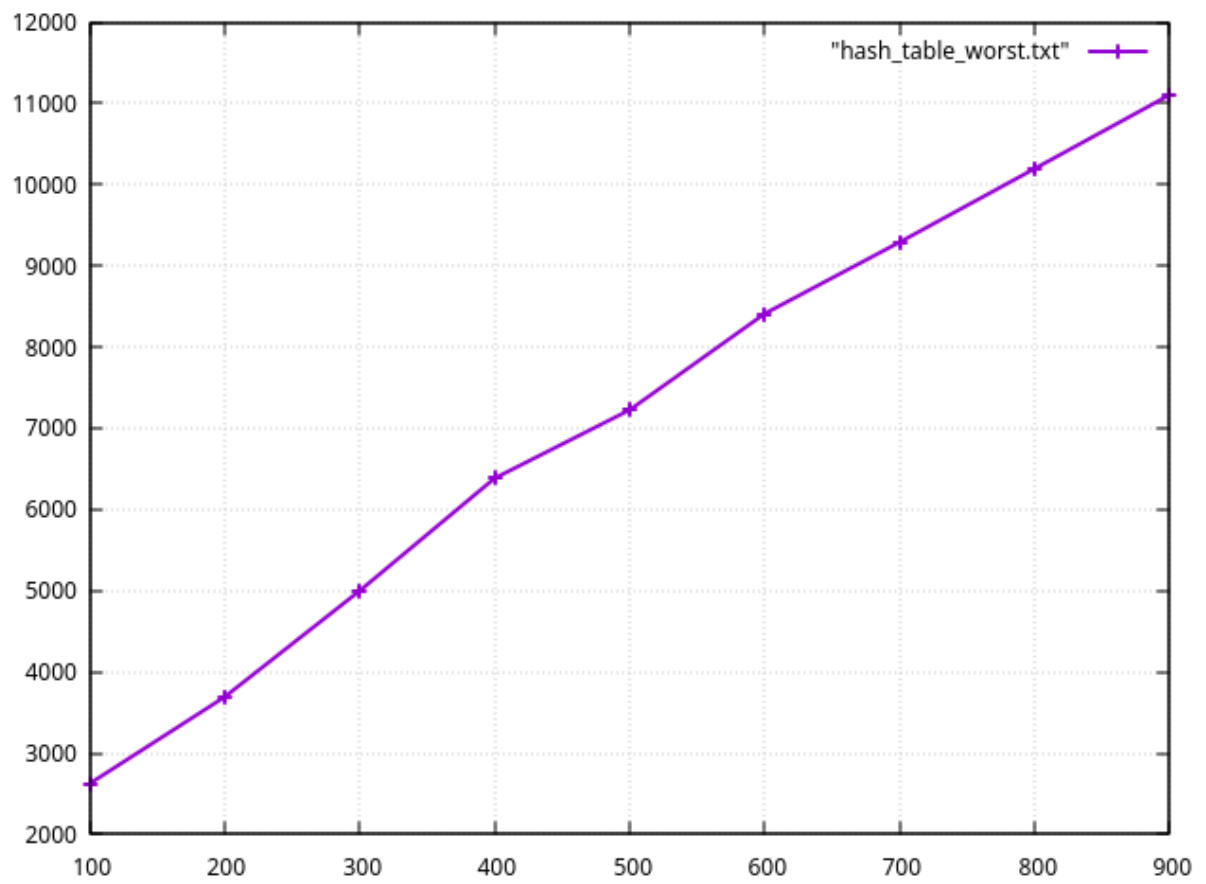


Figura 3.4: Gráfico do tempo de execução do pior caso na busca na tabela hash

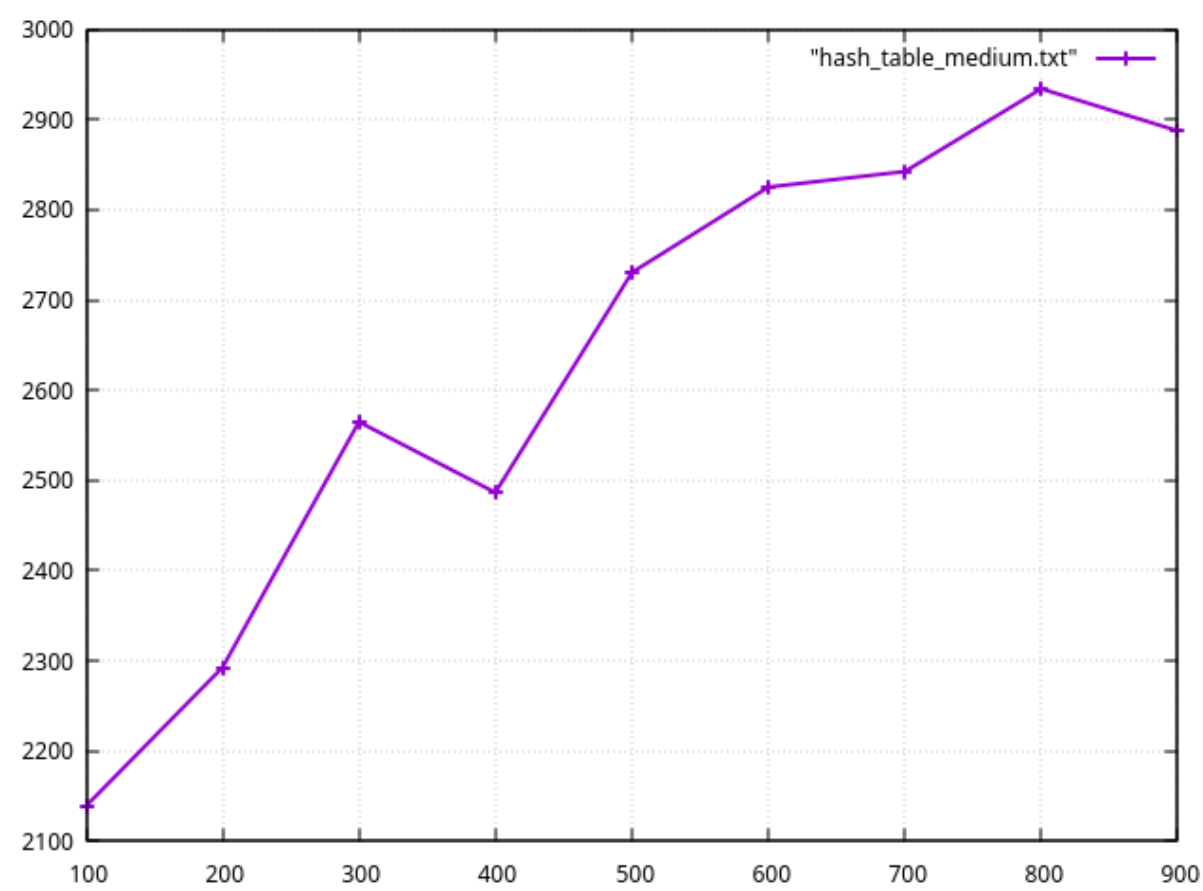


Figura 3.5: Gráfico do tempo de execução do caso médio na busca na tabela hash

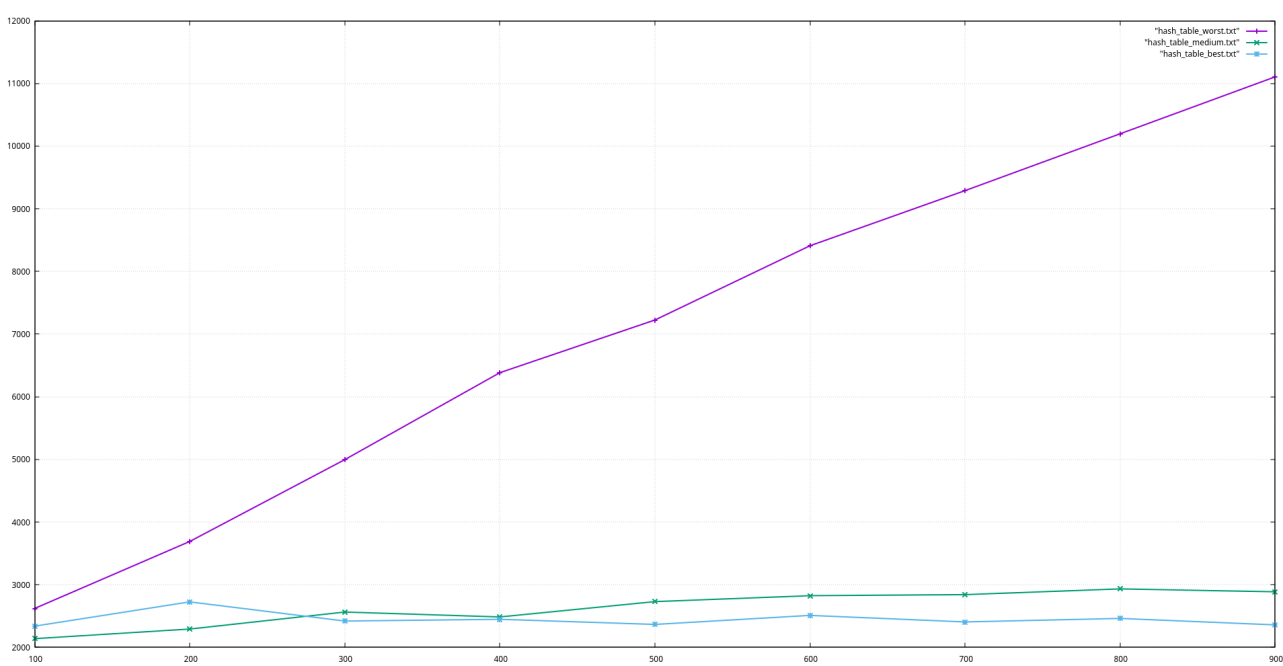


Figura 3.6: Gráfico do tempo de execução de todos os casos na busca na tabela hash

4. Conclusões

Como visto em 3.5, o tempo de execução da tabela hash se manteve em uma constante perto de 2000 e 3000 nano-segundos, e manteve uma boa média em comparação aos outros algoritmos que em seus desempenhos tiveram no mínimo 4500 nano-segundos como tempo de execução. Logo a tabela hash se mostra uma alternativa interessante como estrutura de dados de tempo de busca eficiente, tendo com um dos melhores exemplos os dicionários que são estruturas de dados da linguagem python, que foi utilizada para implementar esses algoritmos.

4.1 Links Externos

Os algoritmos que foram usados para os testes contidos nesse documento podem ser encontrados no seguinte repositório remoto no github: [data-structure](#)