Trabalho de relatório de análise de algoritmos associados ao problema de busca em árvores binária, balanceada e de dispersão

Cláudio P. T. Araújo* João P. S. Medeiros†

Recebido em 14 de agosto de 2024, aceito em 14 de agosto de 2024.

Resumo

Este relatório documenta o desenvolvimento e a análise de algoritmos associados a três estruturas de dados fundamentais para a eficiência de operações de busca: árvores binárias, árvores AVL e tabelas de dispersão (Hash). O trabalho foi realizado utilizando a linguagem de programação C, com foco na implementação, otimização e comparação do desempenho dessas estruturas em diferentes cenários de uso.

1 Introdução

A eficiência de operações de busca é crucial em diversos sistemas e aplicações computacionais. A escolha da estrutura de dados adequada pode impactar significativamente o desempenho de uma aplicação, especialmente em termos de tempo de execução e uso de memória. Entre as várias estruturas de dados disponíveis, as árvores binárias de busca (BST), as árvores AVL e as tabelas de dispersão (hash tables) se destacam pela sua versatilidade e eficiência.

1.1 Contexto

- **Árvore Binária de Busca (BST)**: As árvores binárias de busca se comportam onde cada nó possui no máximo dois filhos, e os nós são organizados de forma que todos os nós na subárvore esquerda têm valores menores, enquanto os da subárvore direita têm valores maiores. Essa propriedade facilita a busca, inserção e remoção de elementos.

- **Árvore AVL**: As árvores AVL são uma extensão das árvores binárias de busca que garantem que a árvore permaneça balanceada após cada operação de inserção ou remoção. O balanceamento é alcançado através de alterações da própria árvore no qual rotaciona os devidos valores, que mantêm a diferença de altura entre as subárvores de qualquer nó em no máximo um, melhorando a eficiência das operações.

- **Tabela de Dispersão (Hash Table)**: As tabelas de dispersão utilizam funções de hash para mapear chaves a índices de uma tabela, permitindo que as operações de busca, inserção e remoção sejam realizadas em tempo constante no caso ideal. Diferentes técnicas de resolução de colisões, como encadeamento separado e endereçamento aberto, são usadas para lidar com colisões de hash.

*Aluno do Bacharelado em Sistemas de Informaçãoda Universidade Federal do Rio Grande do Norte. (e-mail: claudio.pereira.710@ufrn.edu.br)

†Professor do Departamento de Computação e Tecnologiada Universidade Federal do Rio Grande do Norte. (e-mail: jpsm1985@gmail.com)

2 Metodologia

A análise será conduzida através de uma série de experimentos, onde conjuntos de dados variados serão usados para testar o desempenho das operações em cada estrutura. Ferramenta de medição de tempo será empregada para coletar dados sobre o tempo de execução. Os resultados obtidos serão comparados para fornecer uma visão clara das vantagens e desvantagens de cada algoritmo em diferentes contextos.

3 Experimentos

Nesta sessão, será feito os experimentos para análise do tempo de execução de cada algoritmo dado no escopo do trabalho e falado brevemente na subsessão 1.1 no qual cada um tem a sua distinta peculiaridade, dado isso a seguir teremos uma explicação mais detalhada de cada um dos algoritmos e também a explicação de cada um dos seus casos de tempo de execução.

3.1 Árvore Binária

Uma árvore binária é uma estrutura de dados onde cada nó possui, no máximo, dois filhos, um na esquerda e outro pela direita, no qual cada um segue uma regra distinta, que todos os elementos a esquerda seriam menores doque seu respectivo "pai", pois cada valor colocado em uma árvore binaria também pode transformar um nó filho em um nó pai que também vai carregar um pela esquerda e direita, e como a esquerda vai ser sempre menor, o da direita seguindo a lógica sempre vai ser maior doque o seu respectivo pai. Esta estrutura é bastante utilizada em várias áreas da ciência da computação, incluindo pesquisa, organização de dados, e implementação de algoritmos de busca.

O funcionamento básico de uma árvore binária envolve as operações de inserção, remoção e busca de nós, todas seguindo regras que mantêm a organização hierárquica da árvore, mas que para o nosso relatório será apenas discutido a parte da busca das mesmas, por isso processo de busca em uma árvore binária pode ser descrito em três passos principais:

Início na Raiz: A busca começa no nó raiz da árvore. Se o valor do nó raiz é igual ao valor procurado, a busca termina com sucesso.

Deslocamento à Esquerda ou à Direita: Se o valor procurado for menor que o valor do nó atual, a busca continua na subárvore esquerda; se for maior, na subárvore direita.

Repetição ou Terminação: O processo se repete até que o valor seja encontrado ou até que a subárvore esquerda ou direita seja nula, o que indica que o valor não está presente na árvore.

Agora para se falar de complexidade do tempo de execução de busca, o algoritmo da Árvore binária possui 3 tipos diferentes, sendo o unico dentre os 3 algoritmos analisados neste relatório que possui mais de 2 tempos de execução diferentes, sendo o melhor caso constante $O(1)$ o médio caso sendo logaritmo $O(n \log n)$ e no pior caso linear $O(n)$ que serão mais bem analisados a seguir com seus respectivos gráficos.

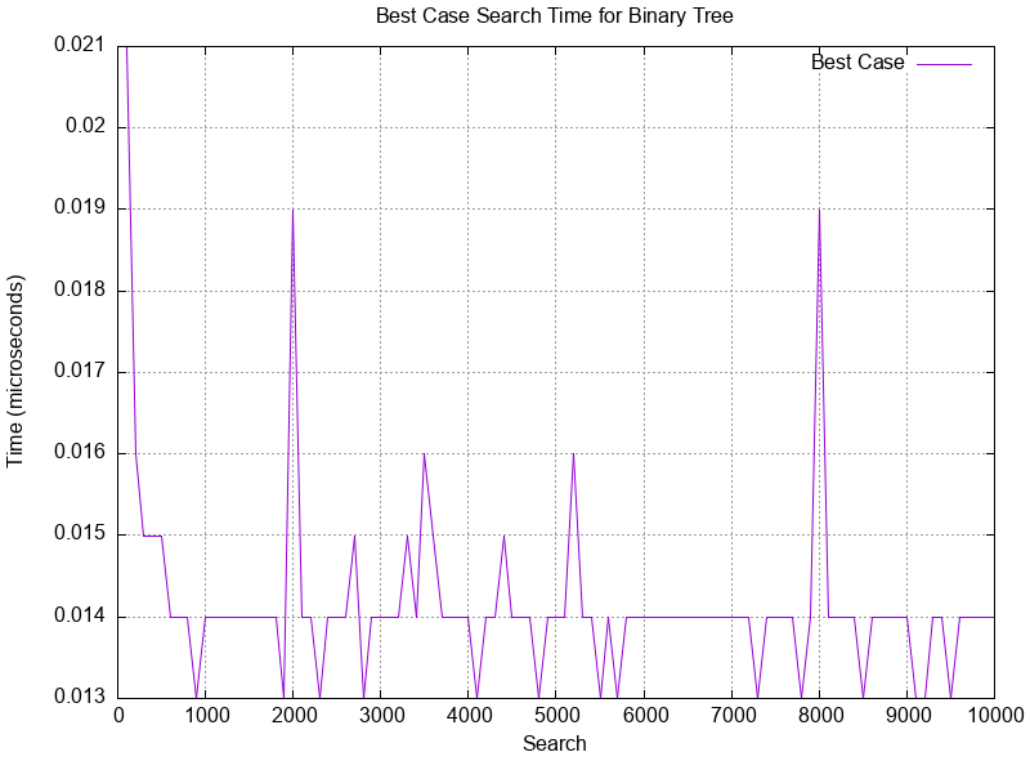

Figura 1: Gráfico do melhor caso Binary Tree Search

Melhor Caso $O(1)$ **:** Ocorre quando o valor procurado está no nó raiz da árvore. Neste cenário, a busca é realizada em tempo constante $O(1)$ pois não há necessidade de percorrer mais nós, que se torna evidente no gráfico acima já que mesmo possuindo algumas partes que aumentam e diminuem, ainda assim a tendência do gráfico foi se manter no 0.014 microsegundos que torna essa atividade sendo constante.

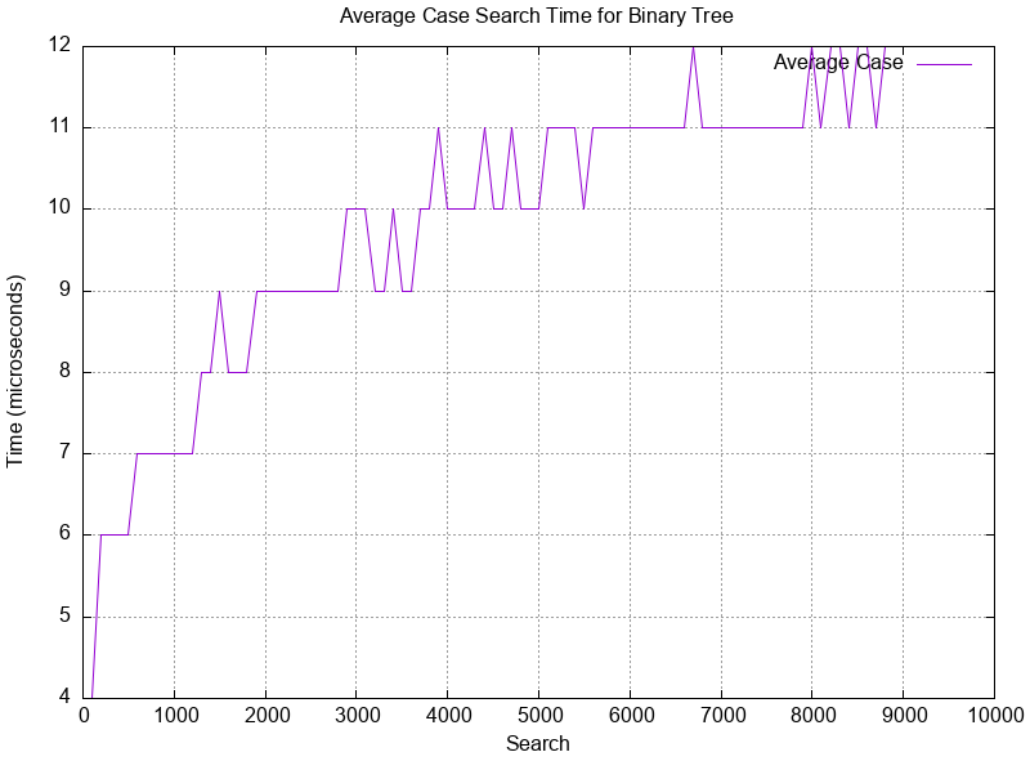

Figura 2: Gráfico do médio caso Binary Tree Search

Médio Caso $O(\log n)$ **:** Ocorre quando a árvore está mais balanceada e se busca um resultado qualquer dentro da mesma uma vez que a busca se divide entre as subárvores esquerda e direita, reduzindo significativamente o número de comparações necessárias, que no gráfico da pra se analisar a curva logaritma $O(n \log n)$ do gráfico por ser uma arvore moderamente mais balanceada, tornando a busca assim bem eficiente.

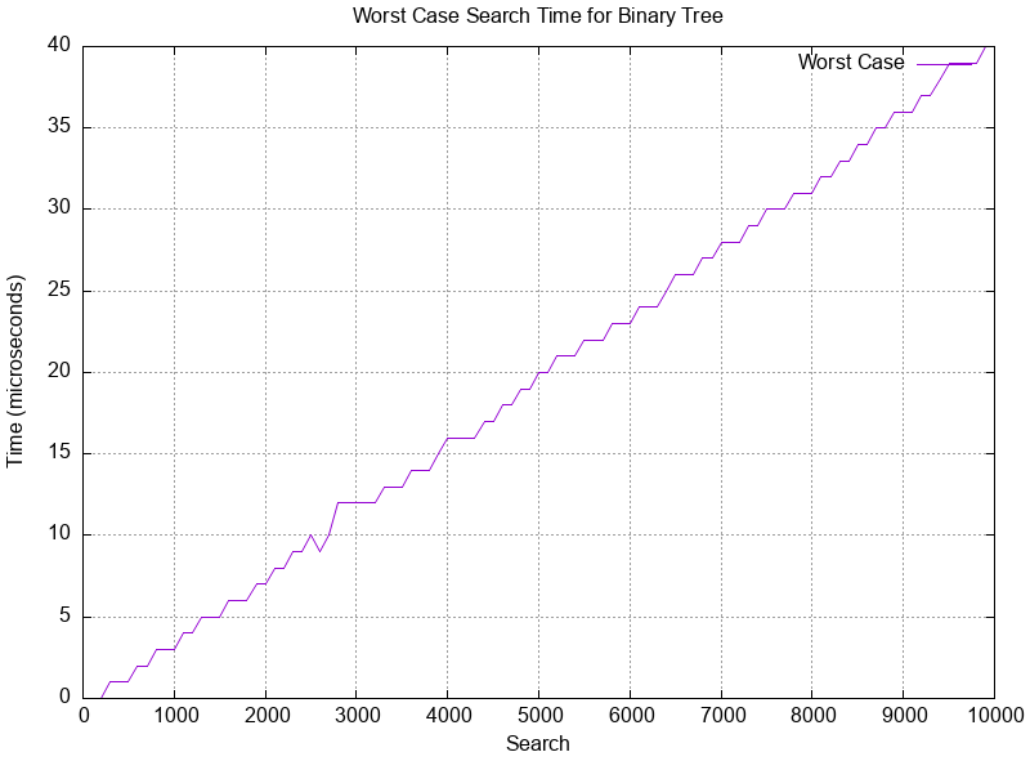

Figura 3: Gráfico do pior caso Binary Tree Search

Pior Caso $O(n)$ **:** Ocorre quando a árvore se degenera em uma lista encadeada (por exemplo, se os elementos são inseridos em ordem crescente ou decrescente, sendo assim uma lista apenas com elementos a direita ou elementos apenas à esquerda). Nesse caso, a complexidade da busca é $O(n)$ onde n é o número de nós na árvore o que diminui significativamente a eficiência.

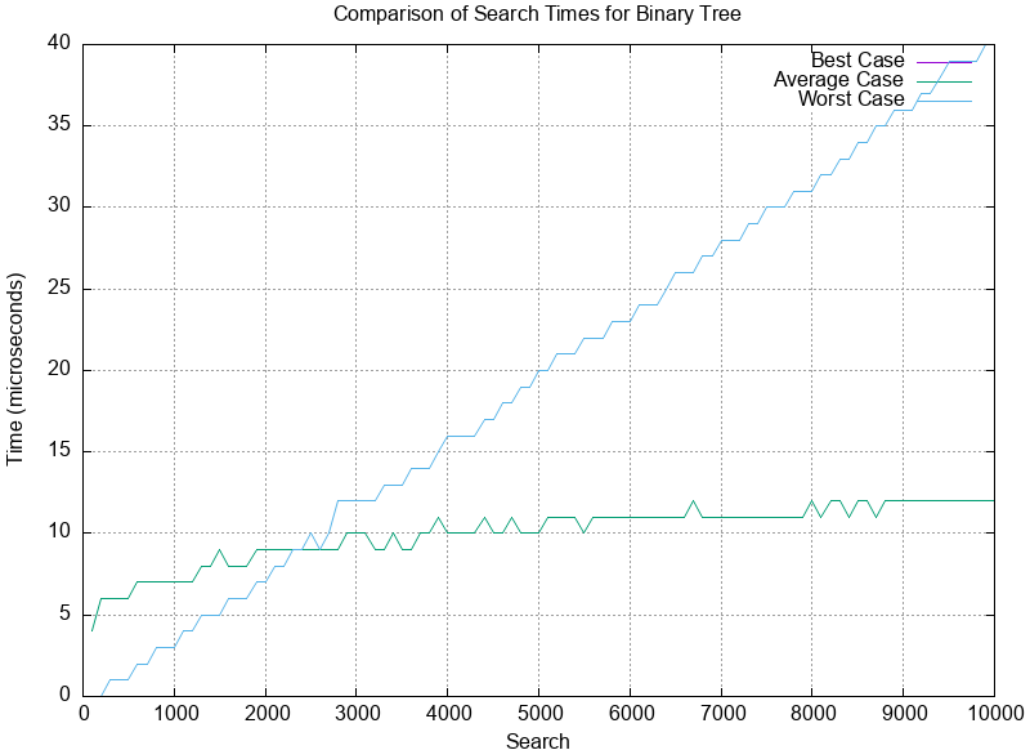

Figura 4: Gráfico de todos os casos da Binary Tree Search

Com isso ao comparar os gráficos vemos uma diferença discrepante entre todos os casos analisados pelo fato de cada um possuir tempos de execução diferentes, mas de modo geral a eficiência depende do balanceamento da árvore, coisa que para a árvore binária não existe nenhuma forma de se balancear ao ficar adicionando mais elementos, já que para esse algoritmo em específico para a árvore ser balanceada deve-se ter já um numero fixo de nós e que o usuário passe bem para o programa a ordem correta para se balancear a árvore já que se for de forma aleatória provavelmente ela vai ficar desbalanceada e assim custando mais tempo de execução, pois, em cenários onde a árvore se degenera, a operação de busca pode se tornar tão ineficiente quanto em estruturas lineares. Para manter a eficiência, é essencial considerar métodos de balanceamento, ou seja, se utilizar do algoritmo da árvore AVL, mas para a árvore binaria existe algumas vantagens e desvantagens na implementação da mesma que são:

Vantagens:

- Árvores binárias são relativamente simples de implementar e compreender, tornando-as uma escolha comum em aplicações onde o balanceamento automático não é crítico.
- Árvores binárias podem ser facilmente adaptadas para incluir outras operações, como inserção e remoção, além de serem base para estruturas mais complexas, como árvores AVL e árvores rubro-negras.

Desvantagens:

- A eficiência da árvore pode ser significativamente impactada pela ordem em que os elementos são inseridos. Inserções ordenadas podem resultar em uma árvore desbalanceada e ineficiente.
- Ao contrário de estruturas como árvores AVL, a árvore binária não se reequilibra automaticamente, exigindo intervenção manual ou o uso de técnicas adicionais para garantir o desempenho ideal.

3.2 Árvore Balanceada (AVL)

As árvores AVL são uma extensão das árvores binárias, projetadas especificamente para garantir que a árvore permaneça balanceada após cada operação de inserção ou remoção. Isso é alcançado mantendo a diferença de altura entre as subárvores esquerda e direita de qualquer nó (também conhecida como fator de balanceamento) em menor ou igual a 1. A operação de busca em árvores AVL segue a mesma lógica das árvores binárias, mas com a vantagem de que o balanceamento automático da árvore assegura uma eficiência consistente.

Com isso o funcionamento da árvore AVL não se difere muito da árvore binária já que, como foi dito, a AVL é uma extensão da mesma por isso compartilha das mesmas caracteristicas de funcionamento, apenas diferindo em um ponto que adiciona uma variável na Struct para medir a altura da árvore e ver se está balanceada ou não, que ao saber que está balanceada vai se ter funções expecíficas para tratar disso e fazer com que a árvore seja totalmente balanceada e se pareça realmente como uma árvore no qual da raiz vai se crescendo vários galhos uniformemente e que a cada novo nó adicionado a árvore vai se refazer para poder cumprir a limitação de a soma das alturas serem menores ou iguais a 1, com isso em mente os tempos de execução do melhor, médio e pior caso vão prover uma análise melhor de como funcionam.

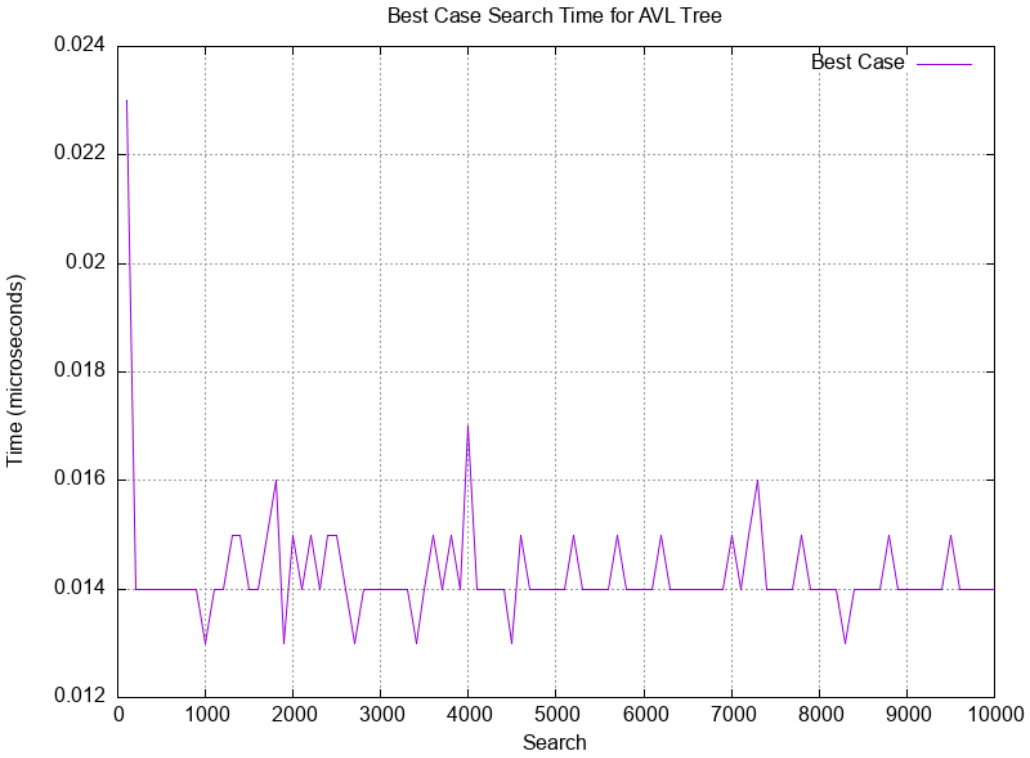

Figura 5: Gráfico do melhor caso AVL Search

Melhor Caso $O(1)$ **:** Ocorre quando o valor procurado está no nó raiz, resultando em um tempo de execução constante $O(1)$ semelhante a propria árvore binária antes vista no relatório, pois não há necessidade de percorrer mais nós, que se torna evidente no gráfico acima já que mesmo possuindo algumas partes que aumentam e diminuem, ainda assim a tendência do gráfico foi se manter no 0.014 microsegundos que torna essa atividade sendo constante.

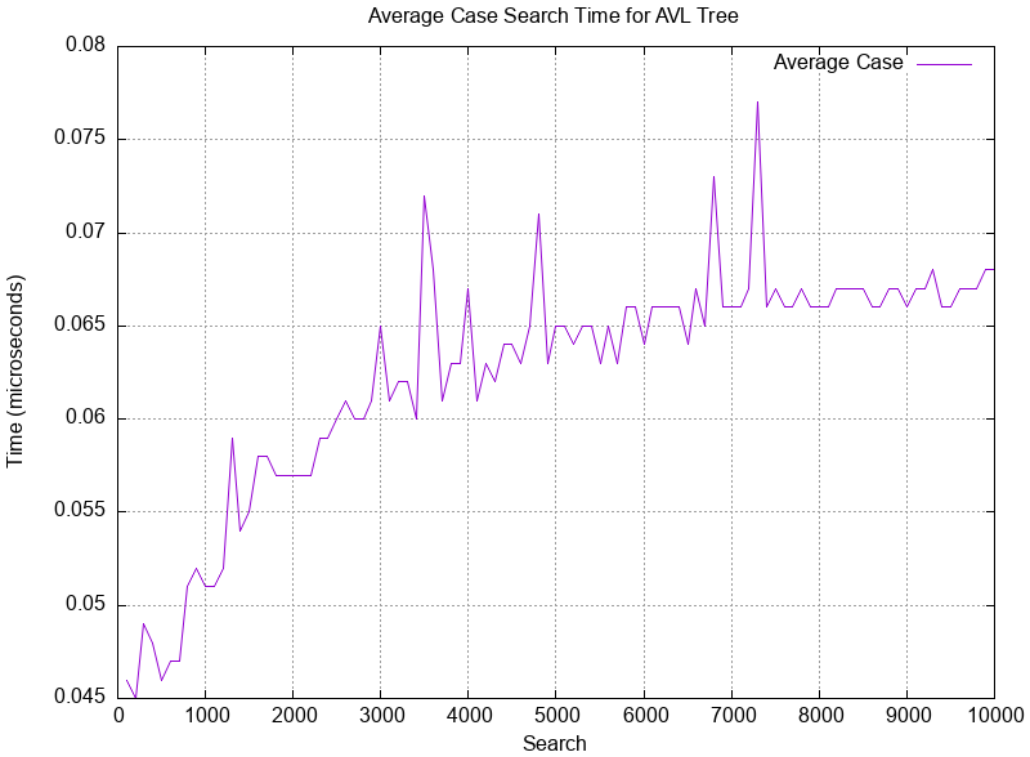

Figura 6: Gráfico do médio caso AVL Search

Médio Caso $O(\log n)$ **:** Ocorre quando se busca um resultado qualquer uma vez que a busca se divide entre as subárvores esquerda e direita, reduzindo significativamente o número de comparações necessárias, pois graças ao balanceamento automático, a árvore AVL mantém uma complexidade logarítmica $O(n \log n)$ na maioria dos cenários, o que é ilustrado no gráfico como esperado de uma árvore AVL.

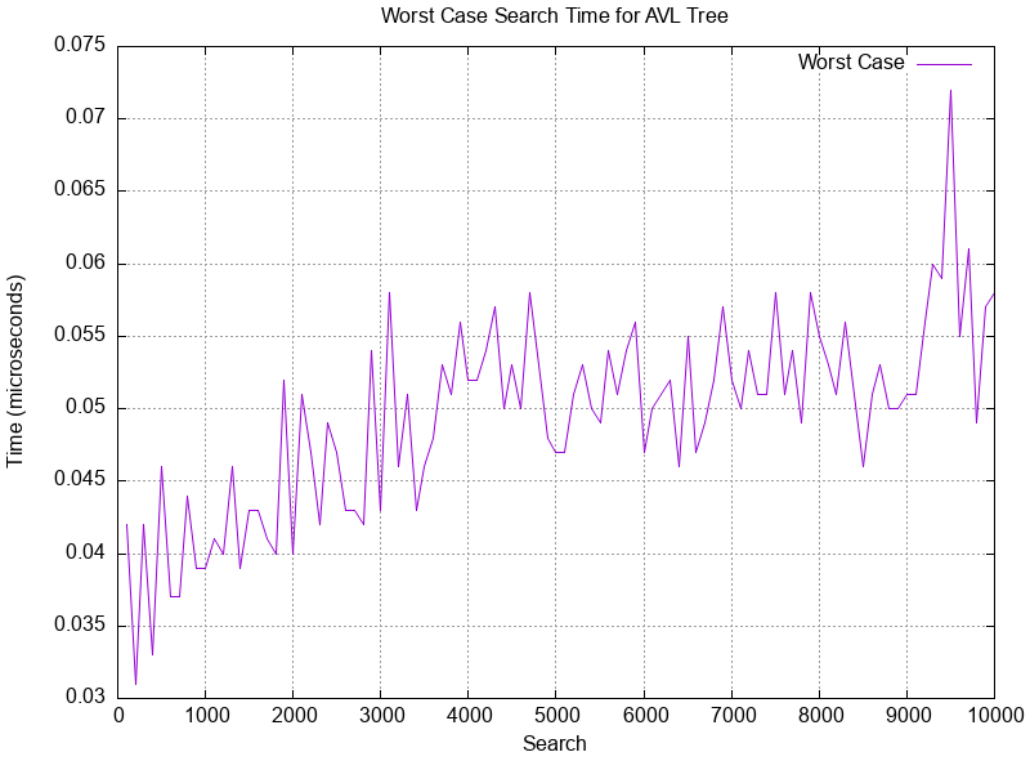

Figura 7: Gráfico do pior caso AVL Search

Pior Caso $O(\log n)$ **:** Ocorre quando Mesmo no pior caso, onde o valor está em uma folha distante, a profundidade da árvore é logarítmica, garantindo que o tempo de execução permaneça $O(n \log n)$, pois o balanceamento da árvore assegura que a profundidade máxima da árvore seja sempre logarítmica em relação ao número de nós, prevenindo a degeneração da árvore e também mudanças no tempo de execução.

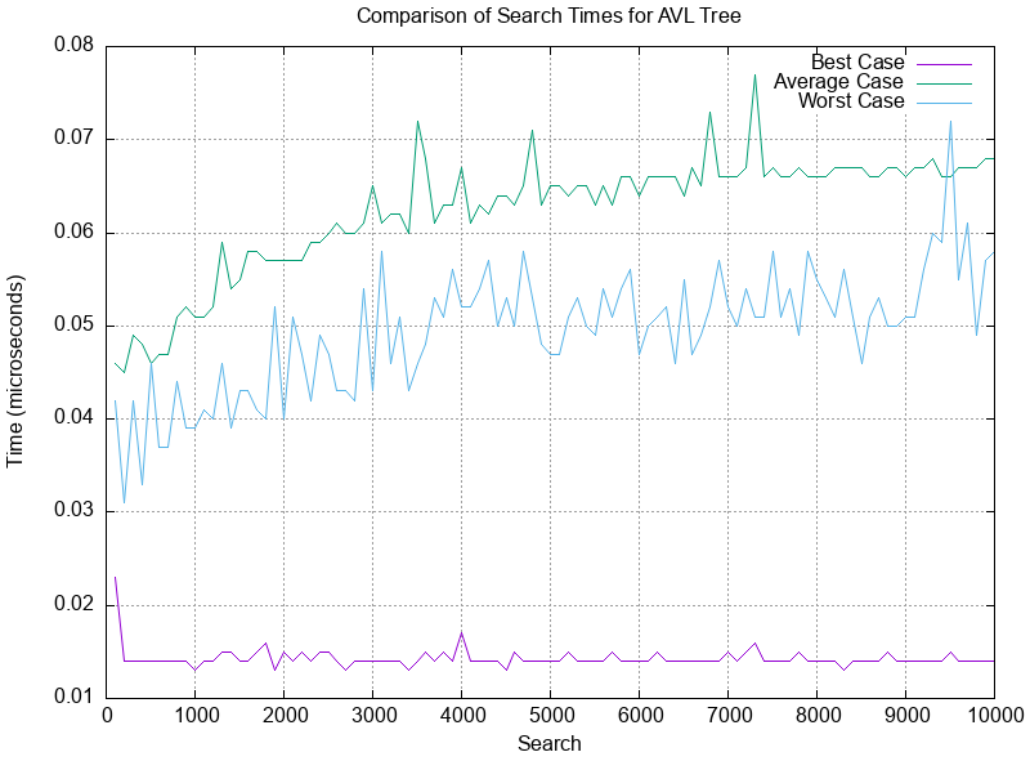

Figura 8: Gráfico de todos os casos da AVL Search

Com isso ao comparar todos os gráficos podemos ter uma breve visão de que realmente é um tipo de algoritmo muito rápido e que não vai crescer tanto no seu tempo de execução por ser logaritmo $O(n \log n)$, com isso ao ver o gráfico pode-se notar algo estranho de que em teoria o caso médio está custando mais tempo de execução doque o pior caso, mas isso é meramente ilustrativo e também so mostra que não existe diferença entre os algoritmos pois mesmo que "esteja custando mais"é um intervalo muito pequeno para constar uma diferença gritante já que eles estão rodando na casa dos nanosegundos mesmo que o gráfico mostre sendo em microsegundos, então de uma forma mais ampla não existe diferença entre o pior e médio caso em uma árvore AVL, sendo muito superior a sua forma antecessora que era a árvore binária tanto por tempo de execução e tanto por organização da árvore em si, por isso ao considerar o uso de árvores AVL para operações de busca, é importante analisar suas principais vantagens e desvantagens:

Vantagens:

- Ao contrário das árvores binárias simples, uma árvore AVL nunca se degenera em uma lista encadeada, garantindo que a eficiência da busca seja mantida mesmo com inserções ordenadas.
- A árvore AVL otimiza a estrutura da árvore, assegurando que o espaço seja utilizado de forma eficaz sem desperdiçar memória em ramificações desbalanceadas.

Desvantagens:

- O processo de balanceamento pode adicionar tempo extra às operações de inserção e remoção, embora isso seja compensado pelo ganho de eficiência em operações subsequentes de busca.
- Em cenários onde a inserção e remoção de nós são frequentes e a busca não é crítica, o overhead de manutenção do balanceamento pode não justificar o uso de árvores AVL.

3.3 Tabela de Dispersão (Hash)

As tabelas hash são estruturas de dados amplamente utilizadas devido à sua eficiência em operações de inserção, busca e remoção. Elas funcionam através do uso de uma função hash que mapeia chaves para posições específicas em uma tabela, permitindo acesso direto a essas posições, fazendo com que tempos de execução se tornem muito rapidos tanto que geralmente quando usado o seu tempo de execução se torna constante mas isso vemos mais adiante nos gráficos.

Na operação de busca, a função hash é aplicada à chave que está sendo procurada, gerando um índice na tabela. Se a chave correspondente ao índice é a mesma que a chave procurada, o valor associado é retornado. Caso contrário, estratégias como encadeamento ou sondagem linear são usadas para resolver colisões, onde múltiplas chaves podem mapear para o mesmo índice, por isso o processo de busca em uma tabela hash envolve os seguintes passos:

Cálculo do Índice: A função hash é aplicada à chave procurada, gerando um índice na tabela.

Acesso ao Índice: A chave associada ao índice é comparada com a chave procurada.

- Se as chaves coincidem, o valor é retornado.
- Se há uma colisão (o índice já está ocupado por outra chave), um método de resolução de colisões é utilizado.

Resolução de Colisões: Dependendo da estratégia implementada (por exemplo, encadeamento ou sondagem linear), a busca continua até que a chave seja encontrada ou se conclua que a chave não está presente na tabela.

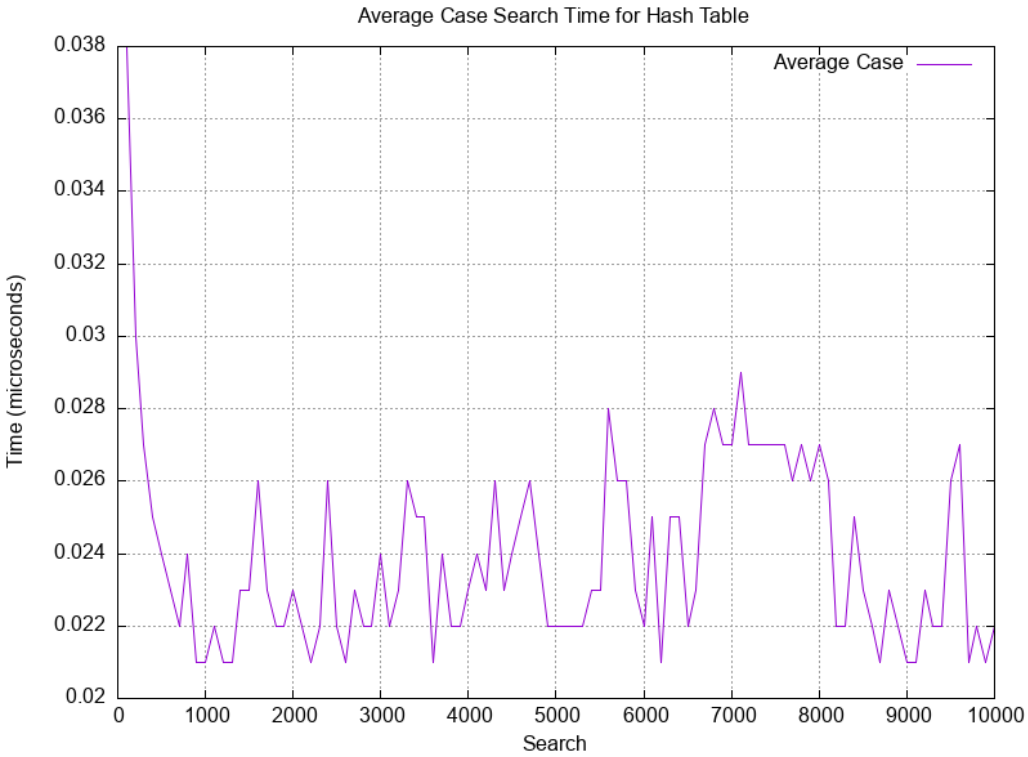

Figura 9: Gráfico do médio caso Hash Table Search

Médio Caso $O(1)$ **:** Em situações ideais, onde poucas colisões ocorrem e a função hash distribui as chaves uniformemente, a busca tem complexidade constante $O(1)$. Isso significa que, em média, a operação de busca pode ser realizada em tempo constante, minimizando o número de colisões independentemente do número de elementos na tabela, mantendo-se em tempo de execução constante $O(1)$ para maioria das ocasiões, mesmo que no gráfico tenha muitos interseções de certa forma, ele ainda assim tende-se em comportamento constante.

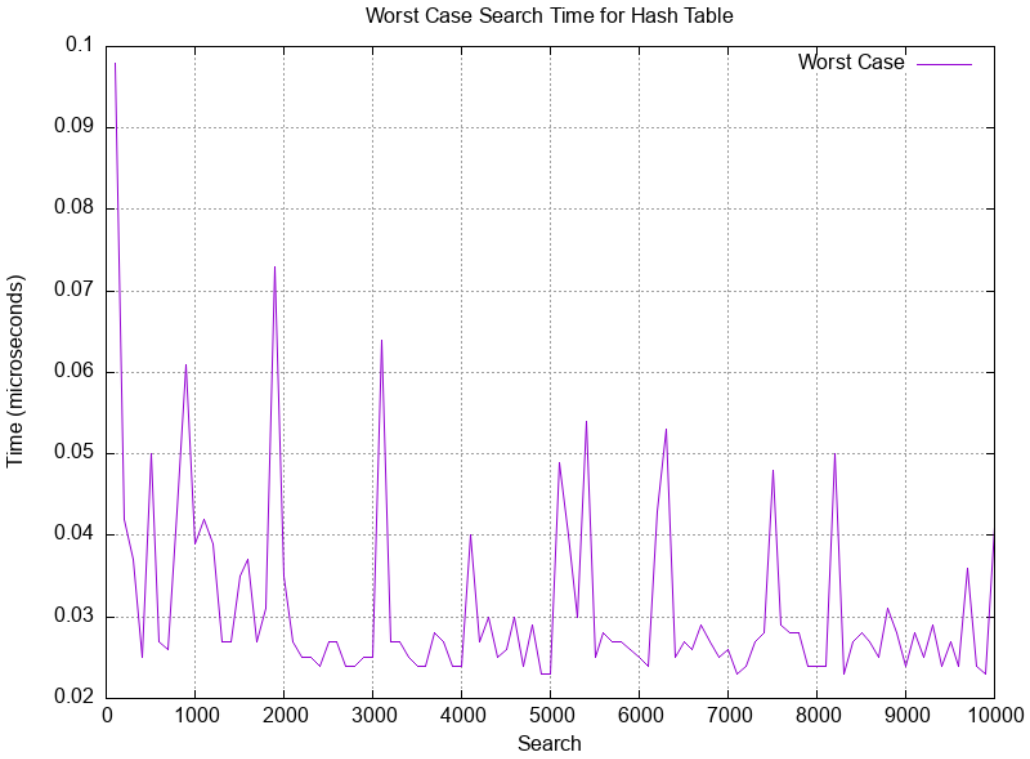

Figura 10: Gráfico do pior caso Hash Table Search

Pior Caso $O(n)$ **:** Ocorre quando muitas colisões ocorrem e várias chaves são mapeadas para o mesmo índice, a complexidade da busca pode degradar para $O(n)$, onde n é o número de elementos na tabela. Isso ocorre quando todas as chaves são mapeadas para o mesmo índice, e a resolução de colisões envolve a verificação de todos os elementos, especialmente em situações onde a função hash não é ideal ou a tabela está sobrecarregada, que é oque podemos ver no gráfico pois mesmo sendo linear ele ainda demoraria para tomar essa característica.

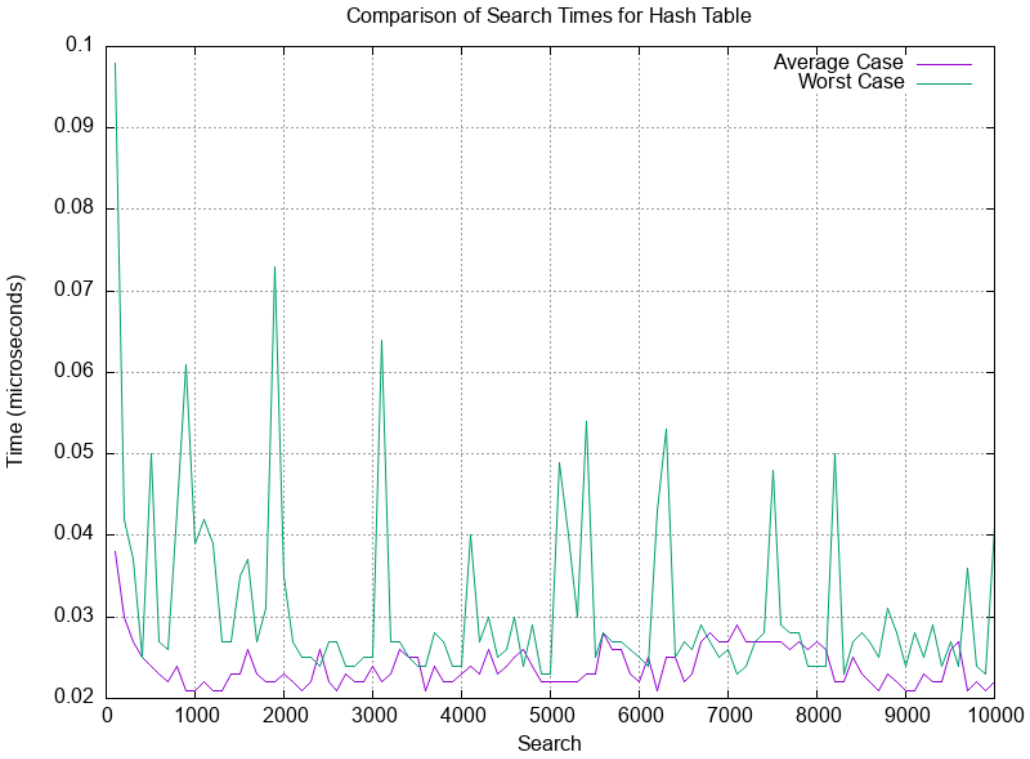

Figura 11: Gráfico de todos os casos da Hash Table Search

Com isso, diferente das árvores foi escolhido apenas tratar o médio e pior caso da tabela de dispersão hash, pois como o caso médio já era constante não se fez necessário ter que confirmar que seu melhor caso também seria constante e teria o mesmo tempo de execução já que esse tipo de algoritmo trabalha mais com chaves de busca, e falando dos gráficos pode-se notar que os tempos mesmo sendo muito diferentes eles ainda estão bastante semelhantes quando exibidos em comparação em gráfico, isso se deve ao fato de que mesmo sendo diferentes em seus tempos de execução, ainda assim o algoritmo de busca consegue ser muito veloz, para se alcançar um grafico realmente linear seria necessário um tamanho de vetor muito maior que 10000, com isso se chega a conclusão de que mesmo sendo diferentes ainda assim possuem caracteristicas que irão deixar a execução de busca extremamente rápida, além de que a tabela hash como já dito anteriormente, funciona com base em chaves de busca que geram indicies que podem ou não estar com colisões, uma boa implementação dessa tabela hash teria como tratar dessas colisões de forma fácil e rápida assim sempre se mantendo no caso médio de serem constantes, com isso algumas vantagens e desvantagens do algoritmo são:

Vantagens:

- Elas são altamente flexíveis, podendo lidar com uma vasta gama de dados e adaptando-se bem a diferentes tipos de chave.
- A lógica de implementação básica das tabelas hash é relativamente simples, embora possa se tornar mais complexa ao implementar técnicas avançadas de resolução de colisões.

Desvantagens:

- O desempenho da tabela hash depende fortemente da qualidade da função hash. Uma função hash mal projetada pode levar a muitas colisões, degradando o desempenho.
- Tabelas hash podem exigir mais espaço de memória do que outras estruturas de dados, especialmente se houver a necessidade de grande espaço reservado para evitar colisões.

3.4 Tempo de Execução Geral

Ao analisar todos os tempos de execução de todos os algoritmos podemos ter como visualizar todos eles juntos para poder se fazer comparações e também com isso tirar conclusões de qual pode ser o melhor ou pior algoritmo para a busca que foi tratado neste relatório, com isso veremos primeiro todos os melhores, médios e piores casos e comentaremos um pouco da discrepância de cada um e logo após será analisado o conjunto com todos os algoritmos presentes.

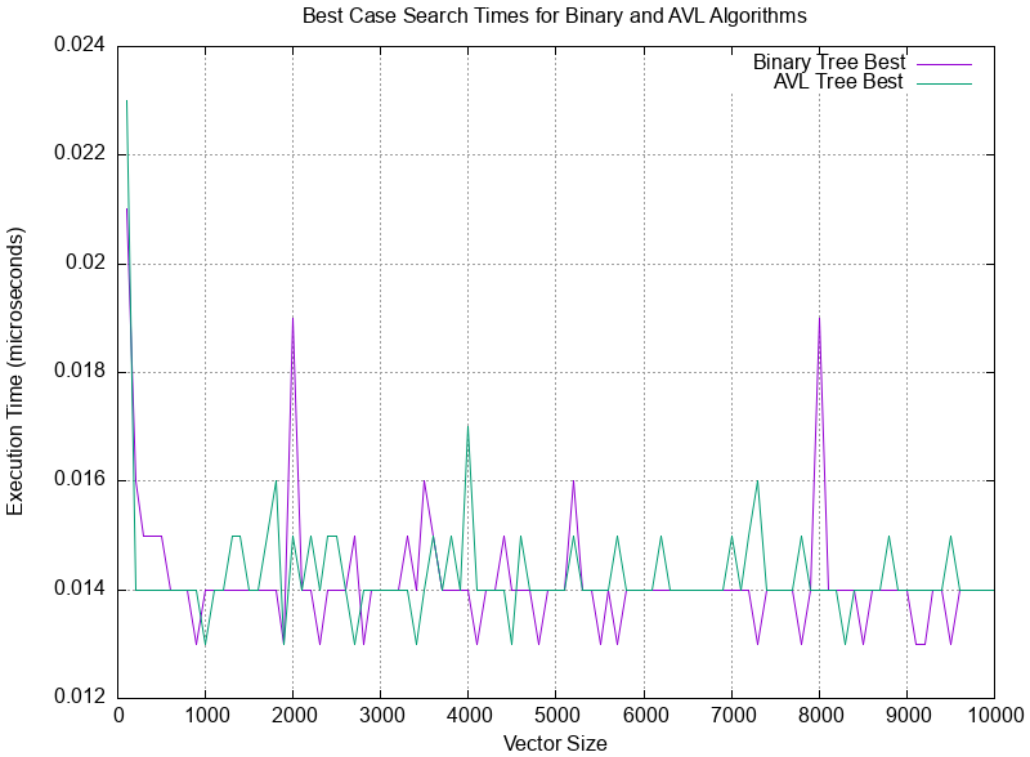

Figura 12: Gráfico de todos os melhores casos

Melhores Casos: De modo geral so se teve dois melhores casos tratados neste relatório que foram o da árvore binária e da árvore AVL por isso os tempos de execução são constantes e se mantem na mesma linha de como funcionam, pois a AVL como já dito anteriormente é uma extensão da árvore binária então os melhores casos funcionam exatamente de mesma forma e com tempo de execução semelhantes por buscarem apenas um numero na raiz de cada árvore, não existe muito oque comentar sobre mas isso so prova que realmente os algoritmos são semelhantes.

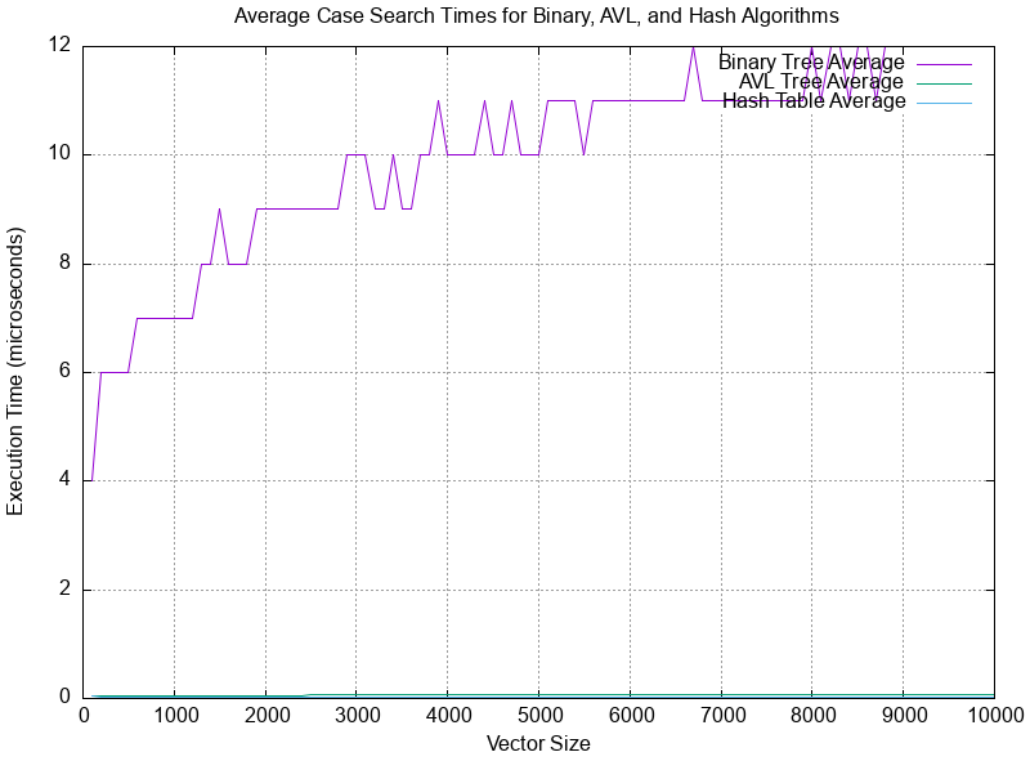

Figura 13: Gráfico de todos os médios casos

Médios Casos: Para o médio caso já começa a se diferenciar do anterior, já que da para se notar uma diferença gritante da árvore binária para as outras, que a AVL e a tabela hash nao chegaram a ter nem ao menos 1 microsegundo inteiro de tempo de execução enquanto, devido a imprevisibilidade de montagem da árvore binária sendo apenas parcialmente balanceada, mostra que mesmo sendo evidente a função logaritma ela ainda assim consegue ser muito mais lenta doque as demais que nem ao menos chegam a aparecer de forma clara no grafico de comparação, com isso mostrando que a árvore AVL sendo balanceada custea muito menos por ter uma limitação muito melhor para a busca doque da árvore sem ser totalmente balanceda, isso sem levar em conta a tabela hash que vai ser mais rápida ainda por ser constante mesmo nessa situação.

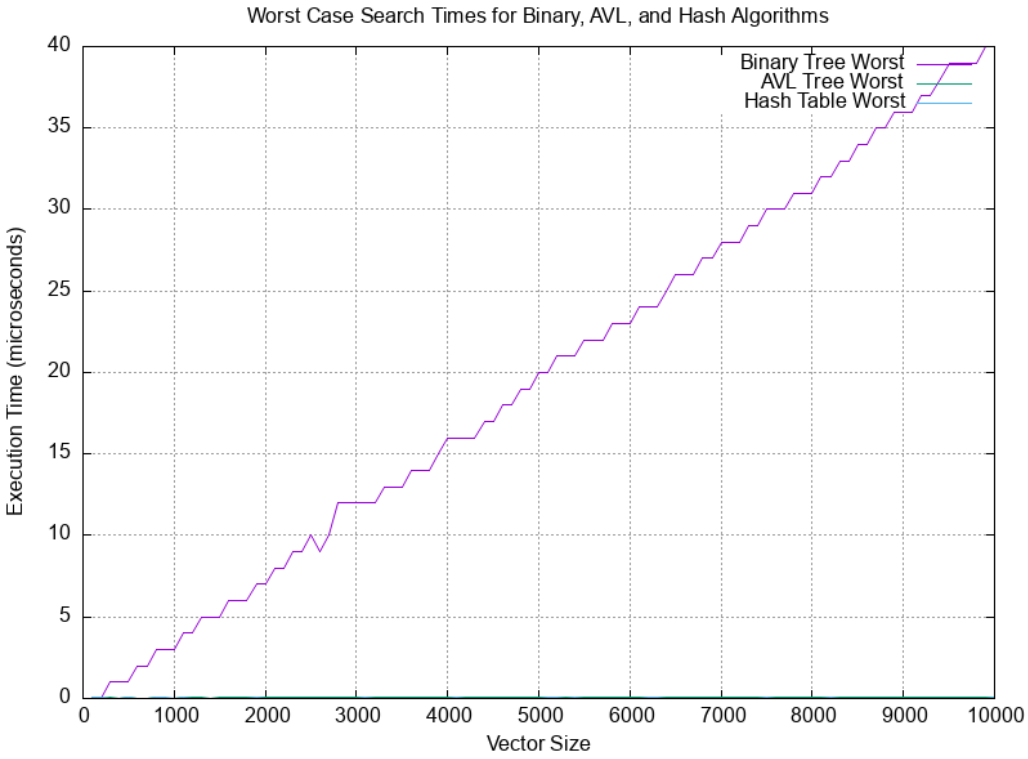

Figura 14: Gráfico de todos os piores casos

Piores Casos: Para os piores casos, oque já dava para ver que estava discrepante entre os gráficos no tempo médio, nos piores já da para ver melhor ainda que existe uma diferença muito grande entre os algoritmos já que os da árvore AVL e da tabela hash nem ao menos aparecem no grafico de comparação nos seus piores casos, que so mostra o quanto a árvore binária desbalanceada pode ser custosa em seu uso sem o controle de como pode-se organizar a mesma e se tornando totalmente inviável usá-la de maneira sem controle de como os nós irão se portar, e que mesmo assim mostrando esse gráfico so mostra que mesmo com seus piores casos, a AVL e tabela hash ainda assim se tornam opções viáveis já que para olhos humanos o tempo de execução da mesma mal iria dar para perceber o tempo demorado.

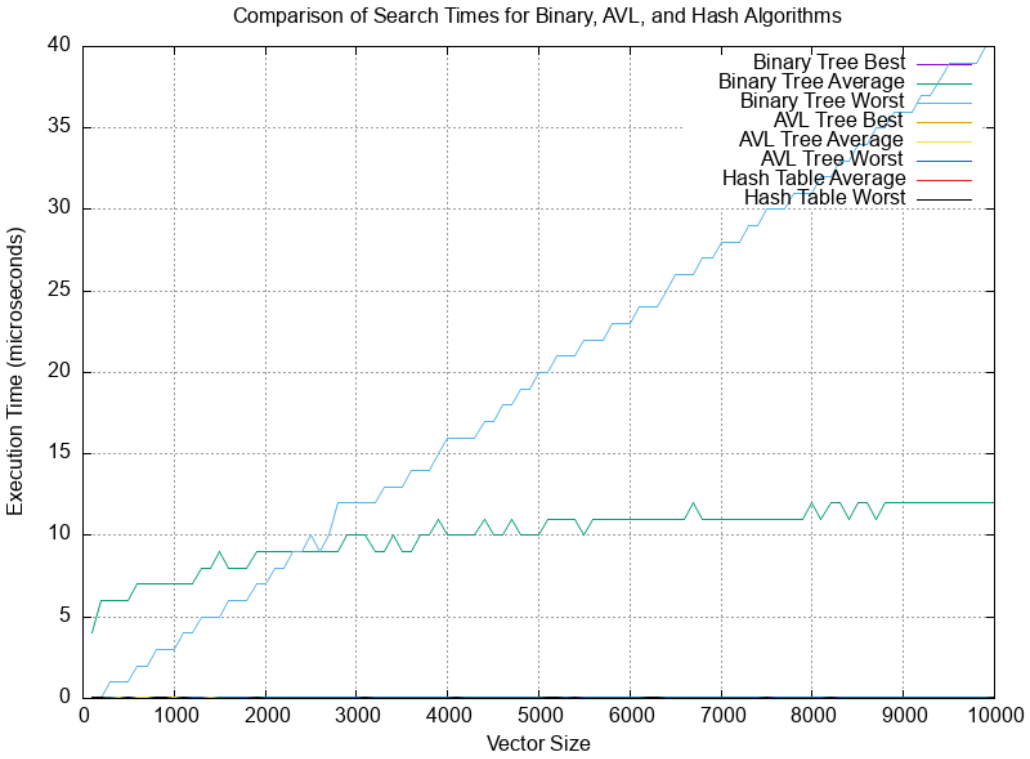

Figura 15: Gráfico de todos os casos

Agora para a análise de todos os casos juntos em um só grafico da para perceber que o pior e médio caso da árvore binária tomou conta do gráfico e mostrou apenas eles, isso se deve ao fato de realmente terem sido muito custosos em seu tempo de execução, pois enquanto maioria dos algoritmos foram em nanosegundos convertidos para microsegundos, os da árvore binária não necessitou desse tipo de conversão justamente por de fato ter um tempo muito maior de execução doque os demais, pois tirando o melhor caso que é constante já que so busca pela a raiz, os outros dois casos se mostram completamente lentos e não sendo geralmente indicados para nós sem controle, já que facilmente pode tornar a árvore desbalanceada e sem poder ter como ajustar sem implementar o método da árvore AVL, dado isso, para mostrar melhor graficamente foi criado outro gráfico no qual tem todos os casos dos algoritmos citados mas sem o pior e médio caso da árvore binária.

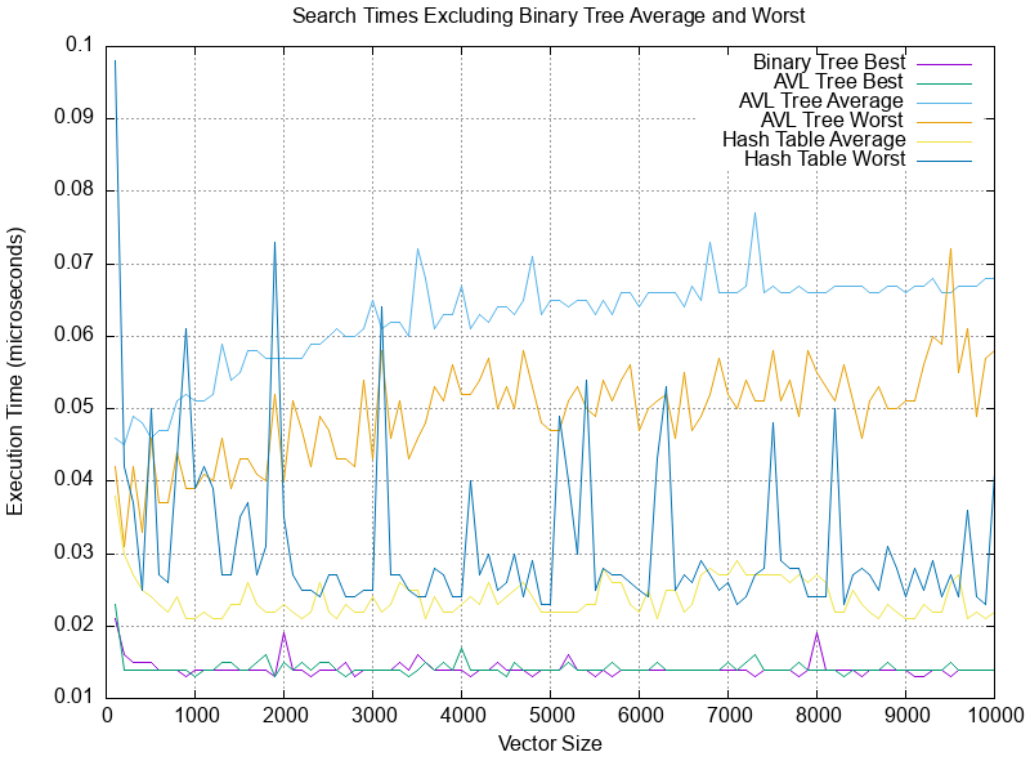

Figura 16: Gráfico de todos os casos sem o médio e pior da Binary Tree Search

Como anteriormente não dava para se analisar com totalidade a comparação de todos os gráficos por terem ficado "invisíveis"pelo alto custo de tempo de execução da árvore binária, a Figura 16 mostra com totalidade (tirando o pior e médio caso da árvore binária) todos os tempos de execução e que assim possibilitanto uma análise de qual foi mais rápido doque outro e assim por diante, e de modo geral da para se confirmar que realmente o melhor caso das árvores são os de tempo menor mas isso aplicados no cotidiano seria quase impossível existir pelo fato de apenas buscarem pela raiz de cada uma, logo após da para se ver os casos da tabela hash que realmente são bem rapidos mesmo no pior caso possuindo tempo de execução linear $O(n)$ ele ainda assim se mantem na mesma linha pelo algoritmo depender de chaves de busca que o tornam bastante eficiente mesmo com colisões, e por ultimo dá para se ver o médio e pior caso da árvore AVL que por terem o mesmo tempo de execução logaritmo $O(\log n)$ eles se mantem bastante semelhantes mesmo que para um ponto de vista computacional possa ser diferente ainda assim no ponto de vista humano é praticamente nulo a diferença dentre os dois.

E para demonstar de forma geral um ranking de melhor para o pior foi feito uma tabela dos tempos de execução de cada algoritmo, assim mostrando de forma geral quem custou mais e menos de acordo com seus resultados a seguir na Tabela 1:

Colocação	Algoritmos de Busca	Maior Tempo de Execução (microsegundos)
1°	Binary Tree(Melhor Caso)	0.014
2°	AVL Tree(Melhor Caso)	0.014
3°	Hash Table(Médio Caso)	0.022
4°	Hash Table(Pior Caso)	0.041
5°	AVL Tree(Pior Caso)	0.058
6°	AVL Tree(Médio Caso)	0.068
7°	Binary Tree(Médio Caso)	12
8°	Binary Tree(Pior Caso)	40

Tabela 1: Resultados do tempo médio de algoritmos de busca do menor para o maior

Por fim, uma tabela mostrando o resultado final de cada algoritmo de busca presente neste relatório, vale ressaltar que foi feito testes com apenas 10000 vetores em todos os experimentos, por maioria ser de tempo de execução logaritmo $O(\log n)$ é de se esperar que não haja uma mudança muito significativa com vetores maiores tirando o pior caso da árvore binária que levou um pouco mais de tempo para rodar já com 10000 vetores e que pode piorar caso seja testado com vetores maiores.

Para análise da tabela é normal pensar que o Hash Table seria a melhor opção pela proximidade dos seus dois casos analisados sem contar o melhor caso também que é constante, mas além de ele ser um tipo de algoritmo mais custoso para a memoria já que se existir colisões a memoria de acesso a essas informações é cada vez mais usada, ele ainda assim tem o pior caso sendo linear então mesmo que pelos testes no vetor de 10000 nao tenha sido grande e ficado perto da linha do caso médio que é constante, não tem como prever que ele funcionará assim com vetores maiores também então deve-se levar em conta isso também e que se a memória, como já dito, for um empecilho, não é nada recomendado o uso do Hash Table para busca, apesar de ser bastante eficiente.

4 Conclusão

Ao longo deste relatório foram analisados diversos algoritmos de busca, com foco na comparação dos tempos de execução em diversos cenários, incluindo os melhores, médios e piores casos, no qual o médio e pior caso foram mais levados em conta para comparações já que os melhores casos que se tem no relatório se tratam de tempos constantes. A análise evidenciou que cada algoritmo possui suas próprias vantagens e desvantagens, dependendo da aplicação e do contexto de uso, por isso de modo geral a conclusão chegada em cada algoritmo foi:

Árvore Binária: Embora apresente um tempo de execução constante em seu melhor caso, a árvore binária desbalanceada se mostrou sendo muito lenta em cenários médios e piores, com tempos de execução que podem tornar seu uso impraticável em aplicações que exigem desempenho consistente. Pela falta de balanceamento levou ela a um crescimento excessivo da profundidade, resultando em uma complexidade de tempo muito maior do que as demais.

Árvore AVL: Pelos testes feitos ela de fato se provou ser uma versão mais eficiente da árvore binária, mantendo o balanceamento automático e que garante um tempo de execução logaritmo $O(\log n)$ tanto no médio quanto no pior caso. Isso faz com que ela seja muito mais desejável em sistemas que dependem de um desempenho melhor e confiável, além de também poupar mais o uso da memória doque a tabela hash por exemplo, mesmo em grandes conjuntos de dados, ela nao tem o risco de piorar muito a performance.

Tabela Hash: Se destacou por sua eficiência em termos de tempo de execução no médio e pior caso, especialmente quando não há muitas colisões, mas que quando se tem, deve-se ter um uso adequado de algoritmos para conseguir lidar com essas colisões, por isso seu desempenho varia muito de como isso pode ser tratado, além disso seu uso mais intensivo de memória pode ser uma desvantagem em sistemas com recursos limitados.

Se baseando nas afirmações que foram ditas durante esse relatório inteiro e a análise de cada algoritmo, o melhor para se ter o uso e que além de ser eficiente também possui uma melhor adaptabilidade em qualquer tipo de sistema é a árvore AVL, pois dentre todos foi o unico que se mantém de fato estável e não tem perigo que em piores casos possam ficar de certa forma mais pesados devido a seu tempo de execução logaritmo $O(\log n)$, mas a tabela hash também não fica para trás so perde devido a suas limitações já citadas, por isso se for usada de forma decente também se torna uma opção melhor até que a árvore AVL em alguns casos, mas por consenso geral a árvore binária quase não é util para sistemas que precisam manter uma base de dados para busca e inserção de valores, por isso não se tem muito uso para a mesma.

Por fim, é essencial considerar o tamanho dos dados e a frequência das operações de busca ao escolher o algoritmo mais adequado, pois isso pode impactar diretamente na eficiência e no comportamento do sistema ao longo do tempo.

A Informações Complementares

Códigos usados no relatório:

```c
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
} Node;

Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}

Node* insert(Node* node, int key) {
    if (node == NULL) return createNode(key);
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
    return node;
}

Node* search(Node* root, int key) {
    if (root == NULL || root->key == key) return root;
    if (root->key < key) return search(root->right, key);
    return search(root->left, key);
}

void freeTree(Node* node) {
    if (node == NULL) return;
    freeTree(node->left);
    freeTree(node->right);
    free(node);
}
```

Listing 1: Implementação de Árvore Binária em C

```c
typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

int height(Node *N) {
    return (N == NULL) ? 0 : N->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

Node *rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

Node *leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

int getBalance(Node *N) {
    return (N == NULL) ? 0 : height(N->left) - height(N->right);
}

Node* insert(Node* node, int key) {
    if (node == NULL) return newNode(key);
    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
    else return node;

    node->height = 1 + max(height(node->left), height(node->right));
    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key) return rightRotate(node);
    if (balance < -1 && key > node->right->key) return leftRotate(node);
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
```

Listing 2: Exemplo de Implementação de Árvore AVL em C

```c
typedef struct node {
    int value;
    struct node* next;
} Node;

typedef struct hashTable {
    unsigned int m; // Tamanho da tabela
    unsigned int n; // Quantidade de Elementos
    Node** nodes;
} HashTable;

void insert(HashTable* hashTable, int value);
void reHash(HashTable *hashTable);
int search(HashTable *hashTable, int value);

unsigned int hash(int value, int m) {
    return value % m < 0 ? (value % m) + m : value % m;
}

void reHash(HashTable* hashTable) {
    unsigned int old_m = hashTable->m;
    Node** old_nodes = hashTable->nodes;

    hashTable->m *= 2;
    hashTable->n = 0;
    hashTable->nodes = (Node**)malloc(sizeof(Node*) * hashTable->m);

    for(int i = 0; i < hashTable->m; i++) {
        hashTable->nodes[i] = NULL;
    }

    for(int i = 0; i < old_m; i++) {
        Node* aux = old_nodes[i];
        while(aux != NULL) {
            insert(hashTable, aux->value);
            Node* toFree = aux;
            aux = aux->next;
            free(toFree);
        }
    }
    free(old_nodes);
}

float loadFactor(HashTable* hashTable) {
    return (float)hashTable->n / hashTable->m;
}

Node* create_node(int value) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->value = value;
    node->next = NULL;
    return node;
}

HashTable* create_table(unsigned int m) {
    HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
    hashTable->m = m;
    hashTable->n = 0;
    hashTable->nodes = (Node**)malloc(sizeof(Node*) * m);

    for(int i = 0; i < m; i++) {
        hashTable->nodes[i] = NULL;
    }

    return hashTable;
}

void insert(HashTable* hashTable, int value) {
    unsigned int key = hash(value, hashTable->m);

    if(loadFactor(hashTable) < 1) {
        unsigned int index = hash(key, hashTable->m);
        Node* new_node = create_node(value);

        if(hashTable->nodes[index] == NULL) {
            hashTable->nodes[index] = new_node;
        } else {
            Node* index_node = hashTable->nodes[index];
            while(index_node->next != NULL) {
                index_node = index_node->next;
            }
            index_node->next = new_node;
        }
        hashTable->n++;
    } else {
        reHash(hashTable);
        insert(hashTable, value);
    }
}
```

Listing 3: Implementação de Tabela Hash em C