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

Modelo de Referência para Escrita de Monografias e Relatórios do LabEPI

Vinicius Maia Marinho

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 atividade de Estágio Obrigatório.

Laboratório de Elementos do Processamento da Informação – LabEPI
Caicó, RN, 3 de julho de 2023

UFRN / Biblioteca Central Zila Mamede.

Catalogação da Publicação na Fonte.

Aluno, Vinicius Maia Marinho

Modelo de Referência para Escrita de Monografias e Relatórios do LabEPI. / Nome Vinicius Maia Marnho. – Caicó, RN, 2023.

16 f.: il.

Orientador: Prof. Dr. João Paulo de Souza Medeiros.

Relatório Técnico – Universidade Federal do Rio Grande do Norte. Centro de Ensino Superior do Seridó. Bacharelado em Sistemas de Informação.

1. Estrutura de dados. 2. Algorítmos de busca. 3. Relatório Técnico. I. Professor, João Paulo II. Universidade Federal do Rio Grande do Norte. III. Relatório Técnico para a disciplica de estrutura de dados.

RN/UF/BCZM CDU 004.7

Resumo

Este trabalho apresenta uma análise dos algoritmos de ordenação: distribution, insertion, merge, quick e selection. O objetivo é compreender o funcionamento de cada algoritmo e avaliar o tempo de execução deles.

Palavras-chave: Estrutura de dados. Algorítmos de ordenação. Relatório Técnico.

Abstract

This work presents an analysis of sorting algorithms: distribution, insertion, merge, quick and selection. The goal is to understand the operation of each algorithm and evaluate their execution time.

Keywords: Data structures; Order algorithms; Technical report.

Sumário

Lista de Algoritmos

1. Introdução

"If knowledge can create problems,
it is not through ignorance that we can solve them."
Isaac Asimov

Este relatório tem como objetivo apresentar os resultados obtidos por meio da análise do tempo de execução de diferentes algoritmos de estruturas de dados. As estruturas de dados são elementos fundamentais no campo da computação, pois permitem a organização e manipulação eficiente de informações.

Neste estudo, foram analisadas as seguintes estruturas de dados: árvore binária, árvore AVL e tabela hash. Cada uma dessas estruturas possui características distintas em relação ao tempo de execução e complexidade, o que nos permite compará-las e identificar suas eficiências em diferentes cenários.

O objetivo principal deste relatório é fornecer uma visão geral do desempenho de cada estrutura de dados em relação ao tempo de execução, com base em uma série de testes realizados em conjuntos de dados de diferentes tamanhos. Para isso, foram registrados os tempos de execução de cada estrutura em diferentes situações, permitindo uma análise comparativa.

A análise do tempo de execução dessas estruturas de dados é de grande importância, pois influencia diretamente a eficiência e escalabilidade dos algoritmos que as utilizam. Compreender como essas estruturas se comportam em relação ao tempo de execução nos permite tomar decisões mais embasadas na escolha da estrutura adequada para cada aplicação, levando em consideração os requisitos e restrições específicas de cada cenário.

Ao final deste relatório, será possível identificar quais estruturas de dados são mais eficientes em determinados contextos, auxiliando no processo de seleção e otimização de algoritmos em projetos de desenvolvimento de software.

2. Árvore binária

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."
Alan Mathison Turing

2.1 Introdução

Uma árvore binária é uma estrutura de dados hierárquica composta por nós interconectados. Cada nó pode ter no máximo dois filhos: um à esquerda, outro à direita e o nó do topo é conhecido como raiz ou nó pai. A árvore binária possui uma propriedade importante, onde o valor de cada nó à esquerda é menor do que o valor do nó pai, e o valor de cada nó à direita é maior do que o valor do nó pai. A árvore binária permite a busca, inserção e remoção eficientes de elementos, tornando-a uma estrutura de dados valiosa em diversas aplicações.

Algoritmo 2.1 (Busca em árvore binária)**.** A busca em uma árvore binária é realizada de forma recursiva, comparando o valor buscado com o valor do nó atual e seguindo pela subárvore esquerda ou direita.

algoritmo TreeSearch(root, value)
1: **se** root $\neq$ NULL **então**
2: **se** root→value = value **então**
3: **retorne** root
4: **else se** value < root→value **então**
5: **retorne** TreeSearch(root→lchild), value)
6: **else**
7: **retorne** TreeSearch(root→lchild), value)
8: **fim se**
9: **fim se**
10: **retorne** NULL

□

2.2 Algoritmo de inserção

Para inserir um novo elemento em uma árvore binária, o algoritmo percorre a árvore de forma recursiva, comparando o valor a ser inserido com o valor do nó atual. Se o valor for menor, o algoritmo segue pela subárvore esquerda; se for maior, segue pela subárvore direita. Quando encontra uma posição vazia, o novo elemento é inserido como um novo nó. A complexidade de tempo da inserção em uma árvore binária balanceada é $O(\log n)$, onde

n é o número de elementos na árvore. Isso significa que o tempo de execução aumenta de forma logarítmica à medida que o tamanho da árvore cresce, tornando-a adequada para lidar com grandes conjuntos de dados.

Algoritmo 2.2 (Inserção em árvore binária)**.** **algoritmo** tree_insert(root, w)

```
1: se root = NULL então
2:     root ← w
3: else
4:     se root− > value < w → value então
5:         tree_insert(root→rchild), w)
6:     else
7:         tree_insert(root→lchild), w)
8:     fim se
9: fim se
```

Algoritmo 2.1: Inserção em árvore binária

□

2.3 Conclusão

A árvore binária é uma estrutura de dados eficiente para realizar operações de busca, inserção e remoção. Sua capacidade de organizar os elementos de forma hierárquica, seguindo a propriedade de ordenação, permite um acesso rápido aos dados. No entanto, é importante manter a árvore balanceada para garantir um desempenho ideal. Compreender o funcionamento e as características da árvore binária é essencial para a implementação de algoritmos eficientes e escaláveis

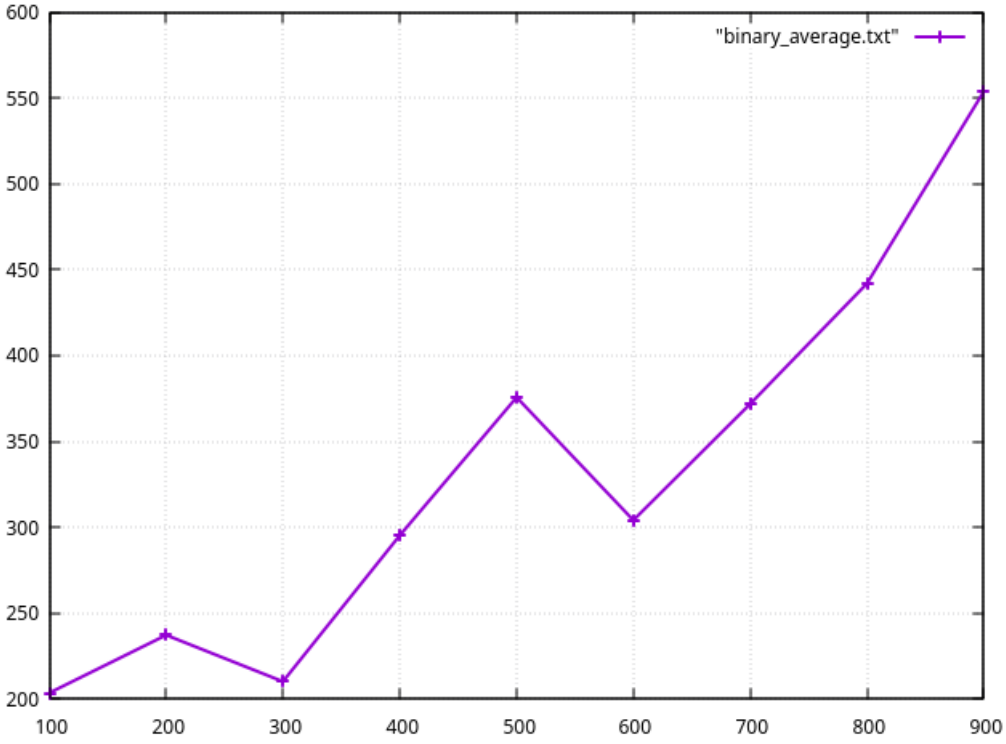

Figura 2.1: Tempo médio da busca em árvore binária

3. Árvore AVL

"Mathematical elegance is not a dispensable luxury
but a factor that decides between success and failure."
Edsger Wybe Dijkstra

3.1 Introdução

A árvore AVL é uma variação da árvore binária de busca que possui um recurso adicional: o balanceamento automático. Essa propriedade faz com que a árvore AVL seja capaz de manter-se sempre balanceada, garantindo que a altura das subárvores esquerda e direita de cada nó difira no máximo em uma unidade. O balanceamento da árvore AVL é realizado automaticamente durante as operações de inserção e remoção de elementos. Sempre que uma alteração é feita na árvore, são realizadas rotações simples ou duplas para reequilibrar os nós afetados. Dessa forma, a árvore AVL evita o problema da degeneração, onde uma árvore desbalanceada pode se transformar em uma lista encadeada, resultando em uma complexidade de tempo linear.

Algoritmo 3.1 (Busca em árvore AVL)**.** **algoritmo** search(root, value)

```
1: se root ≠ NULL então
2:     se root → value = value então
3:         retorne root
4:     else se root → value > value então
5:         retorne search(root→lchild), value)
6:     else
7:         retorne search(root→rchild), value)
8:     fim se
9: fim se
10: retorne NULL
```

Algoritmo 3.1: Busca em árvore avl

□

3.2 Algoritmo de inserção e rotação

O algoritmo de inserção em uma árvore AVL segue o mesmo princípio da árvore binária de busca. No entanto, após a inserção, são verificadas as propriedades de balanceamento

da árvore. Se alguma dessas propriedades for violada, são aplicadas rotações para reequilibrar a estrutura. As rotações podem ser simples ou duplas, dependendo do tipo de desbalanceamento encontrado. Através dessas rotações, os nós são rearranjados de forma a garantir que a árvore AVL mantenha-se balanceada. A complexidade de tempo da inserção em uma árvore AVL balanceada é O(log n), assim como na árvore binária. No entanto, devido ao balanceamento automático, a árvore AVL garante uma altura máxima logarítmica, o que resulta em um desempenho mais eficiente para as operações de busca, inserção e remoção.

Algoritmo 3.2 (Busca em árvore AVL)**.** **algoritmo** insert(aux, node)

```
1: se aux = NULL então
2:     aux ← node
3:     balance(aux, aux)
4: else
5:     node → father ← aux
6:     se aux → value > node → value então
7:         insert(aux→lchild, node)
8:     else
9:         insert(aux→rchild, node)
10:    fim se
11: fim se
```

Algoritmo 3.2: Inserção em árvore avl

□

Algoritmo 3.3 (Balanceamento em árvore avl)**.** **algoritmo** balance(node, root)

```
1: node ← node → father
2: enquanto node ≠ NULL faça
3:     node → height ← height(node)
4:     se differenceHeight(node) > 1 então
5:         check ← whichCase(node)
6:         se check = 1 então
7:             retorne rotateRight(&node)
8:         else se check = 2 então
9:             retorne rotateLeft(&node)
10:        else se check = 3 então
11:            rotateLeft(&(node→lchild))
12:            rotateRight(&node)
13:        else
14:            rotateRight(&(node→rchild))
15:            rotateLeft(&node)
16:        fim se
17:    fim se
18:    node ← node → father
19: fim enquanto
```

Algoritmo 3.3: Balanceamento de árvore avl

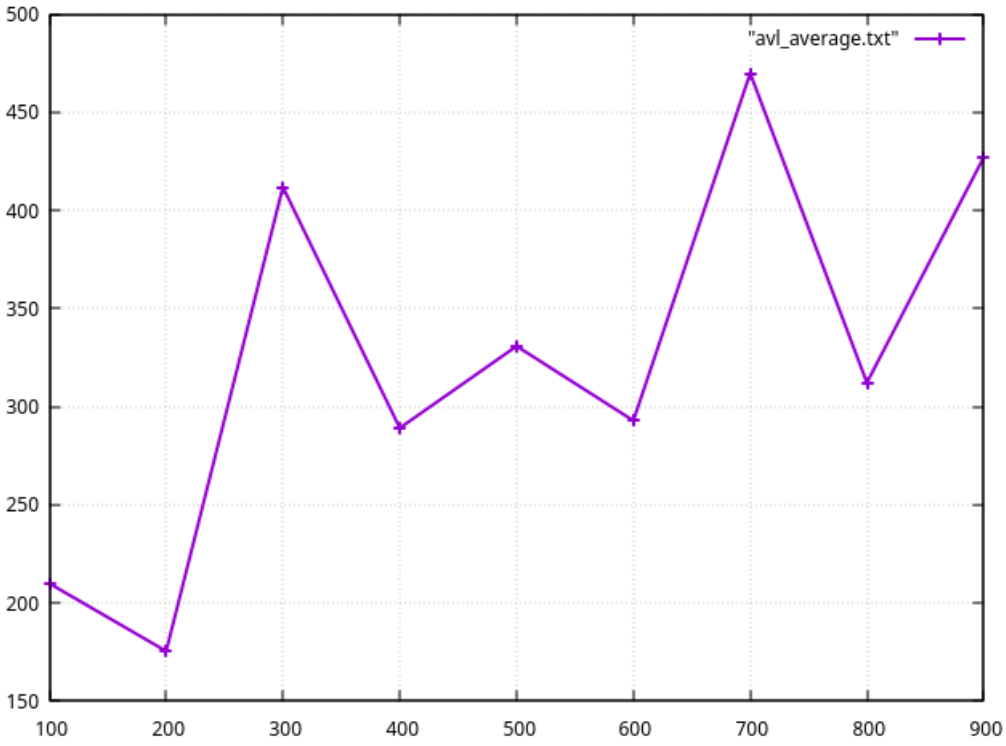

Figura 3.1: Tempo médio da busca em árvore avl

□

3.3 Conclusão

A árvore AVL é uma estrutura de dados avançada que oferece o benefício adicional do balanceamento automático. Isso garante uma altura balanceada e, consequentemente, um desempenho otimizado em operações de busca, inserção e remoção. A compreensão do funcionamento e das rotações da árvore AVL é essencial para a implementação de algoritmos eficientes e escaláveis. Ao utilizar a árvore AVL, é possível alcançar um equilíbrio perfeito entre a estrutura de dados e as operações realizadas, garantindo um desempenho consistente mesmo para grandes conjuntos de dados. A árvore AVL representa um avanço em relação à árvore binária de busca tradicional, proporcionando um melhor controle da altura e um tempo de execução mais estável. Sua utilização é especialmente indicada em situações em que é necessário lidar com operações frequentes de inserção e remoção, mantendo a estrutura sempre balanceada e eficiente.”

4. Tabela Hash

"If we can really understand the problem,
the answer will come out of it,
because the answer is not separate from the problem."
Jiddu Krishnamurti

4.1 Introdução

A tabela hash é uma estrutura de dados que permite o armazenamento e recuperação eficiente de informações. Ela utiliza uma função de dispersão (hash function) para mapear chaves em índices de uma tabela. Dessa forma, é possível acessar os elementos diretamente através de suas chaves, sem a necessidade de percorrer a estrutura de dados. A função de dispersão é responsável por gerar um valor único para cada chave. Esse valor é usado como índice na tabela hash, onde o elemento correspondente à chave é armazenado. Em casos ideais, a função de dispersão distribui uniformemente as chaves na tabela, minimizando as colisões.

Algoritmo 4.1 (Busca em tabela hash)**.** **algoritmo** search(hashTable, key)
1: $index \leftarrow$ hashFunction(key, hashTable $\rightarrow$ size)
2: struct list_node item $\leftarrow$ hashTable $\rightarrow$ table[$index$]
3: **enquanto** item $\neq$ NULL **faça**
4: **se** item $\rightarrow$ value = key **então**
5: **retorne** 1
6: **fim se**
7: item $\leftarrow$ item $\rightarrow$ next
8: **fim enquanto**
9: **retorne** 0

Algoritmo 4.1: Search

□

4.2 Algoritmo de inserção e recuperação

O algoritmo de inserção em uma tabela hash consiste em aplicar a função de dispersão à chave do elemento e encontrar o índice correspondente na tabela. Se o índice estiver vazio, o elemento é inserido. Caso contrário, ocorre uma colisão, e uma estratégia de tratamento de colisão é utilizada para resolver o problema. Existem várias estratégias de

tratamento de colisão, como encadeamento separado e endereçamento aberto. No encadeamento separado, cada posição da tabela contém uma lista encadeada de elementos com chaves que geraram o mesmo índice. No endereçamento aberto, são exploradas posições alternativas na tabela até encontrar uma vaga disponível. A recuperação de elementos em uma tabela hash também é feita através da função de dispersão. A chave do elemento é usada para calcular o índice correspondente, e o elemento é retornado se estiver presente na tabela.

Algoritmo 4.2 (Rehash). **algoritmo** rehash(hashTable)

```
1: newSize ← hashTable → size × 2
2: oldTable ← hashTable → table
3: hashTable → size ← newSize
4: hashTable → table ← malloc(sizeof(struct list_node*) × newSize)
5: para i ← 0 to newSize faça
6:     hashTable → table[i] ← NULL
7: fim para
8: para i ← 0 to hashTable → size/2 faça
9:     struct list_node* item ← oldTable[i]
10:    enquanto item ≠ NULL faça
11:        next ← item → next
12:        index ← hashFunction(item → value, hashTable → size)
13:        item → next ← hashTable → table[index]
14:        hashTable → table[index] ← item
15:        item ← next
16:    fim enquanto
17: fim para
18: free(oldTable)
```

Algoritmo 4.2: Rehash

□

Algoritmo 4.3 (Inserção em tabela hash). **algoritmo** insert(hashTable, value)

```
1: index ← hashFunction(value, hashTable → size)
2: se hashTable → size = hashTable → n então
3:     rehash(hashTable)
4:     index ← hashFunction(value, hashTable → size)
5: fim se
6: newNode ← malloc(sizeof(struct list_node))
7: newNode → value ← value
8: newNode → next ← hashTable → table[index]
9: hashTable → table[index] ← newNode
10: hashTable → n ← hashTable → n + 1
```

Algoritmo 4.3: Insert

□

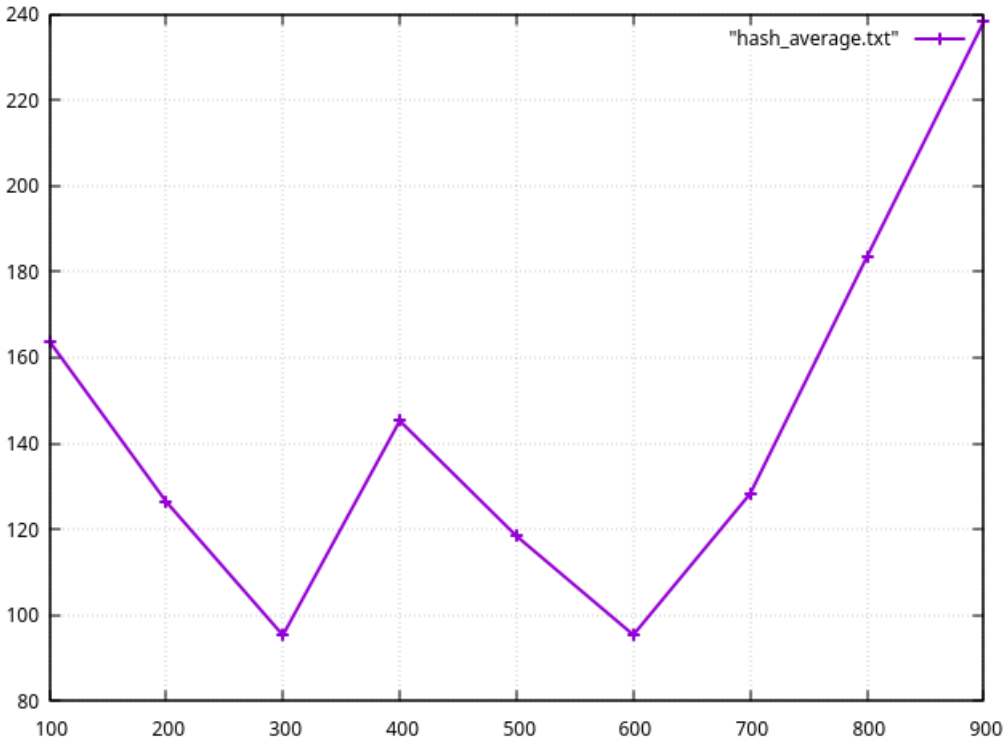

Figura 4.1: Tempo médio da busca em tabela hash

4.3 Conclusão

A tabela hash é uma estrutura de dados eficiente para armazenamento e recuperação de informações. Ela permite um acesso rápido aos elementos, tornando-a ideal para aplicações em que a velocidade é essencial. No entanto, o desempenho da tabela hash depende da função de dispersão utilizada e da estratégia de tratamento de colisão adotada. Ao utilizar uma tabela hash, é importante escolher uma função de dispersão adequada que minimize as colisões e distribua uniformemente as chaves na tabela. Além disso, a escolha da estratégia de tratamento de colisão correta também é fundamental para garantir a eficiência da estrutura. Compreender os princípios e algoritmos por trás da tabela hash é essencial para a implementação de sistemas eficientes que lidam com grandes quantidades de dados. A tabela hash oferece uma abordagem poderosa para o armazenamento e acesso rápido a informações, sendo amplamente utilizada em uma variedade de aplicações computacionais.

5. Conclusão

"If we can really understand the problem,
the answer will come out of it,
because the answer is not separate from the problem."
Jiddu Krishnamurti

5.1 Conclusão

Concluido, as estruturas de dados representam um importante papel para toda a computação, criando um inumero leque de possiilidades já que cada uma possui caracterísiticas e complexidades próprias.

Durante o trabalho foram exploradas algumas das estruturas apresentadas durantes as aulas, como a árvore binária, a árvore binária balanceada (AVL) e a tabela de dispersão (hash). Cada um desses algoritmos tem seus pontos fortes e fracos, e é importante entender essas características ao decidir qual utilizar.

A árvore binária é uma boa escolha quando os dados estão ordenados, pois ela permite uma busca eficiente de ordem logarítmica. No entanto, se a árvore estiver desbalanceada, o tempo de busca pode se tornar linear, o que é uma desvantagem em termos de desempenho.

A árvore binária balanceada, como a AVL, oferece um tempo de busca garantidamente logarítmico, independentemente da distribuição dos dados. Isso ocorre devido às restrições aplicadas à estrutura da árvore, garantindo que as alturas entre os nós nunca sejam maiores do que um. Embora a AVL possa exigir um pouco mais de tempo e espaço para operações de inserção e remoção, ela é uma ótima opção quando a ordem dos dados é desconhecida ou pode variar significativamente.

A tabela de dispersão, por sua vez, oferece um desempenho de busca constante no caso esperado, ou seja, em situações em que não ocorrem colisões. A busca é feita através de uma função de dispersão que mapeia a chave para uma posição na tabela, onde o valor correspondente pode ser encontrado. No entanto, no pior caso, quando ocorrem colisões e várias chaves são mapeadas para a mesma posição, o desempenho pode se tornar linear, prejudicando a eficiência do algoritmo.

Ao escolher uma estrutura de dados, é fundamental considerar as características específicas do conjunto de dados, como tamanho, ordem e possibilidade de ocorrência de colisões, a fim de garantir o melhor desempenho possível