T1 - INF1010
JÚLIA TADEU - 2312392
THEO CANUTO - 2311293

## OBJETIVOS:

- Implementar uma função de hash com endereçamento aberto para armazenar 1000 CPFs, visando acesso rápido e eficiente

- Avaliar o desempenho do método por meio da contagem de colisões e posições vazias, medindo a eficiência da tabela hash

- Gerar um gráfico relacionando o número de chaves inseridas e o número de colisões, em incrementos de 100 até 1000 chaves, para análise visual do comportamento da tabela.

- Análise de Complexidade: Comparar o desempenho do hashing com O(log n), visando demonstrar a eficiência do método em comparação com estruturas de dados balanceadas, como uma árvore AVL.

## TENTATIVA LINEAR

1. Cálculo do Índice: Utiliza a função de hash para obter o índice inicial.

2. Colisão Detectada: Se a posição está ocupada, inicia a sondagem linear.

3. Tentativa Linear: Incrementa o índice e aplica o módulo do tamanho da tabela para continuar procurando na tabela circular até encontrar uma posição vazia.

4. Contabilização de Colisões: Cada tentativa é registrada como uma colisão até encontrar um espaço livre.

5. Inserção Final: Insere o CPF na primeira posição vazia encontrada através da sondagem linear.

# TENTATIVA QUADRÁTICA

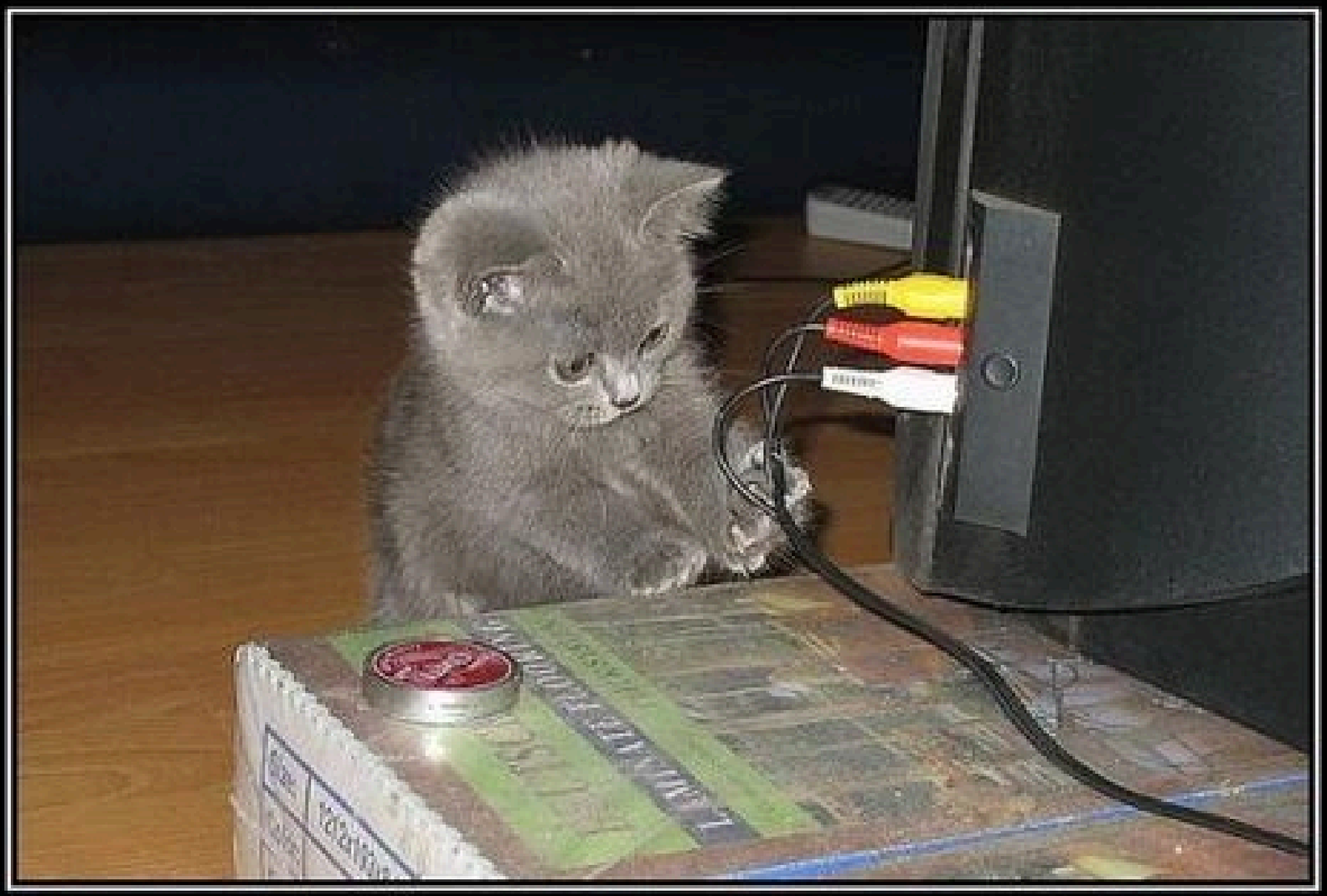
1. Cálculo do Índice: Utiliza a função de hash para obter o índice inicial.

2. Colisão Detectada: Se a posição está ocupada, inicia a sondagem quadrática.

3. Tentativa Quadrática: Incrementa de acordo com uma função quadrática do tipo $h(i)=(h0 + c1 \times i + c2 \times i^2) \bmod m$, onde h0 é o índice inicial, i é o número de tentativas e m é o tamanho da tabela. Essa função gera uma sequência crescente, mas com saltos quadráticos, o que reduz a probabilidade de colisões secundárias.

4. Contabilização de Colisões: Cada tentativa é registrada como uma colisão até encontrar um espaço livre.

5. Inserção Final: Insere o CPF na primeira posição vazia encontrada através da sondagem quadrática.



# WAIT

i'll fix it

## MÉTODO ESCOLHIDO: DISPERSÃO DUPLA

1. Cálculo do Índice Inicial: Utiliza `hash_function1`, que extrai partes do CPF e aplica multiplicação por um número primo.

2. Colisão Detectada: Se a posição está ocupada, ativa o hashing duplo.

3. Cálculo do Passo de Sondagem: Utiliza `hash_function2` para definir um step fixo com base em um segundo número primo, permitindo nova tentativa em uma posição diferente.

4. Sondagem com Hashing Duplo: Aplica a fórmula (index + j * step) mod table_size até encontrar uma posição vazia.

5. Contabilização de Colisões: Cada tentativa é registrada, e o número total de colisões por CPF é salvo em "colisoes_por_cpf.txt".

6. Inserção Final: Insere o CPF na primeira posição livre encontrada através do hashing duplo.

# PORQUE DISPERSÃO DUPLA?

A dispersão dupla foi escolhido neste contexto porque oferece uma solução eficaz para reduzir colisões ao armazenar grandes volumes de CPFs em uma tabela hash com fator de ocupação alto. Com o hashing duplo, duas funções de hash distintas são utilizadas para calcular o índice e o passo de sondagem, evitando que colisões consecutivas criem clusters que prejudicariam o desempenho.

Essa abordagem é particularmente adequada para situações onde a tabela hash contém muitos elementos e visa-se manter a eficiência das operações de inserção e busca. No caso deste trabalho, o uso do hashing duplo buscou proporcionar uma distribuição mais uniforme dos CPFs, visando um desempenho consistente e menor número de colisões comparado a outras técnicas, como a sondagem linear e a quadrática.

## DISPERSÃO DUPLA

Primo 1: 1153
Primo 2: 409
Tamanho da tabela: 1229
Número total de Colisões: 793
Maior colisão: 21 (posição 935)
Posições vazias: 229
Fator de ocupação:  81.37%


**função hash para cálculo do índice:**
$h'(cpf) = (((cpf/10000)\ \&\ 0xFFFF) \times 1153 + (cpf \bmod 10000)) \bmod 1229$

**função hash para cálculo do passo em caso de colisão:**
$h''(cpf) = (409 - (cpf \bmod 409))$ **ou** $h''(cpf) = (409 - (cpf \bmod 409)) + 1$
**para garantir que o resultado seja ímpar**

# PORQUE TABLE_SIZE = 1229 ?
# PORQUE PRIME_NUMBER1 = 1153 ?
# PORQUE PRIME_NUMBER2 = 409 ?

- O uso de números primos favorece a dispersão dos dados, reduzindo o risco de colisões por evitar padrões de repetição nos índices gerados pelas funções de hash. Isso resulta em uma distribuição mais uniforme e no uso mais eficaz do espaço disponível na tabela.

- Para table_size, visamos nos aproximar de 80% de ocupação da tabela. Logo, poderíamos usar números primos próximos a 1225. Isso nos gerou 2 opções: 1223 ou 1229.

- Para otimizar nossa escolha, decidimos desenvolver um algoritmo capaz de realizar todas as combinações possíveis de números primos de 2 a 1213 (menor primo antes de 1223).

Segue o algoritmo:

```c
int test_combinations(long long cpfs[], long long cpf_count, int table_size) {
    int primes[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,
                101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193,
                197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307,
                311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421,
                431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547,
                557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659,
                661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
                809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929,
                937, 941, 947, 953, 967, 971, 977, 983, 991, 997, 1009, 1013, 1019, 1021, 1031, 1033, 1039,
                1049, 1051, 1061, 1063, 1069, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129, 1151, 1153,
                1163, 1171, 1181, 1187, 1193, 1201, 1213 };
    int prime_count = sizeof(primes) / sizeof(primes[0]);

    FILE* file = fopen("combinacoes_resultados.txt", "w");

    if (!file) {
        printf("Erro ao abrir o arquivo de resultados\n");
        return -1;
    }
    long long* hash_table = (long long*)malloc(table_size * sizeof(long long));
    if (!hash_table) {
        printf("Erro de memoria\n");
        return -1;
    }

    for (int i = 0; i < prime_count; i++) {
        for (int j = 0; j < prime_count; j++) {
            initialize_table(hash_table, table_size);
            int collisions = insert_cpfs(cpfs, cpf_count, hash_table, table_size, primes[i], primes[j]);
            fprintf(file, "Tabela tamanho:\t%d\tPrimo1:\t%d\tPrimo2:\t%d\tColisoes:\t%d\n",
                table_size, primes[i], primes[j], collisions);
        }
    }

    free(hash_table);
    fclose(file);
    return 0;
}
```

OBS.: o código completo para esse teste foi anexado como teste_colisoes.c

# LEITURA DE CPFS 100 A 100

100

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 6
numero de posicoes vazias: 1129
fator de ocupacao: 8.14%
```

200

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 17
numero de posicoes vazias: 1029
fator de ocupacao: 16.27%
```

300

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 50
numero de posicoes vazias: 929
fator de ocupacao: 24.41%
```

400

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 84
numero de posicoes vazias: 829
fator de ocupacao: 32.55%
```

500

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 145
numero de posicoes vazias: 729
fator de ocupacao: 40.68%
```

# LEITURA DE CPFS 100 A 100

600

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 225
numero de posicoes vazias: 629
fator de ocupacao: 48.82%
```

700

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 311
numero de posicoes vazias: 529
fator de ocupacao: 56.96%
```

800

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 428
numero de posicoes vazias: 429
fator de ocupacao: 65.09%
```

900

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 573
numero de posicoes vazias: 329
fator de ocupacao: 73.23%
```

# OUTPUT FINAL

```
primo 1: 1153 - primo 2: 409 - tamanho da tabela: 1229
numero de colisoes: 793
numero de posicoes vazias: 229
fator de ocupacao: 81.37%
```

Agora vamos ver uma parte da nossa tabela hash, sinalizando os vazios...

# OUTPUT FINAL: COMO OS CPFS FICARAM ARMAZENADOS?

Primeiras posições:

```
posicao 0: 19015207810
posicao 1: Vazio
posicao 2: 32132838356
posicao 3: 7437959895
posicao 4: Vazio
posicao 5: 93243423716
posicao 6: 55486796781
posicao 7: 72250255466
posicao 8: 51850480508
posicao 9: 98442874305
posicao 10: 78883115635
posicao 11: 48042000294
posicao 12: 4287808908
posicao 13: 17528499498
posicao 14: 21215493312
posicao 15: 5439201270
posicao 16: 76787076061
posicao 17: 41686375492
posicao 18: Vazio
posicao 19: Vazio
posicao 20: Vazio
posicao 21: Vazio
posicao 22: 96649514470
posicao 23: Vazio
posicao 24: 29881506565
posicao 25: 25562328976
```

```
posicao 26: 98820524015
posicao 27: 10141835885
posicao 28: 10748359648
posicao 29: 96133094974
posicao 30: 20742920070
posicao 31: 74844055585
posicao 32: 62845906285
posicao 33: 11994552298
posicao 34: 63796726801
posicao 35: 58433562134
posicao 36: 79216996420
posicao 37: 64161265913
posicao 38: 83444876416
posicao 39: 35665172243
posicao 40: 66286332847
posicao 41: 4011701106
posicao 42: 81738124207
posicao 43: Vazio
posicao 44: 72950756905
posicao 45: 859131386
posicao 46: Vazio
posicao 47: 51486612083
posicao 48: 54791945425
posicao 49: 11672100690
posicao 50: Vazio
```

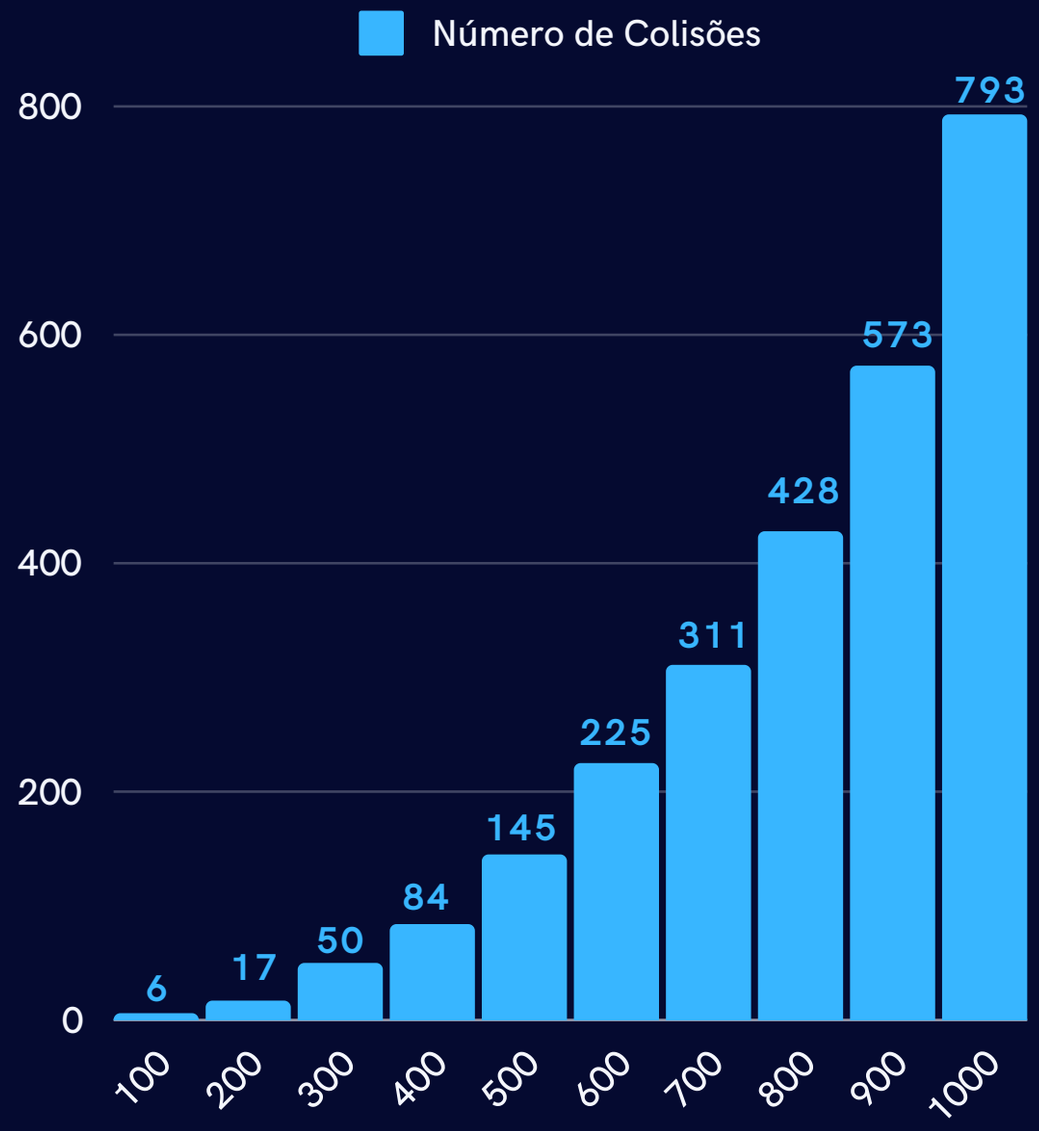
# OUTPUT FINAL: COMO OS CPFS FICARAM ARMAZENADOS?

... Últimas posições:

```
posicao 1174: 32742153411
posicao 1175: 98669439506
posicao 1176: 54418502483
posicao 1177: 62040169970
posicao 1178: 55945614980
posicao 1179: Vazio
posicao 1180: 69748559661
posicao 1181: 14065443610
posicao 1182: 12521085081
posicao 1183: 44899931298
posicao 1184: 64935469072
posicao 1185: 5330901561
posicao 1186: 44428972040
posicao 1187: 85951843995
posicao 1188: 88378968758
posicao 1189: 14203064481
posicao 1190: 35376422615
posicao 1191: 8475564682
posicao 1192: 28645344440
posicao 1193: 13760487106
posicao 1194: Vazio
posicao 1195: 51604886501
posicao 1196: 55607477690
posicao 1197: 65271051420
posicao 1198: Vazio
posicao 1199: 76687972545
```

```
posicao 1200: Vazio
posicao 1201: 29069681021
posicao 1202: Vazio
posicao 1203: 45557943382
posicao 1204: 48116949101
posicao 1205: 82230874993
posicao 1206: 43596884667
posicao 1207: 81316778509
posicao 1208: Vazio
posicao 1209: Vazio
posicao 1210: 36730815353
posicao 1211: 56666297512
posicao 1212: 66286855513
posicao 1213: 70189432802
posicao 1214: 98357966969
posicao 1215: 96513586453
posicao 1216: Vazio
posicao 1217: Vazio
posicao 1218: 2655476603
posicao 1219: 74879803910
posicao 1220: 37446273909
posicao 1221: 315512547
posicao 1222: 68784410132
posicao 1223: Vazio
posicao 1224: 12506716122
posicao 1225: 63816838995
posicao 1226: 93488043401
posicao 1227: 97839630073
posicao 1228: 12357079037
```

# DISPERSÃO DUPLA



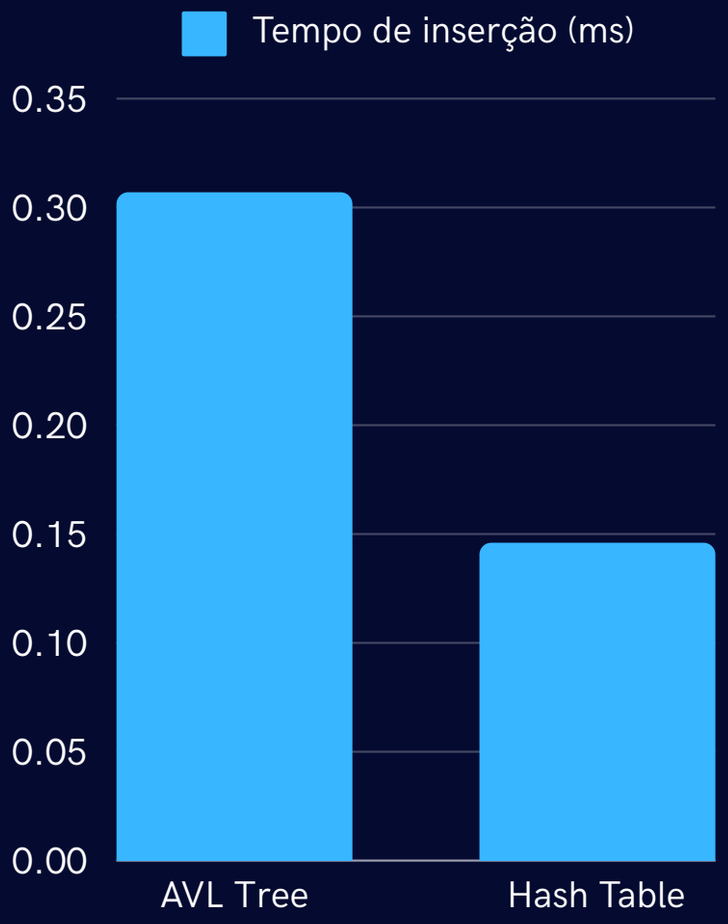
# É MELHOR QUE O (LOG N) ?

- Para responder essa pergunta, comparamos duas estruturas de dados: nossa tabela hash com dispersão dupla e uma árvore AVL.

- O objetivo é compreender se/quando a nossa função hash oferece desempenho superior em relação a uma árvore AVL em termos de complexidade para inserções e buscas, sabendo que uma árvore AVL tem complexidade O(log(n)) para as mesmas operações.

# AVL X TABELA HASH

$$dif\_performance = (0.307 - 0.146)/0.307 \times 100$$
$$dif\_performance = 52.44\%$$




*Valores obtidos através da função clock() implementada e compilada em https://www.onlinegdb.com (para que uma mostra de dados pequena pudesse ter diferença de performance capturada na função). Essa implementação foi removida da versão final do código porque a versão final da função de inserir cpfs na hash também conta com a abertura/gravação de um txt, o que distorceria sua performance de tempo.

# AVL X TABELA HASH

- Qual a média de acessos da função hash?

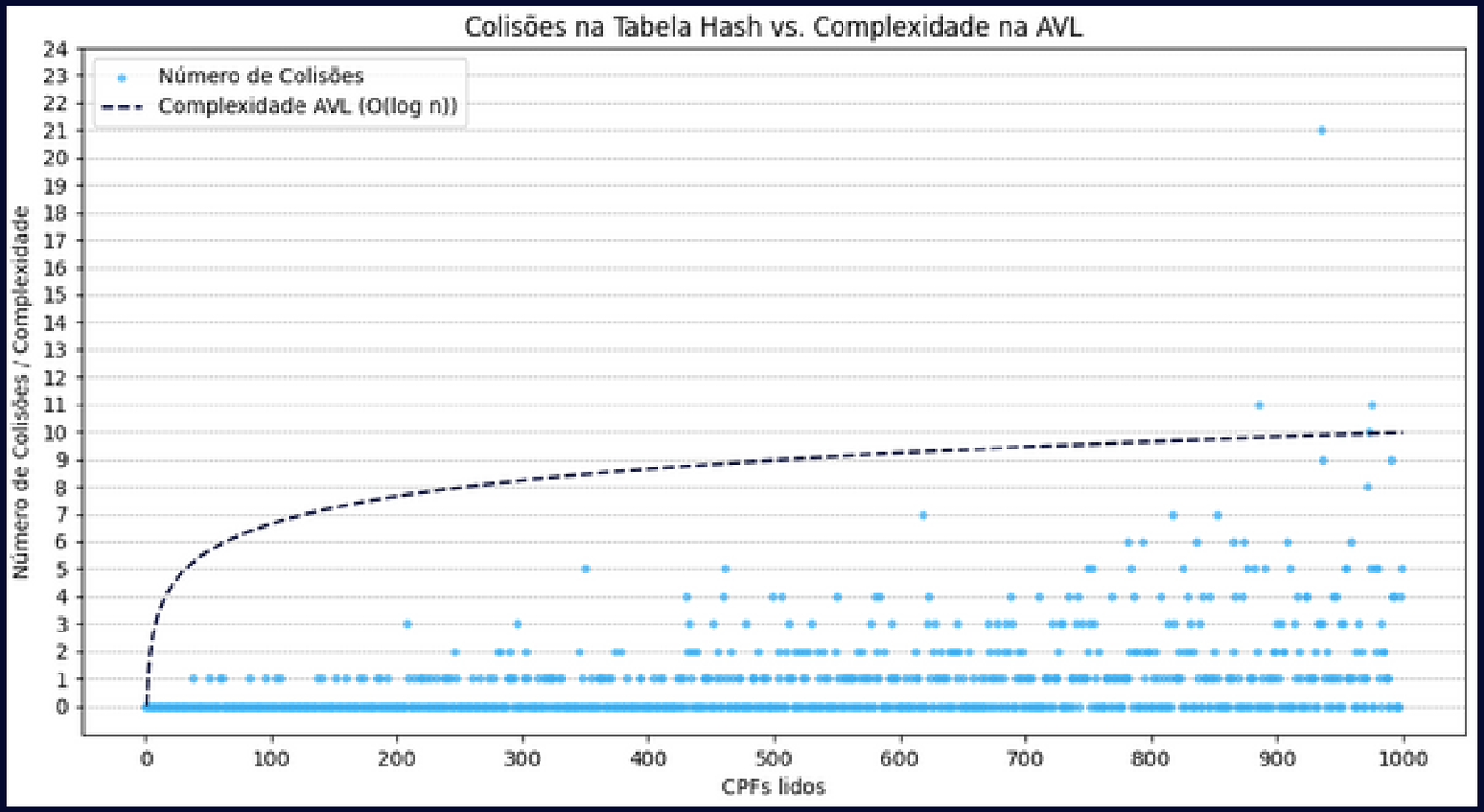
1 + (num_colisoes/num_cpfs) = entre 1 e 2 acessos.

- E a árvore AVL?

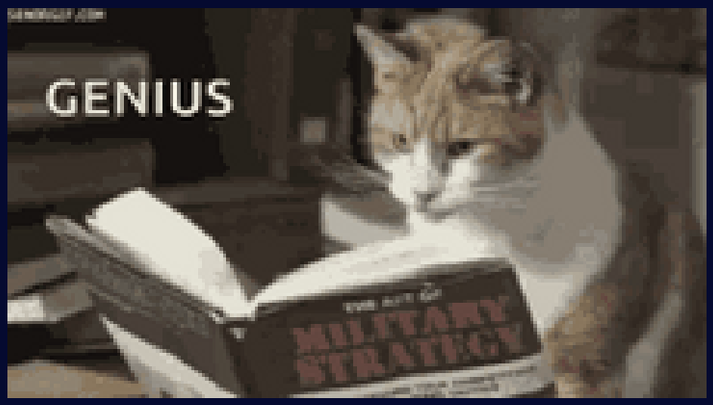
Constantemente O(log n), devido ao seu fator de balanceamento.
O(log 1000) = entre 9 e 10 acessos.

Logo, em média, a função hash apresenta um comportamento superior à árvore AVL. No entanto, vamos investigar a distribuição de colisões por CPF lido e esse comportamento estável da AVL:

# AVL X TABELA HASH



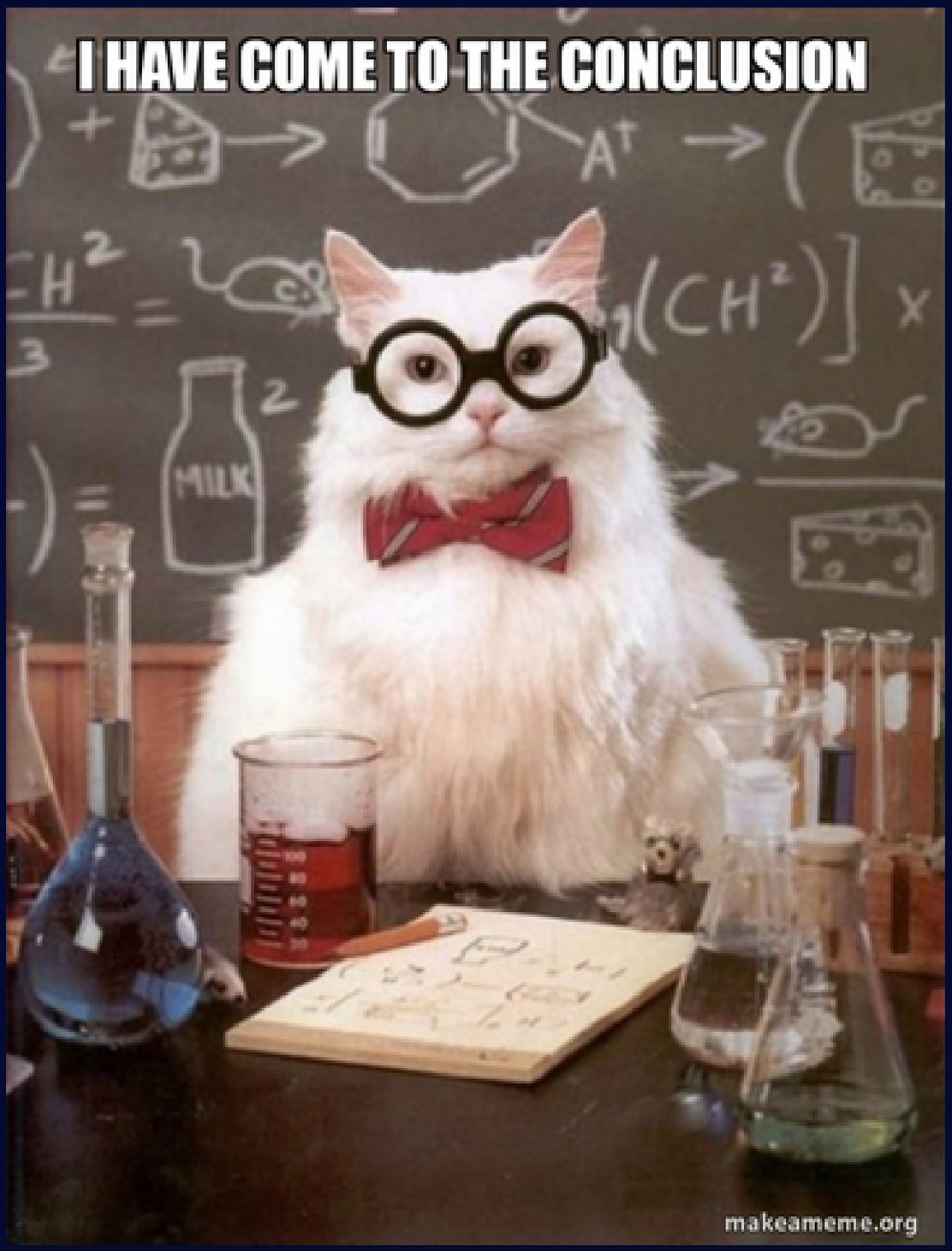

OBS.: Gráfico gerado com o script anexado como avl_vs_tabelaHash.py”



# AVL X TABELA HASH

A função hash apresenta bom desempenho na maioria dos índices de CPF, com poucas colisões, mantendo-se abaixo da linha de complexidade (O(log n)) da AVL em geral.

No entanto, número de colisões varia muito de um índice para outro. Essa dispersão das colisões afeta o tempo médio de busca, tornando a estrutura menos previsível e menos eficiente em alguns casos.


I HAVE COME TO THE CONCLUSION
MILK
makeameme.org

# AVL X TABELA HASH

- Desempenho: A tabela hash com dispersão dupla apresentou desempenho superior à árvore AVL.

- Impacto de Colisões: O aumento do número de colisões foi observado conforme o número de CPFs cresceu. O uso de dispersão dupla com hashing auxiliar permitiu que as colisões fossem resolvidas de forma mais eficaz, visando conservar a performance.

- Recomendação para Dados Maiores: A tabela hash é vantajosa para grandes volumes de dados não ordenados, mas o aumento contínuo de colisões pode exigir aumento no seu tamanho ou modificações na função hash.

T1 - INF1010
JÚLIA TADEU - 2312392
THEO CANUTO - 2311293