CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS III - UNIDADE LEOPOLDINA - Engenharia da Computação

Daniel Mescolin, Isabelly Araújo,
João Conte e Marcelo Filho.

Trabalho Prático - 2025.01
Análise e Projeto de Algoritmo

1 Introdução

O presente trabalho tem como objetivo realizar uma análise comparativa sobre a eficiência de algoritmos de ordenação e de técnicas para tratamento de colisões em tabelas hash. Para isso, analisamos o desempenho do algoritmo MergeSort (em sua forma pura e em uma variação otimizada) e das abordagens de Lista Encadeada e Sondagem Linear, utilizando como base dados reais do sistema de recomendação de filmes MovieLens. Este relatório apresentará os resultados obtidos para cada cenário, mensurados por meio de métricas como tempo de execução, uso de memória, número de comparações e quantidade de cópias, visando determinar a abordagem mais eficaz para cada contexto.

2 Cenário I: Impacto de diferentes estruturas de dados

2.1 Merge Sort em sua forma pura

No primeiro cenário experimental, avaliou-se a eficiência do algoritmo de ordenação Merge Sort em sua implementação clássica. Para a realização dos testes, foi utilizado um arquivo no formato .csv contendo aproximadamente 26 milhões de registros de avaliações de filmes, provenientes do repositório MovieLens.

Para cada valor de N, foram gerados cinco conjuntos distintos de dados, utilizando diferentes sementes baseadas nos campos USERID, MOVIEID, RATING e TIMESTAMP como referência para o gerador de números aleatórios. O processo consistiu na geração de números aleatórios no intervalo de 1 até o número total de linhas do arquivo, de modo a selecionar aleatoriamente os registros correspondentes às posições sorteadas.

As amostras geradas contemplaram os seguintes tamanhos: N = 1000, 5000, 10000, 50000, 100000. Cada algoritmo foi executado cinco vezes para cada valor de N, e os resultados foram avaliados com base na média dos tempos de execução obtidos. Os dados obtidos nesta etapa estão organizados nas tabelas a seguir.

Medida	Comparações	Cópias	Tempo
1	18796	23893	0.000054
2	19031	24188	0.000043
3	19682	24648	0.000044
4	18772	23930	0.000042
5	19047	24178	0.000042
Média	19065.6	24538.6	0.0000522

Tabela 1 para N=1000

Medida	Comparações	Cópias	Tempo
1	121491	160100	0.000282
2	122030	160469	0.000294
3	122659	161314	0.000311
4	123850	162046	0.000274
5	124656	163087	0.000270
Média	122937.2	161403.2	0.0002858

Tabela 2 para N = 5000

Medida	Comparações	Cópias	Tempo
1	251092	341198	0.000568
2	252714	343869	0.000565
3	252284	342283	0.000567
4	252147	342818	0.000562
5	250837	340262	0.000562
Média	251814.8	342086	0.0005648

Tabela 3 para N = 10000

Medida	Comparações	Cópias	Tempo
1	1573294	2156426	0.003650
2	1573385	2159397	0.003640
3	1561112	2148298	0.003593
4	1571324	2153130	0.003807
5	1574294	2156235	0.003775
Média	19516.2	24578.6	0.0000522

Tabela 4 para N = 50000

Medida	Comparações	Cópias	Tempo
1	3219436	4521859	0.008333
2	3208273	4517349	0.008016
3	3199945	4512577	0.008011
4	3200121	4518643	0.007788
5	3202750	4513973	0.007893
Média	3206105	4516880.2	0.0080082

Tabela 5 para N = 100000

As imagens de gráficos a seguir foram incluídas com o objetivo de visualizar o desempenho do algoritmo MergeSort em diferentes tamanhos de entrada. O gráfico representa a média de comparações, cópias e do tempo de execução ao processar vetores com quantidades variadas de elementos, simulando o comportamento do algoritmo em cenários práticos.

Para melhorar a visualização dos números, especialmente considerando que os tamanhos dos vetores variam significativamente, foi utilizada uma escala logarítmica no segundo gráfico, mesmo que utilizando os mesmos valores que o primeiro gráfico. Essa escolha permite que tanto valores pequenos quanto grandes sejam visualizados de forma equilibrada, evitando distorções e proporcionando uma compreensão mais precisa do comportamento assintótico do algoritmo.

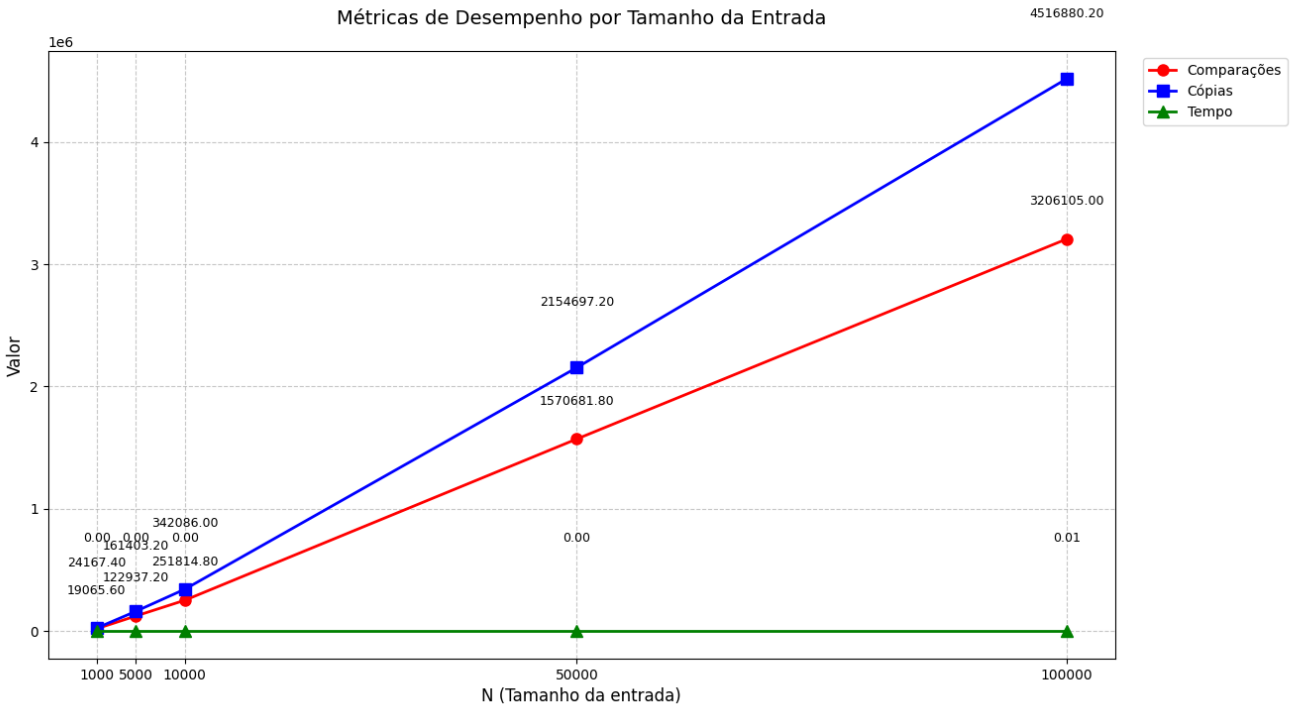

Figura 1: Gráfico das médias de desempenho

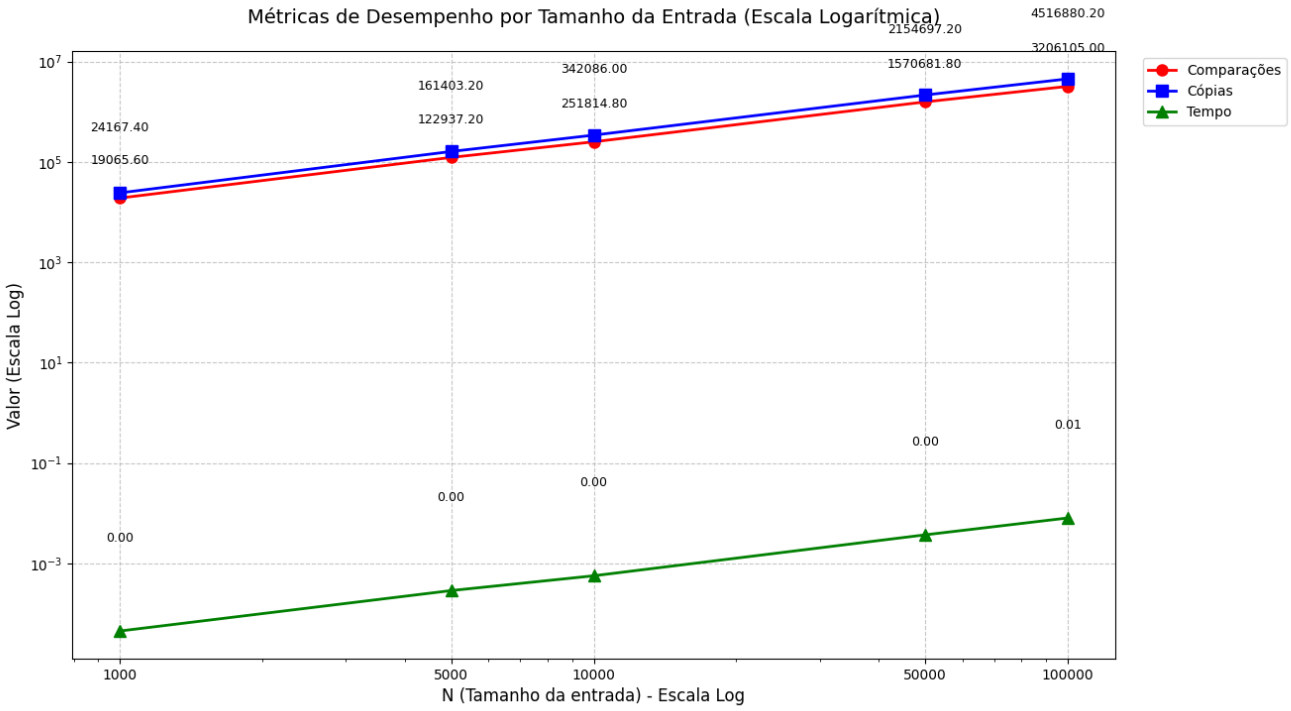

Figura 2: Gráfico das médias de desempenho em escala logaritmica

Como conclusão do cenário um temos a análise de que todas as métricas no final deram um resultado demasiadamente aproximado de O(nlogn) na imagem do gráfico, e quando é representado de forma logarítmica temos a aproximação definida como uma escala linear, como é esperado para esse tipo de análise dos valores. Também é importante ressaltar que a quantidade de dados não interfere diretamente no desempenho do algoritmo, já citado acima como O(nlogn) e que essa complexidade se mantem de forma apartada com a quantidade de dados pré-definidos. Importante ressaltar que o gráfico não teve uma boa visualização dos dados para a métrica de tempo, mas que a mesma, ao visualizar de forma logarítmica, foi vista como um comportamento deveras igual aos demais. Concluímos que a execução do algoritmo seguiu os termos esperados e que tivemos um desempenho dentro das métricas do mesmo.

3 Cenário 2: Impacto de modificação do MergeSort

O algoritmo MergeSort baseia-se no paradigma dividir-e-conquistar, dividindo recursivamente o vetor em duas metades até obter subvetores unitários, seguido pela fase de conquista através da intercalação ordenada desses subvetores. Sua complexidade temporal permanece constante em O(n log n) independentemente da distribuição inicial dos dados, característica que o torna particularmente adequado para cenários onde a previsibilidade de desempenho é crucial.

Já o insertion Sort apresenta uma abordagem incremental, inserindo cada elemento em sua posição correta dentro da porção já ordenada do vetor. Embora possua complexidade $O(n^2)$ no pior caso, demonstra excelente performance para conjuntos pequenos de dados, com complexidade O(n) no melhor caso quando os elementos já se encontram parcialmente ordenados.

O segundo cenário é composto pela variação híbrida, combinando as vantagens de ambos os algoritmos, utilizando o MergeSort para a decomposição inicial do problema e alternando para o Insertion Sort quando o tamanho do subvetor atinge o limiar de 100 elementos. Esta abordagem visa explorar a eficiência do Insertion Sort em conjuntos pequenos, potencialmente reduzindo o overhead de chamadas recursivas desnecessárias do MergeSort. A seguir, encontram-se as tabelas referentes aos dados coletados da execução do cenário 2 :

Medida	Comparações	Cópias	Tempo
1	5298	19952	0.000111
2	5260	19952	0.000084
3	5220	19952	0.000090
4	5209	19952	0.000088
5	5192	19952	0.000085
Média	5235.8	19952	0.0000916

Tabela 6 para N=1000

Medida	Comparações	Cópias	Tempo
1	33082	123616	0.000500
2	33025	123616	0.000583
3	33091	123616	0.000497
4	32889	123616	0.000459
5	32953	123616	0.000424
Média	33008	123616	0.0004926

Tabela 7 para N=5000

Medida	Comparações	Cópias	Tempo
1	71067	267232	0.001010
2	70907	267232	0.000893
3	71030	267232	0.000894
4	71079	267232	0.001018
5	71025	267232	0.001046
Média	71021.6	267232	0.0009722

Tabela 8 para N=10000

Medida	Comparações	Cópias	Tempo
1	411214	1568928	0.005093
2	411677	1568928	0.005158
3	411172	1568928	0.004966
4	411825	1568928	0.005090
5	411869	1568928	0.005092
Média	411551.4	1568928	0.0050798

Tabela 9 para N=50000

Medida	Comparações	Cópias	Tempo
1	873079	3337856	0.010920
2	873159	3337856	0.010859
3	873329	3337856	0.010610
4	872549	3337856	0.010659
5	874006	3337856	0.010661
Média	873224.4	3337856	0.0107418

Tabela 10 para N=100000

O Insertion Sort, embora possua complexidade quadrática no pior caso, se destaca pela sua simplicidade e excelente desempenho em vetores pequenos ou parcialmente ordenados. Já a versão híbrida busca otimizar o desempenho ao alternar dinamicamente entre os algoritmos: utiliza-se o MergeSort para a divisão inicial do vetor, e recorre-se ao Insertion Sort quando os subvetores atingem um tamanho inferior a 100 elementos.

Essas estratégias são analisadas nos gráficos a seguir, evidenciando como o comportamento de cada

abordagem varia conforme o tamanho e a ordenação dos dados de entrada. A seguir, são apresentados gráficos que ilustram o desempenho comparativo entre o algoritmo Insertion Sort e uma versão híbrida que combina MergeSort com Insertion Sort.

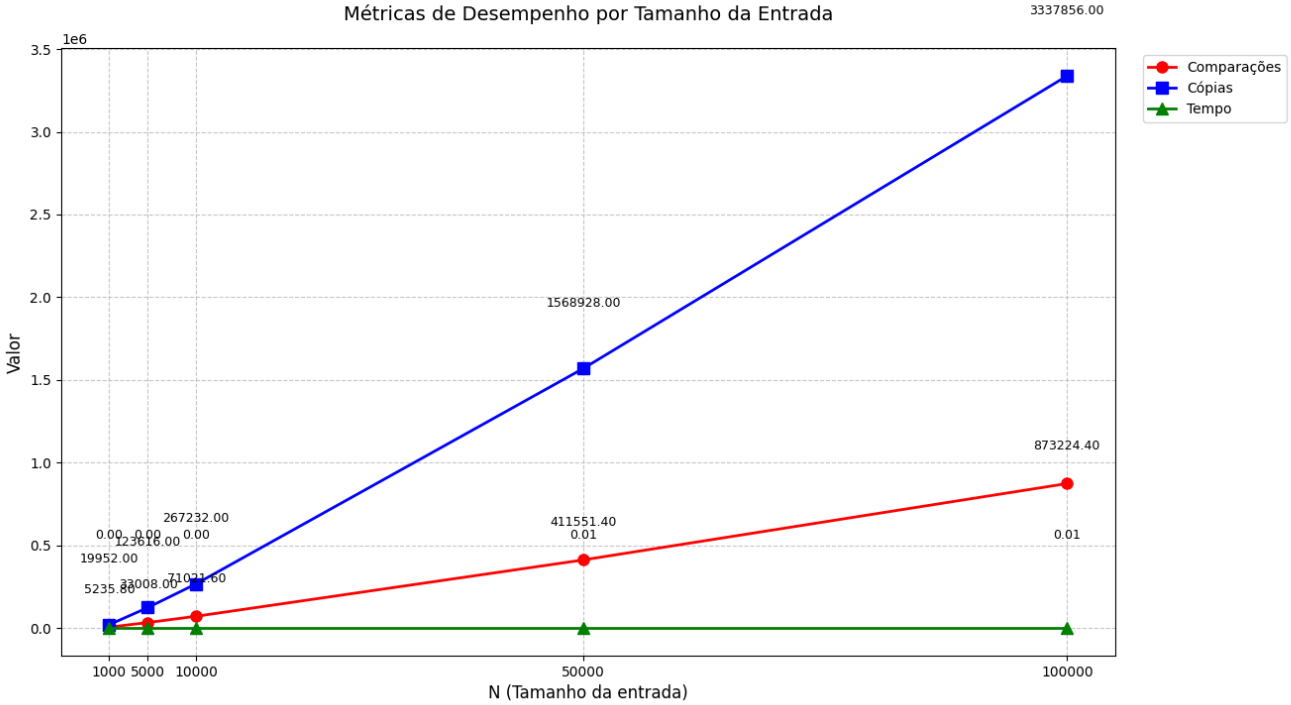

Figura 3: Gráfico das médias de desempenho

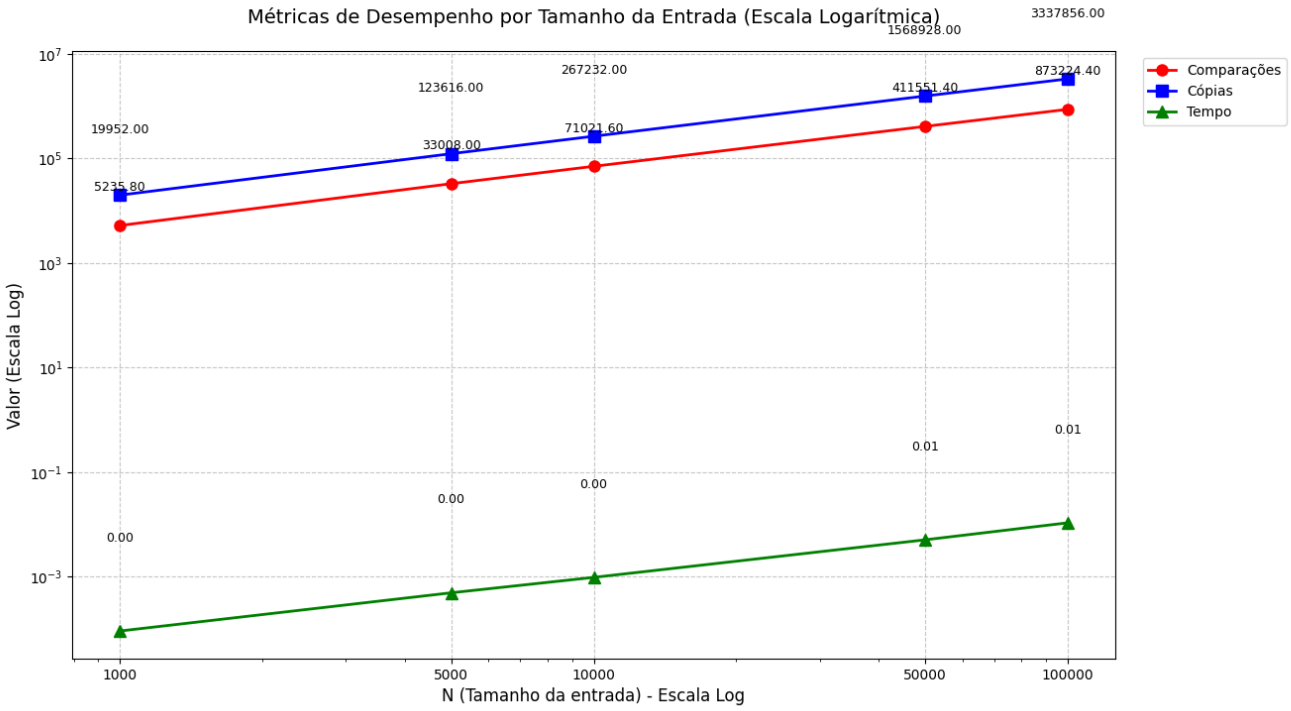

Figura 4: Gráfico das médias de desempenho em escala logaritmica

Como conclusão do cenário dois, observamos que a modificação no algoritmo MergeSort, com a introdução do Insertion Sort para subvetores de até 100 elementos, manteve os valores médios de comparações, cópias e tempo de execução dentro da complexidade esperada de O(n log n). Os gráficos gerados reforçam essa tendência, apresentando uma curva de crescimento compatível com essa complexidade, sobretudo quando analisados em escala logarítmica, onde o comportamento se aproxima de uma reta, como é típico de algoritmos com esse padrão de crescimento.

Um ponto importante a ser destacado é que, embora o Insertion Sort possua complexidade $O(n^2)$ no pior caso, essa característica não compromete a eficiência geral da abordagem híbrida, pois o algoritmo é aplicado apenas a subvetores pequenos, com até 100 elementos, independentemente do tamanho total da entrada. Isso ocorre porque o MergeSort divide o vetor recursivamente até atingir esse limiar, momento em que o Insertion Sort é utilizado para ordenar localmente. Nesse contexto, o custo agregado dessa operação é proporcional a $O(n^2)$, o que não altera a complexidade assintótica final do algoritmo,

que permanece O(n log n). Na prática, essa substituição contribui para reduzir o overhead gerado pelas chamadas recursivas em subestruturas pequenas, resultando em melhor desempenho.

Apesar de certa dificuldade de visualização da métrica de tempo nos gráficos em escala linear, ao utilizar a escala logarítmica foi possível perceber que seu comportamento acompanha de forma estável as demais métricas, confirmando a coerência dos dados obtidos. Assim, concluímos que a execução da versão híbrida seguiu os padrões esperados. A combinação estratégica entre MergeSort e Insertion Sort preservou a eficiência teórica do algoritmo e demonstrou bons resultados práticos, principalmente em termos de estabilidade e desempenho em diferentes tamanhos de entrada.

4 Cenário 3: Tratamento de Colisões

Neste cenário, a análise foca no desempenho de algoritmos para tratamento de colisão em tabelas hash, comparando duas abordagens clássicas: a Lista Encadeada (ou Encadeamento Separado) e a Sondagem Linear (ou Endereçamento Aberto).

4.1 Lista Encadeada

A Lista Encadeada resolve colisões tratando cada posição da tabela hash como um ponteiro para o início de uma lista. Todos os elementos cujas chaves mapeiam para o mesmo índice são armazenados nessa lista. A eficiência desta abordagem está diretamente ligada ao fator de carga (alpha) da tabela. A busca por um elemento exige percorrer a lista correspondente, resultando em um número de comparações que cresce linearmente com o número de elementos naquele índice, mas que se mantém eficiente mesmo com a tabela cheia. Seguem as tabelas relacionados as execuções seguinte o tratamento de colisão da lista encadeada.

Execução	Comparações	Memória gasta
1	487	32024
2	511	32024
3	482	32024
Média	493.34	32024

Tabela 11 para N=1000

Execução	Comparações	Memória gasta
1	2462	160024
2	2430	160024
3	2586	160024
Média	2492.666666	160024

Tabela 12 para N=5000

Execução	Comparações	Memória gasta
1	4972	320024
2	4981	320024
3	5022	319904
Média	4991.66666666	319984

Tabela 13 para N=10000

Execução	Comparações	Memória gasta
1	24795	1599160
2	24908	1598680
3	24946	1598536
Média	24.883	1.598.792

Tabela 14 para N=50000

Execução	Comparações	Memória gasta
1	49784	3195512
2	49979	3195248
3	49762	3195056
Média	49841.66666	3195272

Tabela 15 para N=100000

Execução	Comparações	Memória gasta
1	251957	15887272
2	250706	15886816
3	250673	15885904
Média	251112	15886664

Tabela 16 para N=500000

Execução	Comparações	Memória gasta
1	506816	31550168
2	505270	31551536
3	504614	31550768
Média	505566.6666	31550824

Tabela 17 para N=1000000

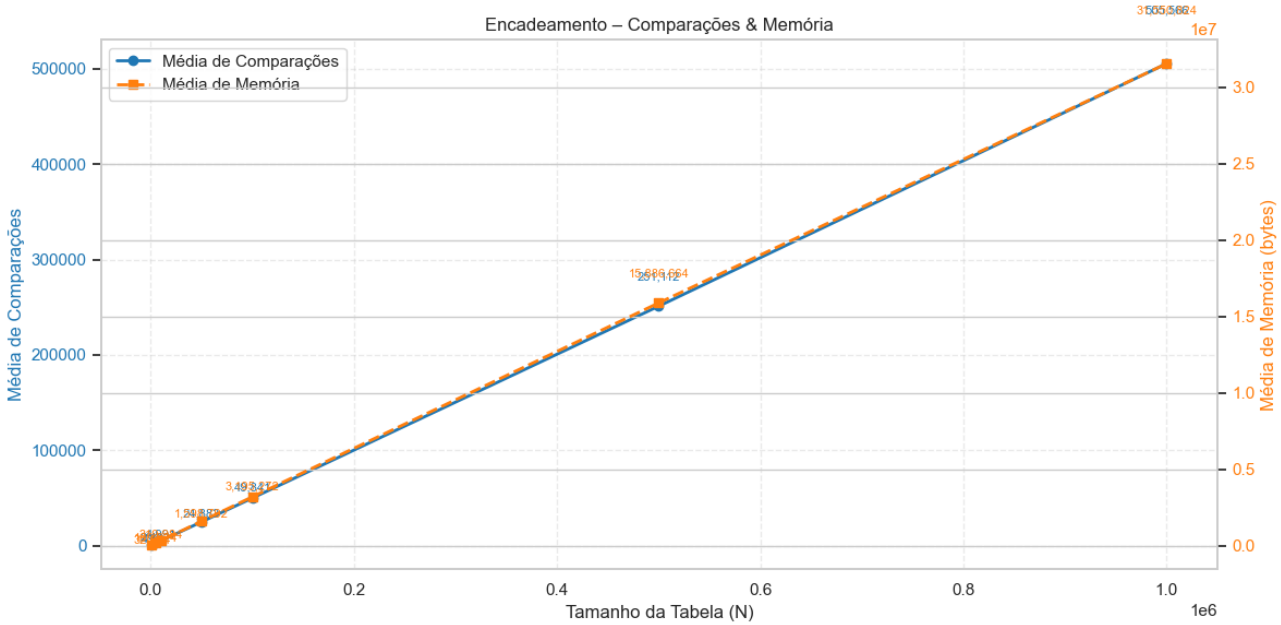

Figura 5: Gráfico de comparações e memória utilizada

4.2 Sondagem Linear

Já a Sondagem Linear adota uma estratégia de endereçamento aberto. Em caso de colisão no índice i, o algoritmo "sonda"as posições subsequentes (i+1, i+2, ...) de forma sequencial até encontrar um espaço vazio. Embora seja muito eficiente em termos de uso de memória cache para fatores de carga baixos, sua performance se degrada rapidamente à medida que a tabela se aproxima de sua capacidade máxima, devido ao fenômeno de agrupamento primário (primary clustering), que aumenta drasticamente o número médio de comparações necessárias para encontrar um espaço livre. A seguir, encontram-se as tabelas de execução do tratamento com sondagem linear para o cenário 3 :

Execução	Comparações	Memória gasta
1	18994	16024
2	29459	16024
3	22453	16024
Média	23635	16024

Tabela 18 para N=1000

Execução	Comparações	Memória gasta
1	229587	80024
2	365780	80024
3	151796	80024
Média	249054	80024

Tabela 19 para N=5000

Execução	Comparações	Memória gasta
1	475330	160024
2	400537	160024
3	429758	160024
Média	435208	160024

Tabela 20 para N=10000

Execução	Comparações	Memória gasta
1	6821407	800024
2	7935344	800024
3	6393557	800024
Média	7050103	800024

Tabela 21 para N=50000

Execução	Comparações	Memória gasta
1	19283815	1600024
2	18250911	1600024
3	26961480	1600024
Média	21498735	1600024

Tabela 22 para N=100000

Execução	Comparações	Memória gasta
1	141113558	8000024
2	104470093	8000024
3	121337509	8000024
Média	122307053	8000024

Tabela 23 para N=500000

Execução	Comparações	Memória gasta
1	269507725	16000024
2	223388863	16000024
3	242549842	16000024
Média	245148810	16000024

Tabela 24 para N=1000000

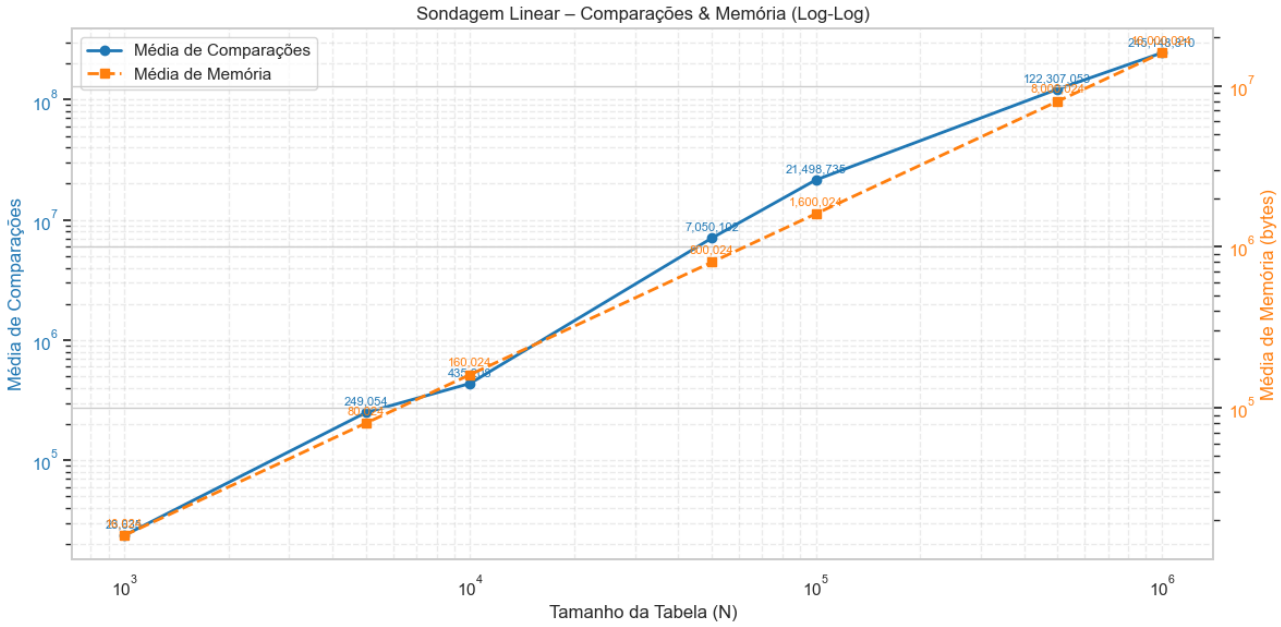

Figura 6: Gráfico de comparações e memória utilizada

No tratamento de colisões em tabelas hash, observamos diferenças significativas entre as abordagens de lista encadeada e sondagem linear, tanto em termos de desempenho de busca, quanto de uso de memória.

Em lista encadeada, a abordagem demonstrou comportamento linear estável no número de comparações à medida que o número de elementos aumentava. Mesmo com o número de entradas progredindo até 1.000.000 de elementos, o número de comparações se manteve proporcional ao crescimento de N, com escalabilidade controlada e uso de memória crescente, como esperado, já que com o crescimento de entradas, também tem o crescimento de ponteiros. Isso é corente com a complexidade esperada de busca média O(1+α), sendo $\alpha = n/m$, o fator de carga. O custo de memória cresce linearmente, mas em termos de desempenho, a abordagem mostrou consistência e robustez, mesmo sob alta carga.

Em contrapartida, a sondagem linear apresentou forte queda de desempenho à medida que a tabela se tornava mais carregada. Como evidenciado nos testes, para N = 1.000.000, o número médio de comparações ultrapassou 250 milhões, o que representa uma ordem de grandeza muito maior que no método de lista encadeada. Isso ocorre devido ao fenômeno de agrupamento primário, que aumenta a dificuldade de inserções e buscas em tabelas densas. A complexidade média, que é próxima de O(1) para baixas cargas, se aproxima de O(n) em cenários mais saturados. No entanto, a memória utilizada permaneceu quase constante, já que não há alocação adicional além da própria tabela. Portanto, a lista encadeada se mostrou mais eficiente em tempo quando a tabela está com alta carga, mesmo com maior consumo de memoria (devido a alocação de ponteiros), já a sondagem linear , apesar do uso eficiente de memória, tornou-se inviável em grandes entradas, devido ao alto custo em comparações.

5 Apêndice

5.1 Arquivo data-handler.c

```c
#include "data_handler.h"
#include <stdlib.h>
#include <string.h>

// Define o tamanho maximo da linha para leitura do arquivo
#define MAX_LINE_LENGTH 1024

// Funcao para importar dados de um arquivo CSV
// Retorna o numero de ratings importados ou -1 em caso de erro
int importData(MoveRating **ratings, int *capacity, const char *filename)
{
    // Abre o arquivo CSV para leitura e verifica se foi aberto
       corretamente
    FILE *fp = fopen(filename, "r");
    if (fp == NULL)
    {
        perror("Error opening file");
```

```
17          return -1;
18      }
19
20      char line[MAX_LINE_LENGTH];
21      int count = 0;
22
23      // Le a primeira linha do arquivo (cabecalho) e ignora
24      // Se a primeira linha nao puder ser lida, retorna 0 ratings
            importados
25      if (fgets(line, sizeof(line), fp) == NULL)
26      {
27          fclose(fp);
28          return 0;
29      }
30
31      // Le o arquivo linha por linha
32      while (fgets(line, sizeof(line), fp) != NULL)
33      {
34          if (count == *capacity)
35          {
36              // Se o numero de ratings atingir a capacidade, dobra a
                    capacidade
37              *capacity *= 2;
38              MoveRating *temp = realloc(*ratings, *capacity * sizeof(
                    MoveRating));
39              // Se a realocacao falhar, libera a memoria e retorna o numero
                     de ratings importados
40              if (temp == NULL)
41              {
42                  fprintf(stderr, "Memory reallocation failed\n");
43                  fclose(fp);
44                  return count;
45              }
46              // Atualiza o ponteiro para os ratings
47              *ratings = temp;
48          }
49
50          // Le os dados da linha e armazena no array de ratings
51          // Se a leitura falhar, imprime um aviso e continua para a proxima
                 linha
52          // Espera que a linha tenha o formato: userId,moveId,rating,
                timestamp
53          if (sscanf(line, "%d,%d,%f,%d",
54                     &(*ratings)[count].userId,
55                     &(*ratings)[count].moveId,
56                     &(*ratings)[count].rating,
57                     &(*ratings)[count].timestamp) == 4)
58          {
```

```
            count++;
        }
        else
        {
            fprintf(stderr, "Warning: Error parsing line, skipping: %s",
                line);
        }
    }

    fclose(fp);
    return count;
}

// Funcao para exibir os dados dos ratings (nao utilizada no codigo
    principal, mas pode ser util para depuracao)
void showData(const MoveRating *ratings, int count)
{
    for (int i = 0; i < count; i++)
    {
        printf("User ID: %d, Move ID: %d, Rating: %.1f, Timestamp: %d\n",
               ratings[i].userId,
               ratings[i].moveId,
               ratings[i].rating,
               ratings[i].timestamp);
    }
}
```

5.2 Arquivo data-handler.h

```
#ifndef DATA_HANDLER_H
#define DATA_HANDLER_H

#include <stdio.h>

// Estruturas de dados para armazenar os ratings
typedef struct MoveRating
{
    int userId;
    int moveId;
    float rating;
    int timestamp;
} MoveRating;

// Declara as das funcoes para manipulacao de dados
int importData(MoveRating **ratings, int *capacity, const char *filename);
```

```
17 void showData(const MoveRating *ratings, int count);
18
19 #endif
```

5.3 Arquivo hash-table.c

```
1
2  #include "hash_table.h"
3  #include <stdlib.h>
4  #include <string.h>
5
6  // Funcao de hash para calcular o indice baseado no userId e moveId
7  // Usa o algoritmo djb2 para gerar um hash unico
8  // Recebe o userId, moveId e o tamanho da tabela
9  // Retorna o indice calculado
10 static int hash(int userId, int moveId, int size)
11 {
12     unsigned long h = 5381;
13     h = ((h << 5) + h) + userId;
14     h = ((h << 5) + h) + moveId;
15     return h % size;
16 }
17
18 // Funcao para criar uma tabela de hash com encadeamento
19 // Recebe o tamanho da tabela
20 // Aloca memoria para a tabela e inicializa os valores
21 // Retorna um ponteiro para a tabela de hash criada
22 HashTableChaining *createHashTableChaining(int size)
23 {
24     // Aloca memoria para a tabela de hash
25     HashTableChaining *ht = malloc(sizeof(HashTableChaining));
26     if (!ht)
27         return NULL;
28     ht->table = calloc(size, sizeof(Node *));
29     // Verifica se a alocacao foi bem-sucedida
30     if (!ht->table)
31     {
32         free(ht);
33         return NULL;
34     }
35     // Inicializa o tamanho e as comparacoes
36     ht->size = size;
37     ht->comparisons = 0;
38     return ht;
39 }
```

```
40
41 // Funcao para inserir um MoveRating na tabela de hash com encadeamento
42 // Recebe a tabela de hash e o MoveRating a ser inserido
43 // Calcula o indice usando a funcao hash
44 // Verifica se ja existe um no com o mesmo userId e moveId
45 // Se existir, nao insere o novo no
46 // Se nao existir, cria um novo no, inicializa com os dados e insere no
47 // inicio da lista encadeada
48 void insertChaining(HashTableChaining *ht, MoveRating data)
49 {
50     // Calcula o indice usando a funcao hash
51     int index = hash(data.userId, data.moveId, ht->size);
52     // Verifica se ja existe um no com o mesmo userId e moveId
53     Node *current = ht->table[index];
54     // Itera pela lista encadeada para verificar se o no ja existe
55     // Se encontrar um no com os mesmos userId e moveId, nao insere
56     // Se nao encontrar, cria um novo no e o insere no inicio da lista
57     while (current != NULL)
58     {
59         ht->comparisons++;
60         if (current->data.userId == data.userId && current->data.moveId ==
               data.moveId)
61         {
62             return;
63         }
64         current = current->next;
65     }
66
67     // Cria um novo no, inicializa com os dados e insere no inicio da
           lista
68     Node *newNode = malloc(sizeof(Node));
69     if (!newNode)
70         return;
71
72     // Verifica se a alocacao foi bem-sucedida
73     newNode->data = data;
74     newNode->next = ht->table[index];
75     ht->table[index] = newNode;
76 }
77
78 // Funcao para liberar a memoria da tabela de hash com encadeamento
79 // Itera por cada indice da tabela, libera os nos encadeados e a tabela
80 // Recebe a tabela de hash
81 // Libera a memoria alocada para a tabela e os nos encadeados
82 void freeHashTableChaining(HashTableChaining *ht)
83 {
84     if (!ht)
85         return;
```

```
86      for (int i = 0; i < ht->size; i++)
87      {
88          Node *current = ht->table[i];
89          while (current != NULL)
90          {
91              Node *temp = current;
92              current = current->next;
93              free(temp);
94          }
95      }
96      free(ht->table);
97      free(ht);
98  }
99
100 // Funcao para obter o numero de comparacoes realizadas na tabela de hash
        com encadeamento
101 long long getComparisonsChaining(HashTableChaining *ht)
102 {
103     return ht->comparisons;
104 }
105
106 // Funcao para obter o uso de memoria da tabela de hash com encadeamento
107 long getMemoryUsageChaining(HashTableChaining *ht)
108 {
109     // Calcula o uso de memoria total
110     long memory = sizeof(HashTableChaining) + ht->size * sizeof(Node *);
111     // Itera por cada indice da tabela e soma o tamanho dos nos encadeados
112     for (int i = 0; i < ht->size; i++)
113     {
114         Node *current = ht->table[i];
115         // Soma o tamanho de cada no encadeado
116         while (current != NULL)
117         {
118             memory += sizeof(Node);
119             current = current->next;
120         }
121     }
122     return memory;
123 }
124
125 // Funcao para criar uma tabela de hash linear
126 // Recebe o tamanho da tabela
127 // Aloca memoria para a tabela e inicializa os valores
128 // Retorna um ponteiro para a tabela de hash criada
129 HashTableLinear *createHashTableLinear(int size)
130 {
131     // Aloca memoria para a tabela de hash linear
132     HashTableLinear *ht = malloc(sizeof(HashTableLinear));
```

```
133     if (!ht)
134         return NULL;
135     ht->table = malloc(size * sizeof(MoveRating));
136     // Verifica se a alocacao foi bem-sucedida
137     if (!ht->table)
138     {
139         free(ht);
140         return NULL;
141     }
142     // Inicializa o tamanho e as comparacoes
143     ht->size = size;
144     ht->comparisons = 0;
145     for (int i = 0; i < size; i++)
146     {
147         ht->table[i].userId = -1;
148     }
149     return ht;
150 }
151 
152 // Funcao para inserir um MoveRating na tabela de hash linear
153 // Recebe a tabela de hash e o MoveRating a ser inserido
154 // Calcula o indice usando a funcao hash
155 // Verifica se ja existe um elemento com o mesmo userId e moveId
156 // Se existir, nao insere o novo elemento
157 // Se nao existir, insere o elemento na posicao livre
158 void insertLinearProbing(HashTableLinear *ht, MoveRating data)
159 {
160     // Calcula o indice usando a funcao hash
161     int index = hash(data.userId, data.moveId, ht->size);
162     // Verifica se ja existe um elemento com o mesmo userId e moveId
163     // Se existir, nao insere o novo elemento
164     // Se nao existir, insere o elemento na posicao livre
165     // Itera pela tabela ate encontrar uma posicao livre ou o elemento ja
        existente
166     // Incrementa o contador de comparacoes a cada iteracao
167     // Se encontrar uma posicao livre, insere o elemento
168     // Se encontrar o elemento ja existente, nao insere
169     while (ht->table[index].userId != -1)
170     {
171         ht->comparisons++;
172         if (ht->table[index].userId == data.userId && ht->table[index].
            moveId == data.moveId)
173         {
174             return;
175         }
176         index = (index + 1) % ht->size;
177     }
178     // Insere o elemento na posicao livre
```

```
179     ht->table[index] = data;
180 }
181
182 // Funcao para liberar a memoria da tabela de hash linear
183 void freeHashTableLinear(HashTableLinear *ht)
184 {
185     if (!ht)
186         return;
187     free(ht->table);
188     free(ht);
189 }
190
191 // Funcao para obter o numero de comparacoes realizadas na tabela de hash
    linear
192 long long getComparisonsLinear(HashTableLinear *ht)
193 {
194     return ht->comparisons;
195 }
196
197 // Funcao para obter o uso de memoria da tabela de hash linear
198 long getMemoryUsageLinear(HashTableLinear *ht)
199 {
200     return sizeof(HashTableLinear) + ht->size * sizeof(MoveRating);
201 }
```

5.4 Arquivo hash-table.h

```
1  #ifndef HASH_TABLE_H
2  #define HASH_TABLE_H
3
4  #include "data_handler.h"
5  #include <stdio.h>
6
7  typedef struct Node
8  {
9      MoveRating data;
10     struct Node *next;
11 } Node;
12
13 typedef struct HashTableChaining
14 {
15     Node **table;
16     int size;
17     long long comparisons;
18 } HashTableChaining;
```

```
19
20 typedef struct HashTableLinear
21 {
22     MoveRating *table;
23     int size;
24     long long comparisons;
25 } HashTableLinear;
26
27 HashTableChaining *createHashTableChaining(int size);
28 void insertChaining(HashTableChaining *ht, MoveRating data);
29 void freeHashTableChaining(HashTableChaining *ht);
30 long long getComparisonsChaining(HashTableChaining *ht);
31 long getMemoryUsageChaining(HashTableChaining *ht);
32
33 HashTableLinear *createHashTableLinear(int size);
34 void insertLinearProbing(HashTableLinear *ht, MoveRating data);
35 void freeHashTableLinear(HashTableLinear *ht);
36 long long getComparisonsLinear(HashTableLinear *ht);
37 long getMemoryUsageLinear(HashTableLinear *ht);
38
39 #endif
```

5.5 Arquivo main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "data_handler.h"
4 #include "scenarioI.h"
5 #include "scenarioII.h"
6 #include "scenarioIII.h"
7
8 #define INITIAL_CAPACITY 1000
9
10 int main(int argc, char *argv[])
11 {
12     if (argc < 2)
13     {
14         fprintf(stderr, "Usage: %s <filename.csv>\n", argv[0]);
15         return 1;
16     }
17
18     const char *filename = argv[1];
19     int capacity = INITIAL_CAPACITY;
20     MoveRating *ratings = malloc(capacity * sizeof(MoveRating));
21
```

```
    if (ratings == NULL)
    {
        fprintf(stderr, "Initial memory allocation failed\n");
        return 1;
    }

    printf("Importing data from %s...\n", filename);
    int numRatings = importData(&ratings, &capacity, filename);

    if (numRatings > 0)
    {
        printf("Successfully imported %d ratings.\n", numRatings);

        runScenarioI(ratings, numRatings);
        runScenarioII(ratings, numRatings);
        runScenarioIII(ratings, numRatings);
    }
    else if (numRatings == 0)
    {
        printf("No data was imported from the file.\n");
    }
    else
    {
        fprintf(stderr, "An error occurred during file import.\n");
    }

    free(ratings);
    printf("Program finished.\n");

    return 0;
}
```

5.6 Arquivo scenariol.c

```
// Importa as bibliotecas necessarias
#include "scenarioI.h"
// Importa as funcoes auxiliares
#include "utils.h"
// Importa as funcoes de ordenacao
#include "sorting.h"
// Demais includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

```

```c
// Define os nomes dos arquivos de saida
#define OUTPUT_FILE "scenario1_results.txt"
#define OUTPUT_CSV_FILE "running_scenario1.csv"
// Define o tamanho maximo dos ratings que podem ser carregados na memoria
#define MAX_RATINGS_IN_MEMORY 26024288
// Define a quantidade maxima de execucoes para o cenario 1
#define MAX_QUANTITY_EXECUTIONS 5

// Funcao para executar o cenario 1
// Recebe os ratings e o numero de ratings
// Verifica se ha ratings para processar
// Se nao houver ratings, imprime uma mensagem e retorna
void runScenario1(const MoveRating *ratings, int numRatings)
{
    // Verifica se ha ratings para processar
    if (numRatings <= 0)
    {
        printf("No ratings data to process for Scenario 1.\n");
        return;
    }

    // Define os tamanhos de teste para o cenario 1
    int testSizes[] = {1000, 5000, 10000, 50000, 100000};
    int numTestSizes = sizeof(testSizes) / sizeof(testSizes[0]);

    srand(time(NULL));

    // Abre o arquivo de log para escrita, este arquivo armazenara os
       resultados das execucoes e e um .txt
    FILE *logFile = fopen(OUTPUT_FILE, "w");
    if (logFile == NULL)
    {
        perror("Error opening log or CSV file for writing");
        return;
    }
    // Cria um arquivo CSV para cada execucao, este arquivo armazenara os
       resultados das execucoes e e um .csv
    FILE *csvFile = fopen(OUTPUT_CSV_FILE, "w");
    if (csvFile == NULL)
    {
        perror("Error opening CSV file for writing");
        fclose(logFile);
        return;
    }

    // Itera sobre os tamanhos de teste definidos, para cada tamanho de
       teste
    for (int i = 0; i < numTestSizes; i++)
```

```
    {
        // Imprime no log o numero da execucao atual
        fprintf(logFile, "\n--- Execution %d of Scenario I ---\n", i + 1);
            // Executa o cenario I para a quantidade maxima de execucoes,
           faz as 5 execucoes definidas
        for (int count = 0; count < MAX_QUANTITY_EXECUTIONS; count++)
        {
            // Para cada tamanho de teste, verifica se ha dados
               suficientes
            // Caso nao haja dados suficientes, pula para o proximo
               tamanho
            int currentSize = testSizes[i];
            if (currentSize > numRatings)
            {
                fprintf(logFile, "\nSkipping size %d: not enough data
                   available (have %d).\n", currentSize, numRatings);
                continue;
            }

            // Imprime no log o tamanho atual do teste
            fprintf(logFile, "\n--- Scenario I: Sorting with %d ratings
               ---\n", currentSize);

            // Cria um subconjunto aleatorio dos ratings para o tamanho
               atual do teste
            MoveRating *subset = createRandomSubset(ratings, numRatings,
               currentSize);

            // Executa a ordenacao e registra as metricas
            runAndLogSort(subset, currentSize, "userId", logFile, csvFile,
                count, 2);

            // Libera a memoria alocada para o subconjunto
            free(subset);
        }
    }

    // Fecha o arquivo CSV apos todas as execucoes
    fclose(csvFile);
    // Fecha o arquivo de log apos todas as execucoes
    fclose(logFile);
    printf("\nScenario I completed successfully. Results saved to %s\n",
       OUTPUT_FILE);
}
```

5.7 Arquivo scenarioI.h

```

#ifndef SCENARIOI_H
#define SCENARIOI_H

// Importa o data_handler.h para usar a estrutura MoveRating
#include "data_handler.h"

// Define as funcoes para o cenario I
void runScenarioI(const MoveRating *ratings, int numRatings);

#endif
```

5.8 Arquivo scenarioII.c

```
// Importa as bibliotecas necessarias
#include "scenarioII.h"
// Importa as funcoes auxiliares
#include "utils.h"
// Importa as funcoes de ordenacao
#include "sorting.h"
// Demais includes
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define os nomes dos arquivos de saida
#define OUTPUT_FILE "scenarioII_results.txt"
#define OUTPUT_CSV_FILE "running_scenarioII.csv"
// Define o tamanho maximo dos ratings que podem ser carregados na memoria
#define MAX_RATINGS_IN_MEMORY 26024288
// Define a quantidade maxima de execucoes para o cenario II
#define MAX_QUANTITY_EXECUTIONS 5

// Funcao para executar o cenario II
// Recebe os ratings e o numero de ratings
// Verifica se ha ratings para processar
// Se nao houver ratings, imprime uma mensagem e retorna
void runScenarioII(const MoveRating *ratings, int numRatings)
{
    // Verifica se ha ratings para processar
    if (numRatings <= 0)
    {
        printf("No ratings data to process for Scenario II.\n");
```

```
30          return;
31      }
32
33      // Define os tamanhos de teste para o cenario II
34      int testSizes[] = {1000, 5000, 10000, 50000, 100000};
35      int numTestSizes = sizeof(testSizes) / sizeof(testSizes[0]);
36
37      srand(time(NULL));
38
39      // Abre o arquivo de log para escrita, este arquivo armazenara os
            resultados das execucoes e e um .txt
40      FILE *logFile = fopen(OUTPUT_FILE, "w");
41      if (logFile == NULL)
42      {
43          perror("Error opening log file for writing");
44          return;
45      }
46      // Cria um arquivo CSV para cada execucao, este arquivo armazenara os
            resultados das execucoes e e um .csv
47      FILE *csvFile = fopen(OUTPUT_CSV_FILE, "w");
48      if (csvFile == NULL)
49      {
50          perror("Error opening CSV file for writing");
51          fclose(logFile);
52          return;
53      }
54
55      // Itera sobre os tamanhos de teste definidos, para cada tamanho de
            teste
56      for (int i = 0; i < numTestSizes; i++)
57      {
58          // Imprime no log o numero da execucao atual
59          fprintf(logFile, "\n--- Execution %d of Scenario II ---\n", i + 1)
                ;
60          // Executa o cenario II para a quantidade maxima de execucoes, faz
                as 5 execucoes definidas
61          for (int count = 0; count < MAX_QUANTITY_EXECUTIONS; count++)
62          {
63              // Para cada tamanho de teste, verifica se ha dados
                    suficientes
64              // Caso nao haja dados suficientes, pula para o proximo
                    tamanho
65              int currentSize = testSizes[i];
66              if (currentSize > numRatings)
67              {
68                  fprintf(logFile, "\nSkipping size %d: not enough data
                        available (have %d).\n", currentSize, numRatings);
69                  continue;
```

```
            }

            // Imprime no log o tamanho atual do teste
            fprintf(logFile, "\n--- Scenario II: Sorting with %d ratings ---\n", currentSize);

            // Cria um subconjunto aleatorio dos ratings para o tamanho atual do teste
            MoveRating *subset = createRandomSubset(ratings, numRatings, currentSize);

            // Executa a ordenacao e registra as metricas
            runAndLogSort(subset, currentSize, "userId", logFile, csvFile, count, 1);

            // Libera a memoria alocada para o subconjunto
            free(subset);
        }
    }

    // Fecha o arquivo CSV apos todas as execucoes
    fclose(csvFile);
    // Fecha o arquivo de log apos todas as execucoes
    fclose(logFile);
    printf("\nScenario II completed successfully. Results saved to %s\n", OUTPUT_FILE);
}
```

5.9 Arquivo scenario II.h

```
#ifndef SCENARIOII_H
#define SCENARIOII_H

 //Importa o data_handler.h para usar a estrutura MoveRating
#include "data_handler.h"

// Define as funcoes para o cenario II
void runScenarioII(const MoveRating *ratings, int numRatings);

#endif
```

5.10 Arquivo scenario III.c

```c
// Importa as bibliotecas necessarias
#include "scenarioIII.h"
// Importa as funcoes de hashing
#include "hash_table.h"
// Importa as funcoes auxiliares
#include "utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define os nomes dos arquivos de saida
#define OUTPUT_FILE_CHAINING "scenarioIII_results_chaining.txt"
#define OUTPUT_FILE_LINEAR "scenarioIII_results_linear.txt"
#define OUTPUT_CSV_CHAINING "running_scenarioIII_chaining.csv"
#define OUTPUT_CSV_LINEAR "running_scenarioIII_linear.csv"
// Define o tamanho maximo dos ratings que podem ser carregados na memoria
#define MAX_QUANTITY_EXECUTIONS 3

// Funcao para executar o cenario III
// Recebe os ratings e o numero de ratings
// Verifica se ha ratings para processar
// Se nao houver ratings, imprime uma mensagem e retorna
void runScenarioIII(const MoveRating *ratings, int numRatings)
{
    // Verifica se ha ratings para processar
    if (numRatings <= 0)
    {
        printf("No ratings data to process for Scenario III.\n");
        return;
    }

    // Define os tamanhos de teste para o cenario III
    // Estes tamanhos sao usados para testar a eficiencia dos algoritmos de hashing
    // A lista de tamanhos inclui 1000, 5000, 10000, 50000, 100000, 500000 e 1000000
    int testSizes[] = {1000, 5000, 10000, 50000, 100000, 500000, 1000000};
    int numTestSizes = sizeof(testSizes) / sizeof(testSizes[0]);

    // Abre os arquivos de log para escrita para armazenar os dados do hashing encadeado
    FILE *logFileChaining = fopen(OUTPUT_FILE_CHAINING, "w");
    if (logFileChaining == NULL)
    {
        perror("Error opening log file for writing");
        return;
```

```
44      }
45
46      // Abre os arquivos de log para escrita para armazenar os dados do
            hashing linear
47      FILE *logFileLinear = fopen(OUTPUT_FILE_LINEAR, "w");
48      if (logFileLinear == NULL)
49      {
50          perror("Error opening log file for writing");
51          fclose(logFileChaining);
52          return;
53      }
54
55      // Cria arquivos CSV para armazenar os resultados das execucoes do
            hashing encadeado
56      FILE *csvFileChaining = fopen(OUTPUT_CSV_CHAINING, "w");
57      if (csvFileChaining == NULL)
58      {
59          perror("Error opening CSV file for chaining");
60          fclose(logFileChaining);
61          fclose(logFileLinear);
62          // fclose(csvFileChaining); // This line is redundant as
                csvFileChaining is NULL
63          return;
64      }
65
66      // Cria arquivos CSV para armazenar os resultados das execucoes do
            hashing linear
67      FILE *csvFileLinear = fopen(OUTPUT_CSV_LINEAR, "w");
68      if (csvFileLinear == NULL)
69      {
70          perror("Error opening CSV file for linear probing");
71          fclose(logFileChaining);
72          fclose(logFileLinear);
73          fclose(csvFileChaining);
74          // fclose(csvFileLinear); // This line is redundant as
                csvFileLinear is NULL
75          return;
76      }
77
78      // Itera sobre os tamanhos de teste definidos, para cada tamanho de
            teste
79      // Para cada tamanho de teste, verifica se ha dados suficientes
80      // Caso nao haja dados suficientes, pula para o proximo tamanho
81      // Para cada tamanho de teste, executa o hashing encadeado e o hashing
            linear
82      // Armazena os resultados de comparacao e uso de memoria em arquivos
            CSV
83      // Imprime os resultados medios de comparacao e uso de memoria no log
```

```
84      for (int i = 0; i < numTestSizes; i++)
85      {
86          // Define o tamanho atual do teste
87          // Verifica se o tamanho atual e maior que o numero de ratings
                disponiveis
88          // Se for, imprime uma mensagem e pula para o proximo tamanho
89          int currentSize = testSizes[i];
90          if (currentSize > numRatings)
91          {
92              fprintf(logFileChaining, "\nSkipping size %d: not enough data
                    available (have %d).\n", currentSize, numRatings);
93              continue;
94          }
95
96          // Inicializa as variaveis de comparacao e uso de memoria para o
                hashing encadeado e linear
97          long long totalComparisonsChaining = 0;
98          long totalMemoryChaining = 0;
99          long long totalComparisonsLinear = 0;
100         long totalMemoryLinear = 0;
101
102         // Imprime no log o tamanho atual do teste
103         fprintf(logFileLinear, "\n--- Scenario III: Hashing with N = %d
                ---\n", currentSize);
104         fprintf(logFileChaining, "\n--- Scenario III: Hashing with N = %d
                ---\n", currentSize);
105
106         // Executa o hashing encadeado e o hashing linear para cada
                semente definida, no caso 3 vezes
107         for (int j = 0; j < MAX_QUANTITY_EXECUTIONS; j++)
108         {
109             // Define a semente para a geracao de numeros aleatorios
110             // Cria um subconjunto aleatorio dos ratings para o tamanho
                    atual do teste
111             MoveRating *subset = createRandomSubset(ratings, numRatings,
                    currentSize);
112
113             // Executa o hashing encadeado e o hashing linear
114             // Armazena os resultados de comparacao e uso de memoria em
                    arquivos CSV
115             HashTableChaining *htc = createHashTableChaining(currentSize);
116             for (int k = 0; k < currentSize; k++)
117             {
118                 insertChaining(htc, subset[k]);
119             }
120             long long comparisonsChaining = getComparisonsChaining(htc);
121             long memoryChaining = getMemoryUsageChaining(htc);
122             totalComparisonsChaining += comparisonsChaining;
```

```
123             totalMemoryChaining += memoryChaining;
124             fprintf(csvFileChaining, "%d,%lld,%ld\n", currentSize,
                    comparisonsChaining, memoryChaining);
125
126             // Libera a memoria alocada para o hashing encadeado
127             // Armazena os resultados de comparacao e uso de memoria em
                    arquivos CSV
128             HashTableLinear *htl = createHashTableLinear(currentSize);
129             for (int k = 0; k < currentSize; k++)
130             {
131                 insertLinearProbing(htl, subset[k]);
132             }
133             long long comparisonsLinear = getComparisonsLinear(htl);
134             long memoryLinear = getMemoryUsageLinear(htl);
135             totalComparisonsLinear += comparisonsLinear;
136             totalMemoryLinear += memoryLinear;
137             fprintf(csvFileLinear, "%d,%lld,%ld\n", currentSize,
                    comparisonsLinear, memoryLinear);
138
139             // Libera a memoria alocada para o hashing encadeado
140             freeHashTableChaining(htc);
141             // Libera a memoria alocada para o hashing linear
142             freeHashTableLinear(htl);
143             // Libera a memoria alocada para o subconjunto
144             free(subset);
145         }
146
147         // Imprime os resultados medios de comparacao e uso de memoria no
                log
148         // Calcula a media de comparacoes e uso de memoria para o hashing
                encadeado
149         // Calcula a media de comparacoes e uso de memoria para o hashing
                linear
150         fprintf(logFileChaining, "Chaining - Avg Comparisons: %lld, Avg
                Memory: %ld\n", totalComparisonsChaining /
                MAX_QUANTITY_EXECUTIONS, totalMemoryChaining /
                MAX_QUANTITY_EXECUTIONS);
151         fprintf(logFileLinear, "Linear Probing - Avg Comparisons: %lld,
                Avg Memory: %ld\n", totalComparisonsLinear /
                MAX_QUANTITY_EXECUTIONS, totalMemoryLinear /
                MAX_QUANTITY_EXECUTIONS);
152     }
153
154     // Fecha o arquivo CSV e log apos todas as execucoes
155     fclose(logFileChaining);
156     fclose(logFileLinear);
157     fclose(csvFileChaining);
158     fclose(csvFileLinear);
```

```
159     printf("\nScenario III completed successfully. Results saved to %s, %s
            \n", OUTPUT_FILE_CHAINING, OUTPUT_FILE_LINEAR);
160 }
```

5.11 Arquivo scenario III.h

```
1  #ifndef SCENARIOIII_H
2  #define SCENARIOIII_H
3
4  // Importa o data_handler.h para usar a estrutura MoveRating
5  #include "data_handler.h"
6
7  // Declara a funcao para executar o cenario III
8  void runScenarioIII(const MoveRating *ratings, int numRatings);
9
10 #endif
```

5.12 Arquivo sorting.c

```
1  #include "sorting.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  // Funcao auxiliar para mesclar dois subarrays
6  // Recebe o array, os indices esquerdo, meio e direito, e ponteiros para
7  // as variaveis de comparacao e copia
8  // Realiza a mesclagem dos subarrays e atualiza as metricas de comparacao
       e copia
9  static void merge(MoveRating arr[], int left, int mid, int right, long
       long *comparisons, long long *copies)
10 {
11     int i, j, k;
12     int n1 = mid - left + 1;
13     int n2 = right - mid;
14
15     // Aloca memoria para os subarrays L e R
16     // Verifica se a alocacao foi bem-sucedida
17     // Se falhar, imprime uma mensagem de erro e retorna
18     MoveRating *L = (MoveRating *)malloc(n1 * sizeof(MoveRating));
19     MoveRating *R = (MoveRating *)malloc(n2 * sizeof(MoveRating));
20     if (!L || !R)
21     {
```

```
22          fprintf(stderr, "Failed to allocate memory in merge.\n");
23          free(L);
24          free(R);
25          return;
26      }
27
28      // Preenche o subarray L
29      for (i = 0; i < n1; i++)
30      {
31          L[i] = arr[left + i];
32          (*copies)++;
33      }
34      // Preenche o subarray R
35      for (j = 0; j < n2; j++)
36      {
37          R[j] = arr[mid + 1 + j];
38          (*copies)++;
39      }
40
41      i = 0;
42      j = 0;
43      k = left;
44      // Mescla os subarrays L e R de volta ao array original
45      // Compara os elementos e atualiza as metricas de comparacao e copia
46      // Continua ate que todos os elementos de L e R sejam processados
47      while (i < n1 && j < n2)
48      {
49          (*comparisons)++;
50          if (L[i].moveId <= R[j].userId)
51          {
52              arr[k] = L[i];
53              i++;
54          }
55          else
56          {
57              arr[k] = R[j];
58              j++;
59          }
60          (*copies)++;
61          k++;
62      }
63
64      // Copia os elementos restantes de L e R, se houver
65      // Atualiza as metricas de copia
66      while (i < n1)
67      {
68          arr[k] = L[i];
69          (*copies)++;
```

```
70          i++;
71          k++;
72      }
73
74      // Copia os elementos restantes de R, se houver
75      // Atualiza as metricas de copia
76      while (j < n2)
77      {
78          arr[k] = R[j];
79          (*copies)++;
80          j++;
81          k++;
82      }
83
84      free(L);
85      free(R);
86  }
87
88  // Funcao de ordenacao merge sort
89  // Recebe o array, os indices esquerdo e direito, e ponteiros para as
        variaveis de comparacao e copia
90  // Realiza a ordenacao recursiva do array
91  void mergeSort(MoveRating arr[], int left, int right, long long *
        comparisons, long long *copies)
92  {
93      // Verifica se o array tem mais de um elemento
94      // Se sim, divide o array em dois subarrays e chama mergeSort
            recursivamente
95      // Se nao, retorna, pois o array ja esta ordenado
96      if (left < right)
97      {
98          // Calcula o indice do meio
99          int mid = left + (right - left) / 2;
100
101         // Recursivamente ordena o subarray esquerdo
102         mergeSort(arr, left, mid, comparisons, copies);
103         // Recursivamente ordena o subarray direito
104         mergeSort(arr, mid + 1, right, comparisons, copies);
105         // Mescla os subarrays de volta ao array original
106         merge(arr, left, mid, right, comparisons, copies);
107     }
108 }
109
110 // Funcao de ordenacao insertion sort
111 // Recebe o array, o tamanho, e ponteiros para as variaveis de comparacao
        e copia
112 // Realiza a ordenacao do array usando o algoritmo de insertion sort
113 // Atualiza as metricas de comparacao e copia
```

```c
void insertionSort(MoveRating arr[], int n, long long *comparisons, long long *copies)
{
    // Itera sobre o array, comecando do segundo elemento
    // Compara o elemento atual com os anteriores e insere na posicao correta
    // Atualiza as metricas de comparacao e copia
    // Continua ate que todos os elementos estejam ordenados
    for (int i = 1; i < n; i++)
    {
        MoveRating key = arr[i];
        int j = i - 1;

        (*comparisons)++;
        while (j >= 0 && arr[j].userId > key.userId)
        {
            (*comparisons)++;
            arr[j + 1] = arr[j];
            (*copies)++;
            j--;
        }
        arr[j + 1] = key;
        (*copies)++;
    }
}

// Funcao de ordenacao especial que combina merge sort e insertion sort
// Recebe o array, os indices esquerdo e direito, e ponteiros para as variaveis de comparacao e copia
// Realiza a ordenacao recursiva do array
// Se o tamanho do subarray for menor ou igual a 100, usa insertion sort
// Caso contrario, usa merge sort
// Atualiza as metricas de comparacao e copia
void specialMergeSort(MoveRating arr[], int left, int right, long l
```

5.13 Arquivo sorting.h

```c
#ifndef SORTING_H
#define SORTING_H

// Importa o data_handler.h para usar a estrutura MoveRating
#include "data_handler.h"

// Declara as funcoes de ordenacao
```

```
8  void mergeSort(MoveRating arr[], int left, int right, long long *
       comparisons, long long *copies);
9  void specialMergeSort(MoveRating arr[], int left, int right, long long *
       comparisons, long long *copies);
10
11 #endif
```

5.14 Arquivo utils.c

```
1  #include "sorting.h"
2  #include "utils.h"
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  // Funcao auxiliar para executar e registrar a ordenacao
8  // Recebe o array de MoveRating, o tamanho, o nome da metrica, os arquivos
       de log e CSV, o contador de execucao e o cenario
9  void runAndLogSort(MoveRating *data, int size, const char *metricName,
       FILE *logFile, FILE *csvFile, int count, int scenario)
10 {
11     // Verifica se o array de dados e nulo
12     if (data == NULL)
13         return;
14
15     // Inicializa as variaveis de comparacao e copia
16     long long comparisons = 0, copies = 0;
17
18     // Marca o tempo de inicio da ordenacao
19     clock_t start = clock();
20     // Executa a ordenacao de acordo com o cenario
21     // Se for cenario 1, usa mergeSort, se for cenario 2, usa
           specialMergeSort
22     // specialMergeSort e o merge sort com o insertion sort para pequenos
           arrays
23     if (scenario == 1)
24         mergeSort(data, 0, size - 1, &comparisons, &copies);
25     else if (scenario == 2)
26         specialMergeSort(data, 0, size - 1, &comparisons, &copies);
27     clock_t end = clock();
28
29     // Calcula o tempo gasto na ordenacao
30     // O tempo e calculado como a diferenca entre o tempo de fim e o tempo
           de inicio, dividido pelo CLOCKS_PER_SEC
31     // Isso converte o tempo de clock ticks para segundos
```

```
32      // O resultado e armazenado na variavel timeSpent
33      double timeSpent = ((double)(end - start)) / CLOCKS_PER_SEC;
34
35      // Registra as metricas no arquivo de log
36      fprintf(logFile, "\n--- Metrics from %s ---\n", metricName);
37      fprintf(logFile, "Key Comparisons: %lld\n", comparisons);
38      fprintf(logFile, "Register Copies: %lld\n", copies);
39      fprintf(logFile, "Total Time Spent: %.6f seconds\n", timeSpent);
40
41      // Registra as metricas no arquivo CSV
42      fprintf(csvFile, "%d,%d,%s,%lld,%lld,%.6f\n", count + 1, size,
            metricName, comparisons, copies, timeSpent);
43  }
44
45  // Funcao auxiliar para criar um subconjunto aleatorio de MoveRating
46  // Recebe o array completo de ratings, o total de ratings e o tamanho do
        subconjunto
47  // Retorna um novo array contendo o subconjunto aleatorio
48  MoveRating *createRandomSubset(const MoveRating *allRatings, int
        totalRatings, int subsetSize)
49  {
50      // Aloca memoria para o subconjunto
51      MoveRating *subset = malloc(subsetSize * sizeof(MoveRating));
52      // Verifica se a alocacao foi bem-sucedida
53      if (subset == NULL)
54      {
55          fprintf(stderr, "Memory allocation failed for subset array.\n");
56          return NULL;
57      }
58
59      // Preenche o subconjunto com elementos aleatorios do array original
60      for (int i = 0; i < subsetSize; i++)
61      {
62          int randomIndex = rand() % totalRatings;
63          subset[i] = allRatings[randomIndex];
64      }
65      // Retorna o subconjunto aleatorio
66      return subset;
67  }
```

5.15 Arquivo utils.h

```
1
2   #ifndef UTILS_H
3   #define UTILS_H
```

```
4
5  // Importa o data_handler.h para usar a estrutura MoveRating
6  #include "data_handler.h"
7
8  // Declara as funcoes auxiliares
9  void runAndLogSort(MoveRating *data, int size, const char *metricName,
       FILE *logFile, FILE *csvFile, int count, int scenario);
10 MoveRating *createRandomSubset(const MoveRating *allRatings, int
       totalRatings, int subsetSize);
11
12 #endif
```