PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ

CURSO DE ENGENHARIA DE SOFTWARE

Eduardo Lourenço da Silva

Hector Vieira Saldivar

Theo Rocha Otto

Relatório - PBL Tabela Hash

Curitiba

2025

1. Estrutura e Tratamento de Colisões

O tratamento de colisões escolhido foi o **Encadeamento Exterior** onde cada posição da tabela, que chamamos no código de bucket, armazena uma lista dinâmica (EArrayLIst) com as chaves que colidiram naquele índice.

Essa abordagem foi escolhida por causa da sua simplicidade de implementação e a possibilidade de usar o ArrayList autoral da equipe (EArrayList).

O fator de carga (load factor) escolhido foi de 5, que é um valor propositalmente alto para aumentar o número de colisões, diminuir o tamanho final da tabela e facilitar a obtenção de resultados mais expressivos. Na prática, o fator de carga ideal gira em torno de 0,75. Nos testes um fator de carga baixo gerava uma quantidade massiva de posições na tabela, dificultando as capturas de tela.

2. Função 1 – Baseada no Tamanho da Palavra

A primeira função hash testada foi uma implementação simples que utilizava apenas o comprimento da palavra (key.length()) para determinar a posição de inserção dentro da tabela.

```
public class HashTableSimple extends HashTable<String> {  2 usages  Edua
    @Override  1 usage  Eduardo Lourenço
    public int hashFunction(String key) { return key.length() - 1; }
}
```

```
--Hash Table Simple----------------------
Inserção -> 12,232 ms | 408835,62 ops/s
Qtd. de colisões: 4987 (99,72%)
Busca -> 12,250 ms | 408258,23 ops/s
Encontrados: 5001 de 5001
Total itens: 5001 | Buckets: 1024 | Load factor: 4,884
Buckets vazios: 1010 (98,6%)
Maior bucket: 5 (tam=1258)
```

2.2 Análise de Colisões

- Total de inserções: 5001
- Colisões registradas: 4987
- Taxa de colisão: 99,72%
- Maior bucket: Posição 5, contendo 1258 chaves

```
bucket[00]: 0
bucket[01]: 8
bucket[02]: 125
bucket[03]: 536
bucket[04]: 1218
bucket[05]: 1258
bucket[06]: 952
bucket[07]: 550
bucket[08]: 245
bucket[09]: 83
bucket[10]: 16
bucket[11]: 6
bucket[12]: 2
bucket[13]: 1
bucket[14]: 1
```

Essa função apresentou uma distribuição extremamente concentrada, com a maioria das chaves inseridas nos mesmos índices. O espalhamento foi considerado muito baixo, pois múltiplas palavras diferentes podem ter o mesmo comprimento, levando a colisões inevitáveis.

Mesmo reduzindo o fator de carga, o comportamento continuaria ineficiente devido à natureza da função.

2.3 Desempenho

- Tempo total de inserção: 12ms
- Tempo total de busca: 12,5ms

Apesar de aparentemente baixos, esses tempos se devem ao tamanho reduzido da quantidade de dados (5001) em comparação à um ambiente real com milhares, ou talvez milhões, de registros.

3. Função 2 – Baseada na função hashCode()

A segunda função hash implementada foi uma versão mais complexa, baseada na função hashCode() da classe String e foi retirada da internet assim como foi sugerido na documentação do projeto.

Essa técnica é inspirada na implementação do HashMap do Java, que aplica um deslocamento e operação XOR para misturar os bits altos e baixos do hash original.

Essa abordagem permite que o índice final seja calculado posteriormente com base na capacidade atual da tabela, garantindo que o valor retornado por hashFunction seja independente do tamanho atual da estrutura.

```java
public class HashTableComplex extends HashTable<String> {
    @Override  1 usage  Eduardo Lourenço +1
    public int hashFunction(String key) {
        if (key == null) return 0;
        int h = key.hashCode();
        h ^= (h >>> 16);
        return h;
    }
}
```

```
--Hash Table Complex--------------------
Inserção -> 2,368 ms | 2111641,26 ops/s
Qtd. de colisões: 4809 (96,16%)
Busca -> 0,539 ms | 9271412,68 ops/s
Encontrados: 5001 de 5001
Total itens: 5001 | Buckets: 1024 | Load factor: 4,884
Buckets vazios: 10 (1,0%)
Maior bucket: 250 (tam=13)
```

3.2. Análise de colisões

- Total de inserções: 5001
- Colisões registradas: 4809
- Taxa de colisão: 96,16%
- Maior bucket: posição 250, com 13 chaves
- Buckets vazios: 10 (1,0%)

Apesar de ainda existirem colisões (inevitáveis em função do alto fator de carga escolhido), a distribuição das chaves foi muito mais uniforme em comparação à função anterior.

O maior bucket contém apenas 13 elementos, o que é um resultado excelente, considerando o número total de registros e o fator de carga próximo de 5.

Um ponto importante é que o contador de colisões também incluiu **duplicatas**, que são as chaves que já existiam e não foram reinseridas. Assim, a quantidade real de colisões efetivas é ligeiramente menor.

3.3 Desempenho

- Tempo total de inserção: 2,368ms
- Tempo total de busca: 0,539ms

Esses tempos demonstram uma melhoria significativa de desempenho em relação à função simples.
A redução de colisões e a melhor distribuição dos elementos resultaram em uma queda de aproximadamente 80% no tempo de inserção e 95% no tempo de busca, comprovando a eficiência da nova função hash.

4. Fator de carga recomendável

```
--Hash Table Simple--------------------
Inserção -> 12,921 ms | 387044,35 ops/s
Qtd. de colisões: 4987 (99,72%)
Busca -> 13,255 ms | 377305,82 ops/s
Encontrados: 5001 de 5001
Total itens: 5001 | Buckets: 8192 | Load factor: 0,610
Buckets vazios: 8178 (99,8%)
Maior bucket: 5 (tam=1258)

--Hash Table Complex--------------------
Inserção -> 2,151 ms | 2325181,33 ops/s
Qtd. de colisões: 2084 (41,67%)
Busca -> 0,596 ms | 8386718,09 ops/s
Encontrados: 5001 de 5001
Total itens: 5001 | Buckets: 8192 | Load factor: 0,610
Buckets vazios: 4455 (54,4%)
Maior bucket: 1695 (tam=5)
```

Para complementar a análise das funções hash, foi realizado um experimento adicional reduzindo o **fator de carga** para **0,75**, valor mais apropriado para aplicações reais.

Com essa configuração, a tabela aumenta de tamanho mais cedo, evitando que muitos elementos se acumulem em poucos buckets e, assim, reduzindo a ocorrência de colisões.

Ao olhar os resultados, foi percebido que a **Função Hash Simples** se manteve ineficiente mesmo com um fator de carga menor. Apesar de a tabela conter um número significativamente maior de buckets, a função continuou concentrando a maioria das chaves em uma única posição, resultando em **baixa eficiência de espalhamento**.

Por outro lado, a **Função Hash Complexa** apresentou desempenho muito superior, demonstrando uma **melhor distribuição das chaves** e uma **redução expressiva no número de colisões**.

Essa melhoria reforça que um bom projeto de função hash é mais determinante para o desempenho da estrutura do que apenas o fator de carga em si.

5. Comparativo entre funções Hash

Métrica	Função 1	Função 2	Função 2 (Fator Correto)
Itens inseridos	5001	5001	5001
Colisões registradas	**4987** (99,72%)	**4809** (96,16%)	**2084** (41,67%)
Maior bucket	1258 chaves (posição 5)	13 chaves (posição 250)	5 chaves (posição 1695)
Buckets vazios	1010 (98,16%)	10 (1,0%)	4455 (54,4%)
Fator de carga	5	5	0,75
Tempo de inserção	12 ms	2,368 ms	2,151 ms
Tempo de busca	12,5 ms	0,539 ms	0,596 ms

6. Especificações do computador de testes

Os testes foram feitos em um notebook com as seguintes configurações:

OS: Windows 11 23H2

Processador: 12th Gen Intel(R) Core(TM) i5-12450H   2.00 GHz

RAM: 12,0 GB