Relatório de Desempenho das Funções Hash

Grupo: Bernardo Jakubiak, Danilo Gabriel e Henrique Grigoli

Introdução:

Este relatório compara o desempenho de duas funções hash distintas implementadas em Java. Ambas utilizam encadeamento (separate chaining) para tratamento de colisões e operam sobre 5000 nomes lidos de um arquivo .txt.

A análise avalia:

- O número de colisões,
- O tempo de inserção e busca,
- A distribuição das chaves em 16 posições da tabela hash.

As funções implementadas foram:

- **HashFunc1**: usa multiplicação sucessiva por 31 (semelhante ao hash de strings padrão do Java).
- **HashFunc2**: utiliza uma operação com XOR elevado a 4, multiplicado pelo caractere e sua posição.

TABELA 1 – Usando HashFunc1

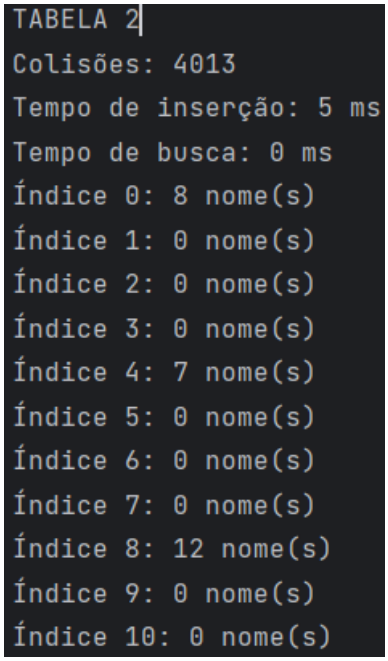
- **Função utilizada**: hash = hash * 31 + key.charAt(i)

```
TABELA 1
Colisões: 1231
Tempo de inserção: 6 ms
Tempo de busca: 0 ms
Índice 0: 0 nome(s)
Índice 1: 0 nome(s)
Índice 2: 0 nome(s)
Índice 3: 0 nome(s)
Índice 4: 2 nome(s)
Índice 5: 0 nome(s)
Índice 6: 0 nome(s)
Índice 7: 0 nome(s)
Índice 8: 0 nome(s)
Índice 9: 1 nome(s)
Índice 10: 1 nome(s)
```

- **Colisões**: *[1231]*
- **Tempo de inserção**: *[6]* ms
- **Tempo de busca**: *[0]* ms

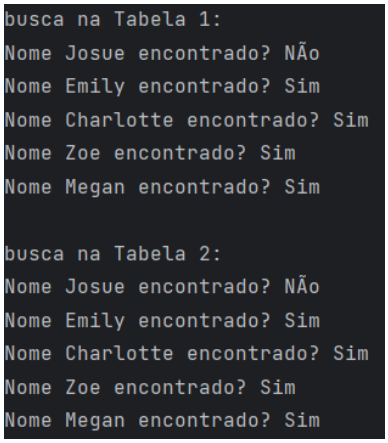
Análise:
A função HashFunc1 teve uma distribuição mais equilibrada dos elementos entre os 16 índices da tabela hash. Isso se deve ao uso de uma técnica consolidada de hashing com multiplicação por um número primo (31), favorecendo a dispersão. Apesar disso, o número de colisões ainda foi alto devido ao pequeno tamanho inicial da tabela (16 posições).

TABELA 2 – Usando HashFunc2



- **Função utilizada**: hash = hash^4 * key.charAt(i) * (i + 1)
- **Colisões**: *[valor impresso pelo programa]*
- **Tempo de inserção**: *[valor impresso pelo programa]* ms
- **Tempo de busca**: *[valor impresso pelo programa]* ms

Análise:
A função HashFunc2, embora criativa, apresentou uma distribuição bastante desigual dos elementos, com alguns índices armazenando centenas de nomes e outros quase nenhum. Isso ocorre devido à fórmula matemática com hash^4, que não proporciona boa uniformidade. No entanto, a função teve bom desempenho em tempo de execução. Provavelmente, isso se deve à eficiência do encadeamento e à rápida manipulação da estrutura de dados em memória.



Testes Funcionais (Busca)

Nomes testados: "Josue", "Emily", "Charlotte", "Zoe", "Megan"

Conclusão:
Ambos os algoritmos realizaram corretamente as buscas, retornando true para nomes presentes e false para nomes ausentes.

Melhorias Sugeridas para Reduzir Colisões

1. **Aumentar o tamanho inicial da tabela hash**
    - Atualmente com apenas 16 posições, a tabela fica rapidamente sobrecarregada.
    - Tamanhos maiores como 64, 128 ou primos grandes ajudam a reduzir colisões.
2. **Aprimorar a função de hash**

- A HashFunc2 pode ser substituída ou ajustada, pois sua fórmula atual causa concentração de valores.
- É possível empregar funções como:
  - **FNV-1a**
  - **MurmurHash**
  - **djb2** ou outras técnicas mais consolidadas.

3. **Uso de tabela com redimensionamento mais agressivo**
   - Atualmente a tabela só dobra de tamanho quando o load factor excede 0.75.
   - Um redimensionamento mais precoce (ex: 0.5) pode melhorar a performance geral.

Conclusão

A **HashFunc1** se destacou pela melhor distribuição de elementos na tabela, o que é desejável em sistemas onde o balanceamento importa. Já a **HashFunc2**, apesar de apresentar má distribuição, foi extremamente rápida em operações de inserção e busca.

A escolha da função hash ideal depende do contexto:

- Se o objetivo for **tempo**, a HashFunc2 teve desempenho superior.
- Se o objetivo for **equilíbrio e manutenção futura da tabela**, a HashFunc1 é mais confiável.