Integrantes: Richard de Oliveira, Priscila Bueno e Nathasha da Cruz

# Relatório Tabela Hash

# Funções Hash

Utilizamos 2 funções hash distintas, sendo elas:

- **Soma dos valores ASCII:**

```
int sum = 0;
for (int i = 0; i < key.length(); i++) {
   sum += key.charAt(i);
}
return sum % capacity;
```

Essa função soma os códigos ASCII dos caracteres e aplica o módulo operador.

É simples, mas gera colisões para chaves com letras semelhantes ou tamanhos parecidos.

- **Hash Polinomial:**

```
int hash = 0;
int p = 31;
int m = capacity;

for (int i = 0; i < key.length(); i++) {
   hash = (hash * p + key.charAt(i)) % m;
}
return hash;
```

Essa função é mais elaborada, utilizando um método polinomial (similar ao algoritmo Rabin-Karp).

Ela espalha melhor os valores e reduz colisões, mantendo boa performance.

# Tratamento de Colisões

O método de encadeamento exterior (separado) foi utilizado.

Cada posição da tabela armazena uma lista encadeada simulada com arrays, permitindo múltiplas chaves no mesmo índice.

Assim, quando ocorre uma colisão, o novo elemento é adicionado no final da lista daquela posição.

Vantagens:

- Fácil implementação
- Boa performance com poucas colisões
- Permite redimensionamento simples (rehash)

# Fator de Carga e Redimensionamento

A tabela foi feita com a capacidade inicial sendo 8 e a capacidade máxima sendo 32.

O fator de carga foi definido em 0.75 conforme o material disponibilizado.

O rehash ocorre quando:

```
public double getLoadFactor() {
   return (double) size / capacity;
}
if (getLoadFactor() > loadFactor //(sendo 0.75)// && capacity
< MAX_CAPACITY) {
   rehash();
}
```

Assim a capacidade é dobrada, tendo o limite de 32 e todos os elementos sendo remapeados.

# Testes de eficiência

**Tabela A**

- Colisões: 4968
- Tempo de inserção: 17.6072 ms

**Distribuição das chaves:**

Posição 0: 160 chave(s)
Posição 1: 171 chave(s)
Posição 2: 161 chave(s)
Posição 3: 146 chave(s)
Posição 4: 150 chave(s)
Posição 5: 149 chave(s)
Posição 6: 167 chave(s)
Posição 7: 159 chave(s)
Posição 8: 164 chave(s)
Posição 9: 146 chave(s)
Posição 10: 155 chave(s)
Posição 11: 149 chave(s)
Posição 12: 176 chave(s)
Posição 13: 150 chave(s)
Posição 14: 182 chave(s)
Posição 15: 148 chave(s)
Posição 16: 139 chave(s)
Posição 17: 146 chave(s)
Posição 18: 162 chave(s)
Posição 19: 160 chave(s)
Posição 20: 148 chave(s)
Posição 21: 152 chave(s)
Posição 22: 158 chave(s)
Posição 23: 159 chave(s)
Posição 24: 147 chave(s)
Posição 25: 148 chave(s)
Posição 26: 157 chave(s)
Posição 27: 147 chave(s)
Posição 28: 160 chave(s)
Posição 29: 159 chave(s)
Posição 30: 177 chave(s)
Posição 31: 148 chave(s)

**Tabela B**

- Colisões: 4968
- Tempo de inserção: 5.6844 ms

**Distribuição das chaves:**
Posição 0: 161 chave(s)
Posição 1: 165 chave(s)
Posição 2: 164 chave(s)
Posição 3: 156 chave(s)
Posição 4: 159 chave(s)
Posição 5: 166 chave(s)
Posição 6: 154 chave(s)
Posição 7: 144 chave(s)
Posição 8: 171 chave(s)
Posição 9: 144 chave(s)
Posição 10: 163 chave(s)
Posição 11: 141 chave(s)
Posição 12: 178 chave(s)
Posição 13: 146 chave(s)
Posição 14: 154 chave(s)
Posição 15: 147 chave(s)
Posição 16: 140 chave(s)
Posição 17: 139 chave(s)
Posição 18: 163 chave(s)
Posição 19: 141 chave(s)
Posição 20: 144 chave(s)
Posição 21: 158 chave(s)
Posição 22: 176 chave(s)
Posição 23: 148 chave(s)
Posição 24: 151 chave(s)
Posição 25: 169 chave(s)
Posição 26: 149 chave(s)
Posição 27: 165 chave(s)
Posição 28: 159 chave(s)
Posição 29: 160 chave(s)
Posição 30: 177 chave(s)
Posição 31: 148 chave(s)

**Testando buscas:**
**Nome: Mary**
- Tabela A: Encontrado (35000 ns)
- Tabela B: Encontrado (17200 ns)

**Nome: Sophia**
- Tabela A: Encontrado (21900 ns)
- Tabela B: Encontrado (21100 ns)

**Nome: Olivia**

- Tabela A: Encontrado (4600 ns)
- Tabela B: Encontrado (11900 ns)

**Nome: Nonexistent**

- Tabela A: Não encontrado (20900 ns)
- Tabela B: Não encontrado (21400 ns)

**Análise dos resultados:**

- Ambas as tabelas apresentaram exatamente o mesmo número de colisões. Isso indica que, embora as funções hash sejam diferentes, a distribuição final acabou sendo muito semelhante devido ao conjunto de dados (5000 nomes) e à capacidade final fixa (32 posições).

| Tabela | Tempo de inserção |
|---|---|
| A | 17.6 ms |
| B | 5.7 ms |

A função polinomial, mesmo sendo teoricamente mais complexa, foi computacionalmente mais eficiente nesse contexto, possivelmente devido à melhor dispersão intermediária que reduziu o custo de realocação interna de arrays (nas colisões encadeadas).

Já a função baseada em soma ASCII, embora simples, pode ter gerado mais concentrações de chaves em certos índices ao longo do processo, aumentando o custo de cópia dos arrays internos.

| Tabela | Mínimo | Máximo |
|---|---|---|
| A | 139 | 182 |
| B | 139 | 178 |

A Tabela B apresentou uma distribuição ligeiramente mais equilibrada, com menos extremos (menos posições muito cheias ou muito vazias).

Isso confirma que a função polinomial espalha as chaves de forma mais homogênea que a soma simples de caracteres.

Entretanto, o limite rígido de 32 posições faz com que a diferença prática na distribuição seja pequena, ambas acabam sofrendo de alta densidade de dados.

| Nome | Tabela A | Tabela B |
|---|---|---|
| Mary | 35 000 ns | 17 200 ns |
| Sophia | 21 900 ns | 21 100 ns |
| Olivia | 4 600 ns | 11 900 ns |
| Nonexistent | 20 900 ns | 21 400 ns |

As buscas ficaram muito próximas em tempo médio, variando conforme a posição da chave na lista encadeada.

A diferença entre as tabelas foi pequena, nenhuma função foi claramente superior em todos os casos.