Tabelas Hash – Estrutura de Dados Eficiente

Com base no livro 'Aprendendo Algoritmos', de Aditya Y. Bhargava

O que é uma Tabela Hash?

- Imagine um supermercado: você quer saber o preço da banana
- Em vez de olhar produto por produto nas prateleiras, você vai direto ao funcionário e pergunta o preço da banana
- você fornece a chave ("banana") e ele retorna o valor (R$12) rapidamente
- O funcionário é um hash table

O que é uma Tabela Hash?

- Estrutura de dados que associa chaves a valores
- Também chamadas de dicionários ou mapas
- Exemplo: {"maçã": 18, "banana": 12, "laranja": 15}

Para que serve?

- Busca rápida de valores a partir de uma chave
- Substitui buscas lineares em listas
- Muito usada em
  - Verificação de existência de elementos
  - Modelar a relação entre dois elementos
  - Caching / memorização de dados

Uso de Hashing em Sistemas de Cache

- Além de armazenar dados de forma eficiente, tabelas hash são amplamente utilizadas para implementar sistemas de cache.

Como funciona um cache com hashing

- A chave representa uma entrada (ex.: URL, ID de recurso, parâmetros de função).
- A função hash gera um índice que aponta para o valor armazenado.
- Se a chave já estiver na tabela: devolve-se o valor (cache hit).
- Caso contrário, o valor é computado, armazenado e retornado (cache miss).
- Esse processo evita cálculos repetidos e acessos redundantes a dados externos.

Vantagens do Cache com Hashing

- Busca em tempo constante (O(1)), ideal para sistemas de alto desempenho.
- Menor uso de recursos computacionais em chamadas repetidas.
- Redução de latência em sistemas web ou aplicações distribuídas.
- Flexível: pode armazenar resultados de funções, páginas web, requisições a banco etc.

Exemplos de uso de cache com hash

- DNS Cache: associa nomes de domínio a IPs resolvidos recentemente.
- Web Cache: armazena páginas e recursos estáticos para evitar novos downloads.
- Memoization: em programação dinâmica, armazena resultados de chamadas recursivas.
- Sistemas de recomendação e inteligência artificial: armazenam inferências recentes.
- Compiladores: cache de análise sintática ou tokens reutilizáveis.

Como funciona?

- Usa uma função hash que transforma a chave em um índice numérico
- Esse índice aponta para uma posição no array
- Colisões podem ocorrer — duas chaves com o mesmo índice

Taxa de Ocupação Ideal

- O que é a taxa de ocupação?
  - É a razão entre o número de elementos armazenados e o tamanho total da tabela:

$$\text{Taxa de Ocupação} = \frac{\text{nº de elementos}}{\text{tamanho da tabela}}$$

- Por que ela é importante?
  - Taxas muito altas aumentam colisões e reduzem o desempenho.
  - Taxas muito baixas desperdiçam memória.

Taxa de Ocupação Ideal

- Qual é o valor ideal?
  - Entre 0,5 e 0,75 na maioria dos casos.
  - Em endereçamento aberto: ideal < 0,7.
  - Em encadeamento: pode ser maior, mas com impacto no desempenho.

- Dica prática:
  - Use reestruturação (rehashing) ao ultrapassar o limite para manter a eficiência.

Complexidade de Tempo

Operação	Tempo Médio	Pior Caso
Inserção	O(1)	O(n)
Remoção	O(1)	O(n)
Busca	O(1)	O(n)

- Desempenho ideal depende de boa função hash e baixa taxa de ocupação

Comparando

Operação	Hash Caso Médio	Hash Pior Caso	Arrays	Listas Ligadas
Inserção	O(1)	O(n)	O(n)	O(1)
Remoção	O(1)	O(n)	O(n)	O(1)
Busca	O(1)	O(n)	O(1)	O(n)

- Tabelas hash são o melhor dos 2 mundos

Colisões

- Quando duas chaves geram o mesmo índice em uma tabela hash, ocorre uma colisão.
- Existem diferentes estratégias para resolver esse problema.
- Vamos conhecer as principais.

Lidando com Colisões

- Estratégias comuns
  - Encadeamento (chaining): listas ligadas por posição
  - Endereçamento aberto: procurar próxima posição livre

- Importante lembrar que cada escolha tem um impacto sobre o desempenho

1. Encadeamento (Chaining)

- Cada posição da tabela aponta para uma lista encadeada de elementos.
- Quando ocorre colisão, os elementos são adicionados nessa lista.
- Vantagens:
  - Fácil de implementar.
- Desvantagens:
  - Pode ter listas muito grandes se houver muitas colisões.

2. Endereçamento Aberto (Open Addressing)

- Todos os elementos ficam armazenados diretamente na tabela.
- Se houver colisão, procura-se a próxima posição livre.
- Formas comuns de sondagem:
  - Linear (i+1, i+2, ...)
  - Quadrática ($i+1^2$, $i+2^2$, ...)
  - Hash duplo (usa uma segunda função hash)
- Vantagens:
  - Não usa ponteiros nem listas auxiliares.
- Desvantagens:
  - Tende a degradar com alta taxa de ocupação.

3. Outras Estratégias Menos Comuns

- Hashing perfeito:
  - Usado quando todas as chaves são conhecidas antecipadamente.
  - Sem colisões, mas difícil de construir.
- Hashing com rehashing:
  - Criação de nova tabela maior com nova função hash quando o limite é atingido.
- Hashing com buckets:
  - Cada posição guarda vários elementos até um limite, semelhante a blocos.

Exemplo prático em Java

```
import java.util.HashMap;

public class ExemploHash {
    public static void main(String[] args) {
        HashMap<String, String> livros = new HashMap<>();
        livros.put("1984", "George Orwell");
        livros.put("Duna", "Frank Herbert");
        System.out.println(livros.get("Duna"));
    }
}
```

Exemplo prático em Python

```
livros = {}
livros["1984"] = "George Orwell"
livros["Duna"] = "Frank Herbert"
print(livros["Duna"])  # Saída: Frank Herbert
```

OS EXEMPLOS MOSTRAM STRING PARA STRING...

Quando uma tabela hash que mapeia strings para strings (como HashMap<String, String> em Java), o tipo de alocação de memória e o posicionamento seguem uma lógica comum à maioria das implementações modernas de tabelas hash.

Alocação da Tabela (Array Base)

- A tabela hash é geralmente implementada como um array de ponteiros (referências) para entradas (pares chave-valor).
- Esse array é alocado estaticamente ou dinamicamente com um tamanho inicial definido (ex.: 16 posições).
- Cada posição do array pode
  - Estar vazia (null)
  - Apontar para uma entrada (Map.Entry)
  - Apontar para o início de uma estrutura de colisão (lista encadeada, árvore etc.)

Hashing da Chave String

- A chave do tipo String é convertida em um número inteiro via uma função hash (ex.: hashCode() do Java).

- Esse inteiro é então reduzido ao intervalo do array com uma operação como hash % tamanho_array.

Alocação das Entradas

- Cada par (chave, valor) é representado por um objeto (por exemplo, um Map.Entry<String, String>) que é alocado no heap, independentemente da posição no array.

- A tabela apenas aponta para esses objetos.

Posicionamento em Memória

- O array está em um bloco contíguo de memória (como qualquer array).
- As entradas (pares chave-valor) são objetos alocados no heap.
- A posição delas na memória não é contígua e depende do gerenciamento de heap da JVM ou do sistema (dependendo da linguagem).

```java
HashMap<String, String> mapa = new HashMap<>();
mapa.put("banana", "amarela");
```

`"banana".hashCode()` → inteiro hash → mapeado para índice i

`table[i]` aponta para `Entry("banana", "amarela")` alocado no heap