Hashing

Referências

- Hashing (PF),
- Hash Tables (S&W),
- slides (S&W)

Vídeo

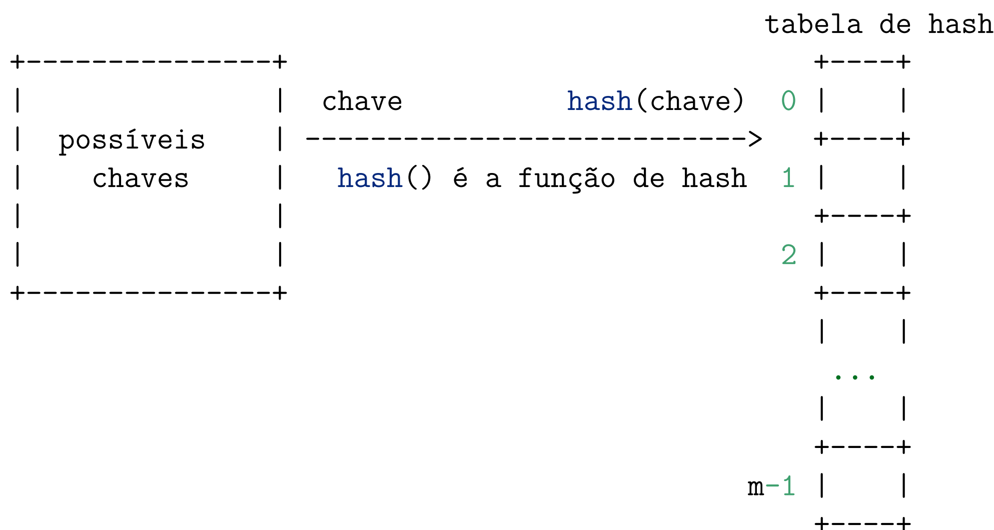
Hashing Functions (S&W)

Tabelas de hash

Uma **tabela de hash** é de certa forma uma generalização de um vetor.

Temos um universo U de possíveis chaves e desejamos construir um esquema que associa cada chave a idealmente uma posição de um vetor ou tabela onde armazenamos o valor associado a chave.

O esquema ou função que associa as chaves a índices desta tabela é chamada de **função de hash**.



Alguns parâmetros importantes

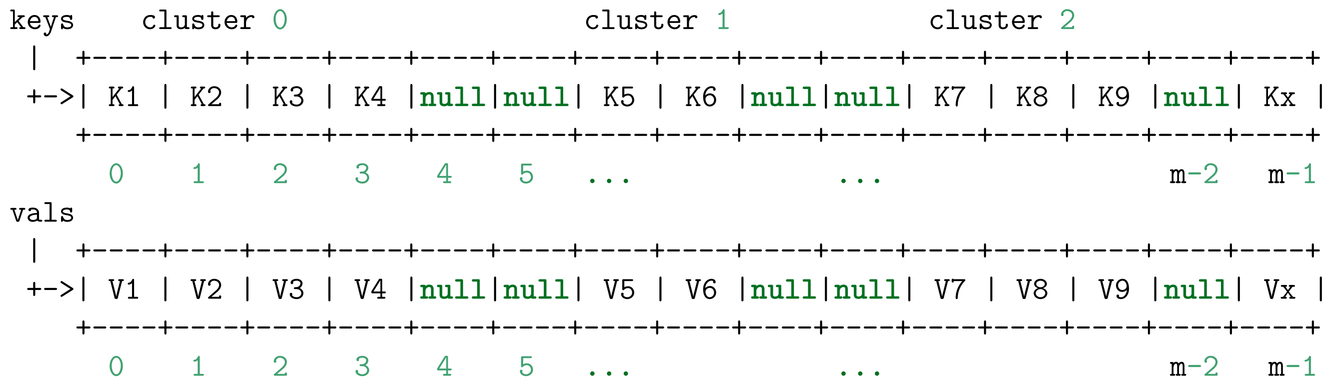
Parâmetros importantes:

- `m` = número de posições na tabela de hash
- `n` = número de chaves da tabela de símbolos
- α = `n/m` = fator de carga (=*load factor*)

Colisões: sondagem linear

Outro jeito de resolver colisões é a sondagem linear (*linear probing*): se uma posição da tabela estiver ocupada, tente a próxima!

```
    private Key[] keys;
    private Value[] vals;
```



Há três possibildades:

- encontramos a chave, paramos a busca;
- posição não-ocupada, paramos a busca;
- posição está ocupada e não é a chave, vamos para a próxima posição

```
public class LinearProbingHashST<Key, Value> {

    private int n;
    private int m = 16;
    private Key[] keys;
    private Value[] vals;

    public LinearProbingHashST() {
        keys = (Key[])   new Object[m];
        vals = (Value[]) new Object[m];
    }

    public LinearProbingHashST(int cap) {
        m = cap;
        keys = (Key[])   new Object[m];
        vals = (Value[]) new Object[m];
    }

    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % m;
    }

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i + 1) % m)
            if (keys[i].equals(key))
                return vals[i];
        return null;
    }

    public void put(Key key, Value val) {
        int i;
```

```
        for (i = hash(key); keys[i] != null; i = (i + 1) % m)
            if (keys[i].equals(key)) {
                vals[i] = val;
                return;
            }
        keys[i] = key;
        vals[i] = val;
        n++;
    }
}
```

Redimensionamento da tabela de sondagem linear

Na sondagem linear, é essencial que α fique bem abaixo de 1.

Convém manter alpha $\leq 1/2$ (ou seja, $n \leq m/2$).

Para manter alpha sob controle, a tabela de hash deve ser redimensionada, quando necessário, no início de put().

```
    public void put(Key key, Value val) {
        if (n >= m/2) resize(2*m);
        . . .
        . . .
    }

    private void resize(int cap) {
        LinearProbingHashST<Key, Value> temp;
        temp = new LinearProbingHashST<Key, Value>(cap);
        for (int i = 0; i < m; i++) {
            if (keys[i] != null) {
                temp.put(keys[i], vals[i]);
            }
        }
        keys = temp.keys;
        vals = temp.vals;
        m    = temp.m;
    }
```

Análise

O consumo de tempo de uma busca em tabelas de hash com sondagem linear depende, no pior caso, do tamanho do maior *cluster* (=fatia da tabela com chaves não nulas).

Proposição: Supondo que vale a hipótese do hashing uniforme, e que alpha está entre 0 e 1 mas não muito perto de 1, o número médio de sondagens em buscas bem-sucedidas é aproximadamente

$$\frac{1}{2}(1 + \frac{1}{(1 - \alpha)})$$

e o número médio de sondagens em buscas malsucedidas (ou inserções) é aproximadamente

$$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$$

Exemplo: quando $\alpha = 0.5$, temos aproximadamente 1.5 sondagens por busca bem-sucedida e aproximadamente 2.5 sondagens por busca malsucedida.

Exemplo: quando $\alpha = 0.25$, temos aproximadamente 1.16 sondagens por busca bem-sucedida e aproximadamente 1.39 por busca malsucedida.

Memória

```
método                  espaço usado para N itens
--------------------    -------------------------
separating chaining     ~48N + 64M
linear probing          entre ~32N e ~128N
BSTs                    ~56N
```