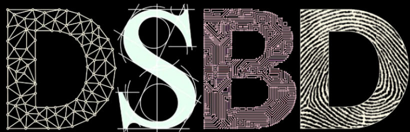
> “Todos os problemas na Ciência da Computação podem ser resolvidos com novos níveis de indireção, exceto o problema de se ter muitos níveis de indireção” (David J. Wheeler).

Tabela Hash

Paulo Ricardo Lisboa de Almeida



Custos

Suponha que vamos implementar um dicionário utilizando as estruturas estudadas até o momento.

Qual o custo das operações?

```
dicionarioIdade['Joao’]  = 19
dicionarioIdade['Maria]  = 20
imprimir(dicionarioIdade['Maria])
```

Custos

Suponha que vamos implementar um dicionário utilizando as estruturas estudadas até o momento.

Qual o custo das operações?

Inserir, buscar e remover custam $O(\log_2 n)$ se usarmos uma árvore de busca binária balanceada.

Ex.: AVL ou Red-Black.

```
dicionarioIdade['Joao’]  = 19
dicionarioIdade['Maria]  = 20
imprimir(dicionarioIdade['Maria])
```

Tabela Hash

Problema:

Montar um dicionário onde o acesso aos dados seja tāo eficiente quanto acessar os dados em um vetor.

Custo constante: *O(1)*.

```
dicionarioIdade['Joao’]  = 19 //O(1)
dicionarioIdade['Maria]  = 20 //O(1)
imprimir(dicionarioIdade['Maria]) //O(1)
```

Ideias?

Suponha que precisamos criar um dicionário onde a chave é o CPF.

O CPF é composto por 11 dígitos.

Exemplo: 999.999.999-99

Supondo que a chave é única (duas pessoas não compartilham um CPF), como fazer um sistema de cadastro onde a inclusão, exclusão, e busca custem *O(1)* de tempo?

```
dicionario[99999999999] = “Dados da Pessoa”
```

Ideia

Criar um vetor de 100.000.000.000 posições.

```
dicionario[99999999998] = “Dados da Pessoa”
```

0	1	2	...	99999999998	99999999999
				“Dados da Pessoa”	

Problemas?

Problemas?

Problemas?

Problemas?

Muita memória pode ser desperdiçada.

Precisamos de alguma forma para mapear chaves que não mapeiam diretamente para uma posição do vetor.

- Exemplo: um dicionário onde a chave é um nome. Para que posição do vetor podemos mapear “Maria”?

Tabela de Endereçamento Direto

A ideia anterior é chamada de T**abela de Endereçamento Direto.**

Suponha um universo *U = {0, 1, ..., m-1}* de chaves possíveis.

Se *m* **não é grande**, usar uma tabela de endereçamento direto pode ser uma boa ideia.

A tabela é *T[0:m-1]*.

Cada posição da tabela é chamada de ***slot*** ou ***bucket***.

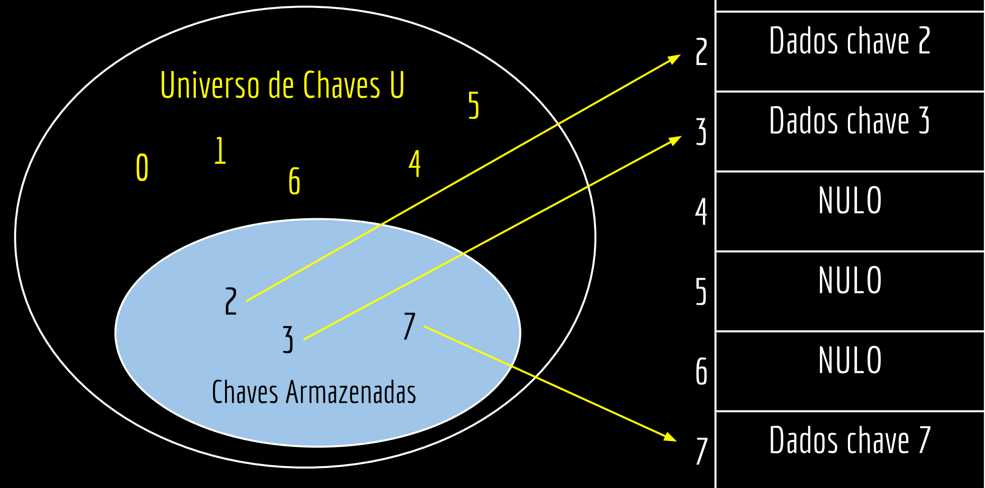


Tabela de Endereçamento Direto

O custo das operações com Endereçamento Direto é *O(1)*.

```
buscar(T,k)
    retorne T[k]

inserir(T,x)
    T[x.chave] = x

excluir(T,x)
    T[x.chave] = NULO
```

Tabela Hash

O endereçamento direto é um problema se:

- $|U|$ é muito grande.
    - Não vamos ter espaço.
- A quantidade de chaves armazenadas é muito menor do que $|U|$.
    - Desperdício de espaço.

Uma questão de espaço

Considere $|U|$ o tamanho do universo de chaves possíveis, e $n = |K|$ o tamanho do conjunto de chaves realmente armazenadas.

Qual o tamanho do armazenamento necessário para o endereçamento direto?

Uma questão de espaço

Considere $|U|$ o tamanho do universo de chaves possíveis, e $n = |K|$ o tamanho do conjunto de chaves realmente armazenadas.

Tamanho do armazenamento necessário para o endereçamento direto.

$\Theta(|U|)$

Em uma Tabela Hash, o custo de espaço é:

$\Theta(|K|) = \Theta(n)$

Tabela Hash

Uma **Tabela Hash** também é conhecida como **Tabela de Espalhamento** ou **Tabela de Dispersão**.

O custo do armazenamento é proporcional ao número de chaves armazenadas.

Custo armazenamento = $\Theta(n)$.

As operações de busca, inserção e exclusão ainda custam O(1).

Tabela Hash

Uma **Tabela Hash** também é conhecida como **Tabela de Espalhamento** ou **Tabela de Dispersão**.

O custo do armazenamento é proporcional ao número de chaves armazenadas.

Custo armazenamento = $\Theta(n)$.

As operações de busca, inserção e exclusão ainda custam O(1).

Mas agora isso só será verdade para o caso médio.

No pior caso, essas operações vão custar $\Theta(n)$.

Função de Hash

Considere um universo de chaves *U*, e uma Tabela Hash *T[0:m-1]*, onde *|U| ‹‹ m*.

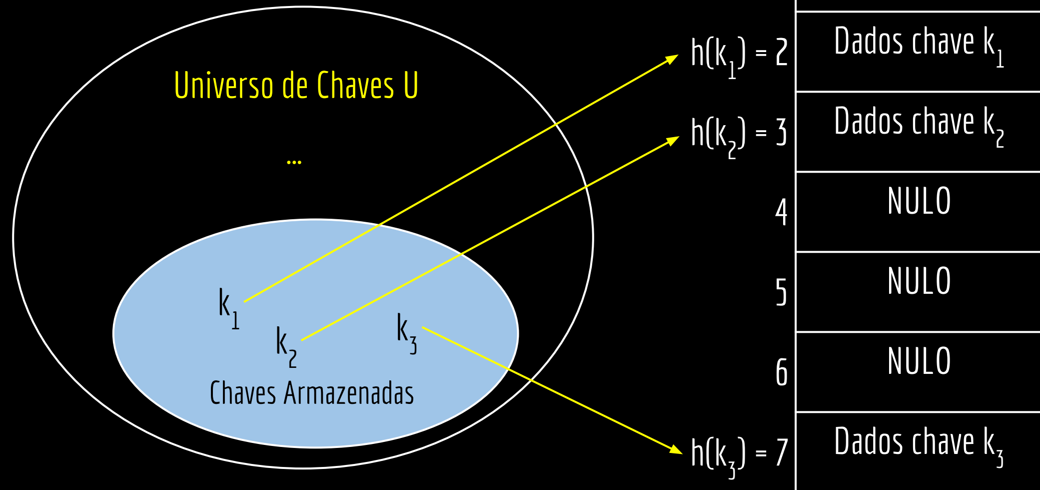
Ou seja, o tamanho da tabela é muito menor que a quantidade de chaves possíveis.

Criar uma função hash *h(k)*, que mapeia uma chave $k \in U$ para um slot em *T*.

$$h : U \mapsto \{0, 1, \ldots, m - 1\}$$

Exemplo

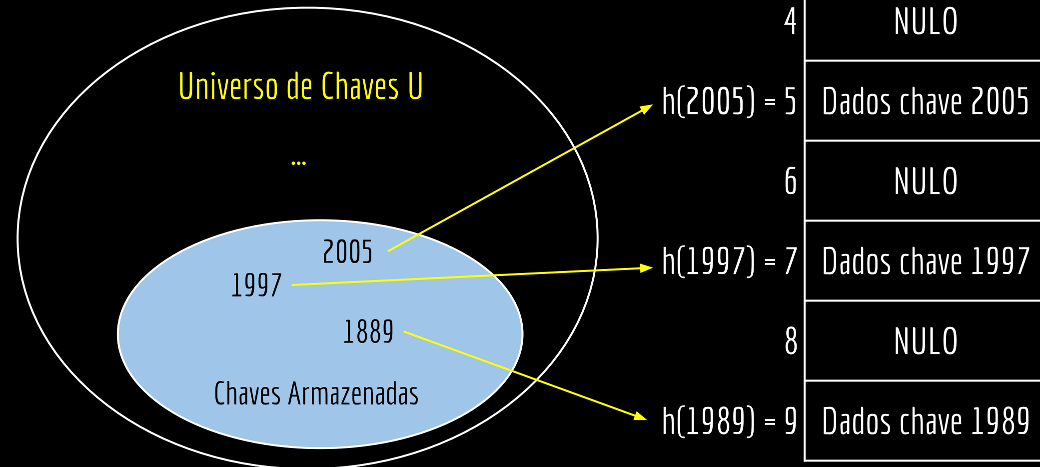
```
inserir(T,x)
    T[h(x.chave)] = x
```



Função de Hash

Vamos considerar uma função de hash simples (mas não muito boa).

$h(x) = x \bmod m$

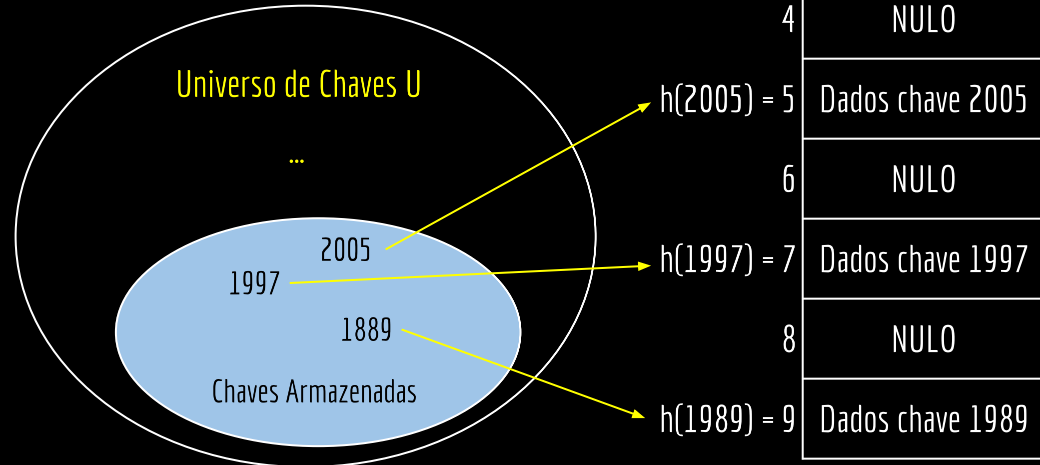


Função de Hash

$h(x) = x \bmod m$

Quando a função h(x) = x mod m pode se mostrar ruim?

Independente da função de hash, quando podemos ter problemas?



$h(x) = x \bmod m$

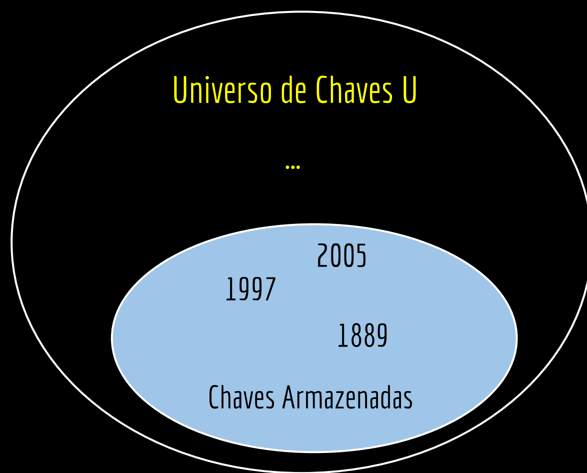
Considere a Tabela Hash T[0:99], e que as chaves são os anos de nascimento das pessoas.

Em um sistema de cadastro qualquer, é muito provável que existam mais pessoas nascidas entre 2000 e 2010, quando comparado com as pessoas nascidas entre 1920 e 1930, por exemplo.

Para a função $h(x) = x \bmod m$

Os slots 20:30 serão subutilizados quando comparados com os slots 0:10.

A função tende a agrupar os dados.



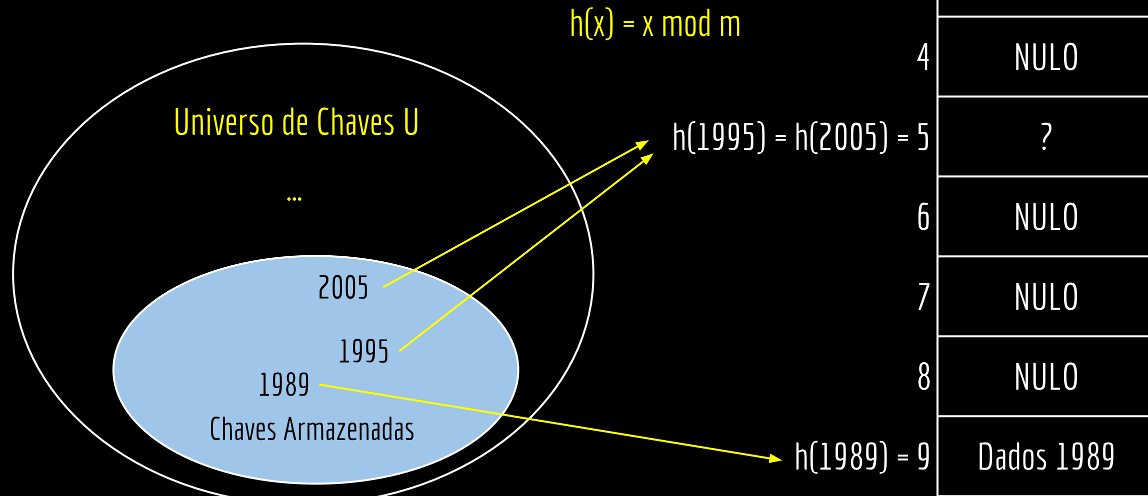
T

	T
0	NULO
1	NULO
2	NULO
3	NULO
4	NULO
5	Dados chave 2005
...	...
97	Dados chave 1997
98	NULO
99	NULO

Colisões

Mesmo com uma função de hash perfeita, colisões podem acontecer.

É garantido que uma **colisão** vai acontecer ao armazenar a chave k_{m+1} na tabela T[0:m-1].



Tratando colisões

Como tratar?

Encadeamento

Podemos resolver colisões por listas encadeadas.

h(x) = x mod m

Universo de Chaves U

...

2005

1995

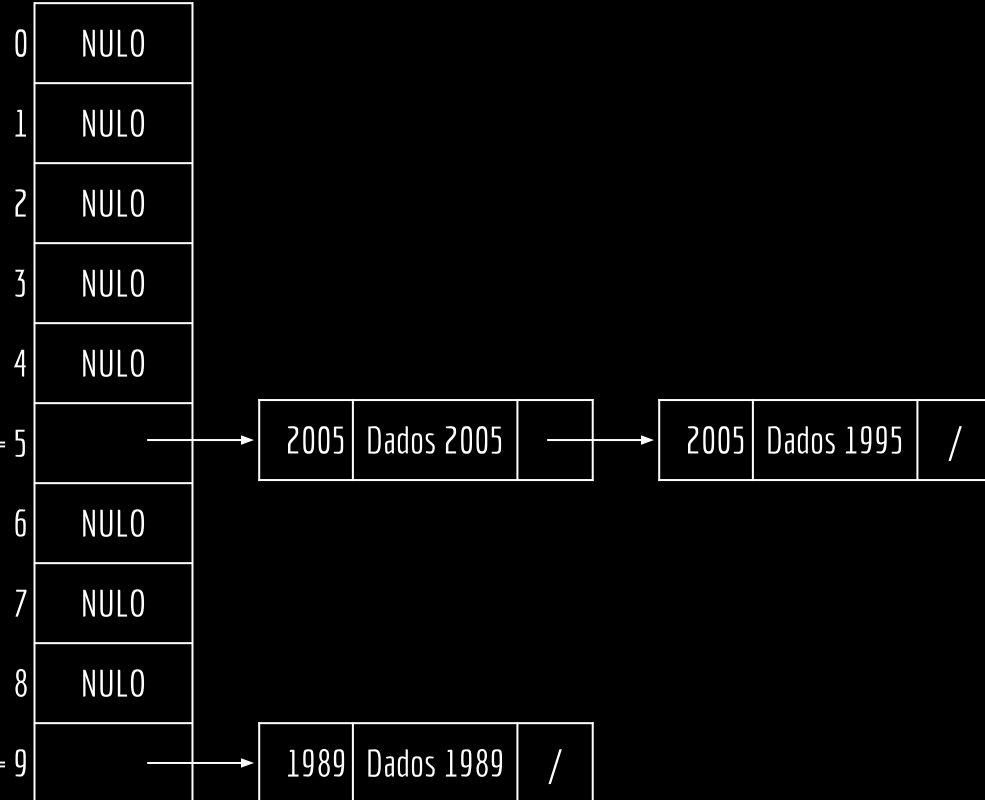
1989

Chaves Armazenadas

h(1995) = h(2005) = 5

h(1989) = 9

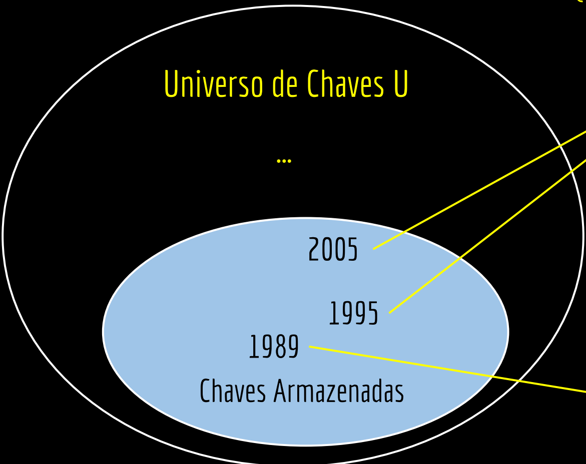
T



Encadeamento

Podemos resolver colisões por listas encadeadas.

```
buscar(T,k)
     buscar por k na lista T[h(k)]

inserir(T,x)
     inserir x na cabeça de T[h(x.chave)]

excluir(T,x)
     excluir x da lista T[h(x.chave)]
```



$h(x) = x \bmod m$

h(1995) = h(2005) = 5

h(1989) = 9

T

	T
0	NULO
1	NULO
2	NULO
3	NULO
4	NULO
5	→ 2005 | Dados 2005 | → 2005 | Dados 1995 | /
6	NULO
7	NULO
8	NULO
9	→ 1989 | Dados 1989 | /

Custo

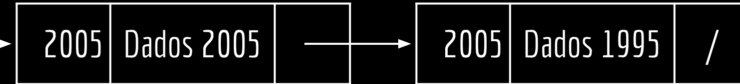
Qual o custo das operações?

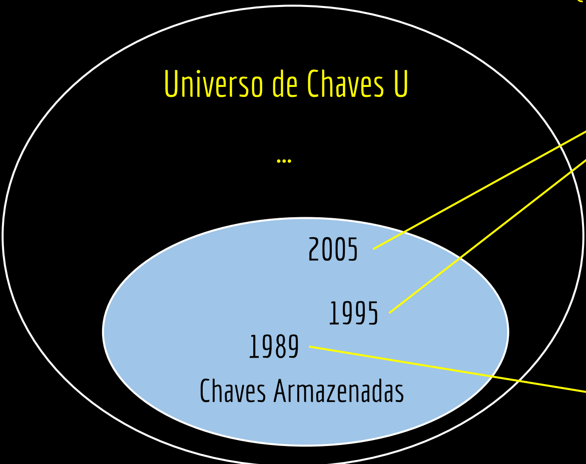
```
buscar(T,k)
     buscar por k na lista T[h(k)]

inserir(T,x)
     inserir x na cabeça de T[h(x.chave)]

excluir(T,x)
     excluir x da lista T[h(x.chave)]
```

$h(x) = x \bmod m$

T

0	NULO
1	NULO
2	NULO
3	NULO
4	NULO
h(1995) = h(2005) = 5	
6	NULO
7	NULO
8	NULO
h(1989) = 9	





Custo

```
inserir(T,x)
    inserir x na cabeça de T[h(x.chave)]
```

Custo *O(1)* no pior caso.

Custo

```
excluir(T,x)
    excluir x da lista T[h(x.chave)]
```

Custo *O(1)* no pior caso.

Assumindo que x é o nodo da lista a ser excluído, e que a lista é duplamente encadeada.

Custo

```
buscar(T,k)
    buscar por k na lista T[h(k)]
```

Se o número de elementos armazenados é proporcional ao número de slots na tabela.

Custo *O(1)* no caso médio.

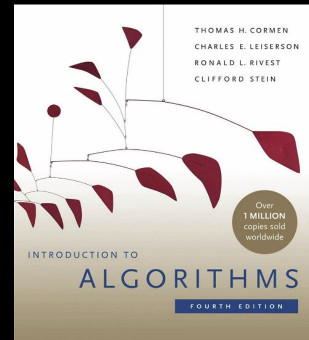
Assumindo uma função de hash independente e uniforme.

Veremos nas próximas aulas.

No pior caso temos um custo *Θ(n)*.

Ex.: todas as chaves foram parar em uma mesma posição da Tabela.

Veja a prova em Cormen et al. (2022).



Boa função de Hash

Uma boa função de hash deveria parecer aleatória.

Mapear chaves similares para locais “não necessariamente similares”

Chaves vizinhas não necessariamente serão vizinhas em T.

Com isso:

- Em uma Tabela Hash, aumentamos a sua eficiência, diminuindo a chance de colisões.

  A função **espalha** os dados.
- Isso é mandatório em uma função de Hash Criptográfico.
  - Utilizado, por exemplo, para armazenar senhas.
  - Aprenda na disciplina de Segurança.

Atenção: uma função de hash precisa ser determinística.

Fator de carga

Data uma Tabela Hash com T[0:m-1] (ou seja, com *m* slots), e *n* elementos armazenados, o fator de carga é:

alpha $\alpha = n/m$

O fator de carga dá o número médio de elementos armazenados em cada lista encadeada.

Dada uma "boa função de hash".

Usos

Em C++, um `unordered_map` é uma Tabela Hash.

```
std::unordered_map<std::string, int> dicionario;
dicionario["Joao"] = 20;
```

Usos

Em C++, um `unordered_map` é uma Tabela Hash.

```
std::unordered_map<std::string, int> dicionario;
dicionario["Joao"] = 20;
```

Em Java, um HashMap é uma Tabela Hash.

```
Map<String, Integer> mapa = new HashMap<String, Integer>();
mapa.put("Joao", 20);
```

Usos

Em C++, um `unordered_map` é uma Tabela Hash.

```
std::unordered_map<std::string, int> dicionario;
dicionario["Joao"] = 20;
```

Em Java, um HashMap é uma Tabela Hash.

```
Map<String, Integer> mapa = new HashMap<String, Integer>();
mapa.put("Joao", 20);
```

Os dicionários em Python são Tabelas Hash.

```
dicionario = {}
dicionario["Joao"] = 20
```

Um exemplo em C++

```
#include <iostream>
#include <unordered_map>
#include <string>

class MinhaFuncaoHash {
    public:
       std::size_t operator()(const unsigned long p) const {
           return p%10; //usa o último dígito como valor de hash
       }
};

int main(){
   //mapear cpfs de pessoas com seus nomes
   std::unordered_map<unsigned long, std::string> mapa;//usa o hash padrão
   //troque a linha acima pela de baixo, onde é usada a função de hash customizada
   //std::unordered_map<unsigned long, std::string, MinhaFuncaoHash> mapa;//usa o hash padrão

   mapa[11111111111] = "João da Silva";
   mapa[12121212121] = "Maria da Silva";
   mapa[33333333333] = "Pedro";
   mapa[15478965211] = "Ana";
   mapa[89657412361] = "José";
   mapa[87412369851] = "Carla";

   std::size_t numeroBuckets{mapa.bucket_count()};
   std::cout << "Mapa com " << numeroBuckets << " slots.\n";
   for (std::size_t i{0}; i < numeroBuckets; ++i) {
       std::cout << "Lista do slot " << i << " com "
            << mapa.bucket_size(i) << " elementos.\n";
   }

   return 0;
}
```

Na Unreal Engine 5

As versões “vanilla” de Tabelas Hash em C++ são criadas 1.383 vezes!

std::unordered_map -> 1.320 referências.

```
grep -rin "std::unordered_map" --include=*.cpp --include=*.hpp --include=*.h --include=*.c --include=*.cc . | wc -l
```

unordered_multimap -> 63 referências.

```
grep -rin "std::unordered_multimap" --include=*.cpp --include=*.hpp --include=*.h --include=*.c --include=*.cc . | wc -l
```

Existem ainda versões customizadas, ou ainda versões declaradas de forma que o grep acima não conseguiu pegar.

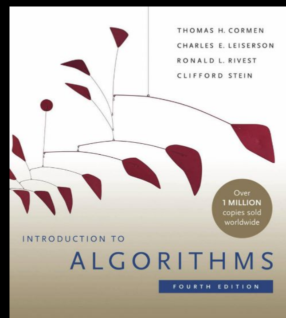
Dados da Unreal 5.1.1

Exercícios

1. Crie um Tabela Hash em C usando como função de hash $h(x) = x \bmod m$, onde m é o tamanho da tabela. A tabela deve mapear unsigned longs (que representarão CPFs) para ponteiros de struct Pessoa.
   a. Primeiro considere uma tabela sem lista encadeada, onde caso ocorra uma colisão, mostre uma mensagem de erro e não insira o item.
   b. Depois, considere o encadeamento para resolver colisões.
   c. Faça testes, e veja qual é a maior lista presente na sua implementação. Qual o fator de carga? Quantos slots permanecem vazios depois de algumas inserções? ...

Referências

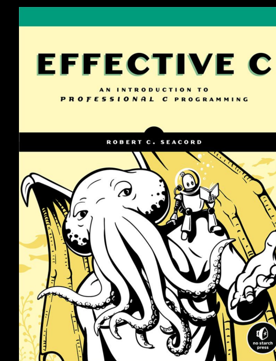
T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e Prática. 4a ed. 2022.



R. Sedgewick, K. Wayne. Algorithms Part I. 4a ed. 2011.

Estrutura de Dados e Algoritmos em C++. A. Drozdek. 4a ed. 2016.

Seacord, R. C. Effective C: An ntroduction to Professional C Programming. 2020.



Licença

Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional.