



## TABELA HASH

Um dos problemas mais elementares presentes na ciência da computação é o da busca de elementos. Em diversos cenários, a tarefa de encontrar um elemento específico numa lista é recorrente e o custo de realizá-la torna-se relevante, especialmente num escopo onde a coleção de itens possui um tamanho elevado. Neste sentido, algumas estruturas de dados se destacam por possuírem um baixo custo computacional para a operação de busca de elementos, dentre as quais pode-se destacar a Tabela Hash.

A Tabela Hash, também conhecida como tabela de dispersão ou tabela de espalhamento, é uma estrutura de dados que, por intermédio de uma função especial, associa o valor absoluto de uma chave a um endereço da tabela. Assim, pode-se definir a Tabela Hash como um array cuja indexação é feita a partir de uma função.

A função hash é uma função que possui a saída de tamanho fixo independente do tamanho de entrada, é eficiente a ponto de não ser relevante na velocidade de processamento e é determinística, ou seja, sempre que houver um mesmo valor de entrada também haverá um mesmo valor de saída. Um exemplo de função hash seria simplesmente somar a posição no alfabeto de cada caractere da palavra, obtendo uma representação numérica do elemento. Essa função cumpre o requisito de ser determinística, ou seja, sempre que uma mesma palavra for usada na função ela retornará o mesmo resultado, de retornar em tempo constante  $O(1)$  e uniforme.

Associando chaves e valores por meio de uma função escolhida, a Tabela Hash forma um vetor de registros. Neste vetor, a busca de um elemento específico não se dá percorrendo todo o vetor, mas aplica-se a função hash escolhida no elemento desejado e o retorno desta função será a posição do elemento na coleção caso não haja colisões (quando há um choque de elementos na mesma posição).

A fim de evitar colisões, cenários onde diferentes chaves têm o mesmo hash, é possível utilizar algumas estratégias como:

- Resolução de colisões por encadeamento: utilização de listas armazenando os objetos.
- Resolução de colisões por endereçamento aberto: implementada buscando outro índice livre da tabela para armazenar o elemento quando houver colisões.

Para este estudo, utilizamos a técnica de tratamento de colisões por encadeamento, e as funções podem ser observadas a seguir:

## FUNÇÕES HASH UTILIZADAS

- **Função 1:** A primeira função itera pelos caracteres da entrada, pegando seu código Unicode e, se esse código for par, o resultado final é multiplicado por esse código elevado a ele mesmo. Se o código for ímpar, então o resultado final é somado em uma unidade e depois multiplicado pelo código.

```
def f1(text:str, size:int):  
    final = 0  
    for i in range(1, len(text)):  
        if ord(text[i]) % 2 == 0:  
            final *= ord(text[i]) ** ord(text[i])  
        else:  
            final += 1  
            final *= ord(text[i])  
    return final % size
```

- **Função 2:** A segunda função é chamada de Hash de rolamento polinomial. Ela é descrita pela fórmula:

$$\text{Hash}(s) = \sum_{i=0}^{n-1} s[i] \cdot p^i \bmod(m)$$

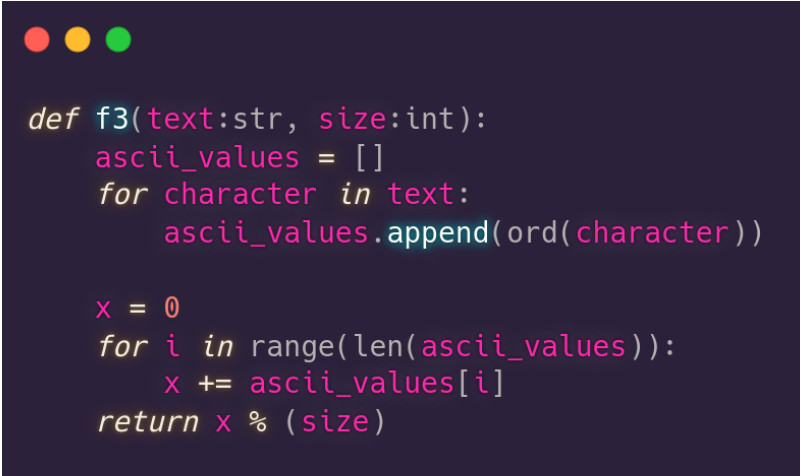
Onde 's' é o valor de entrada, 's[i]' é o valor ASCII do i-ésimo caractere, 'n' é o tamanho da entrada, 'm' é o tamanho da tabela e 'p' é um valor inteiro positivo, normalmente um número primo.

```
def f2(text:str, size:int):  
    p = 31  
    final = 0  
    for i in range(len(text)):  
        final += (ord(text[i]) * (p ** i)) % size  
    return final % size
```

- **Função 3:** A terceira função utiliza os próprios valores ASCII dos caracteres, então para cada palavra, soma-se o valores ASCII obtidos e aplica-se a função de módulo ao resultado, utilizando o tamanho fornecido. A fórmula é a seguinte:

$$Hash(s) = \sum_{i=0}^{n-1} s[i] \mod(m)$$

Onde 's' é o valor de entrada, 's[i]' é o valor ASCII do i-ésimo caractere, 'n' é o tamanho da entrada, 'm' é o tamanho da tabela e 'p' é um valor inteiro positivo, normalmente um número primo.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Python and defines a function named f3. It takes two arguments: 'text' of type 'str' and 'size' of type 'int'. Inside the function, it creates an empty list 'ascii\_values'. It then iterates over each 'character' in 'text' and appends its ASCII value (obtained via 'ord(character)') to the 'ascii\_values' list. After the loop, it initializes a variable 'x' to 0. It then iterates over the indices of 'ascii\_values' (from 0 to len(ascii\_values)-1) and adds each value to 'x'. Finally, it returns the result of 'x % (size)'.

```
def f3(text:str, size:int):  
    ascii_values = []  
    for character in text:  
        ascii_values.append(ord(character))  
  
    x = 0  
    for i in range(len(ascii_values)):  
        x += ascii_values[i]  
    return x % (size)
```

- **Função 4:** A quarta função é bem semelhante à última função. A diferença entre as duas está na soma dos valores ASCII, onde antes de apenas realizar a soma, eleva-se o número a quarta. A fórmula é a seguinte:

$$Hash(s) = \sum_{i=0}^{n-1} s[i]^4 \mod(m)$$

Onde 's' é o valor de entrada, 's[i]' é o valor ASCII do i-ésimo caractere, 'n' é o tamanho da entrada, 'm' é o tamanho da tabela e 'p' é um valor inteiro positivo, normalmente um número primo.

```
def f4(text:str, size:int):
    ascii_values = []
    for character in text:
        ascii_values.append(ord(character))

    x = 0
    for i in range(len(ascii_values)):
        x += ascii_values[i] ** 4
    return x % (size)
```

- **Função 5:** Essa função faz um scan na palavra de entrada e faz a diferença dos valores ASCII entre o caráter em questão com o último. Após isso eleva-se essa diferença a 4 e no final retira-se o módulo da soma final. Ela é descrita pela fórmula:

$$Hash(s) = \sum_{i=1}^{n-1} (s[i] - s[i-1])^4 \mod(m)$$

Onde 's' é o valor de entrada, 's[i]' é o valor ASCII do i-ésimo termo, 'n' é o tamanho do valor de entrada e 'm' é o tamanho da tabela.

```
def f5(text:str, size:int):
    final = 0
    for i in range(1, len(text)):
        final += (ord(text[i]) - ord(text[i-1]))**4
    return final%size
```

## DESCRIÇÃO DA FERRAMENTA

A ferramenta foi implementada em Python, usando Programação Orientada a Objetos. O repositório final conta com 2 arquivos importantes: o **main.py** que consta uma aplicação para um usuário onde ele pode escolher entre as funções desejadas, e o **analysis.py** onde foram feitos os testes para a análise crítica presente mais adiante no texto.

Primeiro foi implementada uma classe para as Listas Encadeadas, cujo código reaproveitamos de uma referência online [2]. Em seguida, escrevemos o código da tabela Hash, que é uma classe com os atributos *size*, *slots*, *meantime\_get*, *meantime\_put* e *col\_cont*.

Os atributos citados representam respectivamente o tamanho da tabela, uma referência para cada posição da tabela (cada slot comporta uma Lista Encadeada), uma tupla com o tempo médio de busca e o número de buscas feitas em uma tabela, o tempo médio de cálculo de hash, e o número de colisões. A classe que implementa a tabela de Hash possui ainda os seguintes métodos: *print\_table*, *take\_f*, *put* e *get*, que servem para imprimir a tabela, atribuir uma função de hash à tabela, inserir dados e buscar dados, respectivamente.

Foram implementados também os códigos das 5 funções que utilizamos. Essas funções foram guardadas num arquivo chamado **fun.py**.

Para o cálculo efetivo do hash de cada chave, consideramos que como há palavras que possuem mesma grafia mas atributos diferentes, a melhor estratégia é passar cada linha do banco de dados como uma tupla, armazenar esses valores na tabela de hash, e na hora de realizar a busca, retornar todas as aparições dessa mesma palavra na lista em que ela está, uma vez que todas as versões de uma palavra com mesma grafia terão o mesmo hash.

Isso acabou colocando uma restrição na nossa implementação da Lista Encadeada, onde agora só podemos passar dados no formato de lista. Isso foi necessário para lidar com a restrição de tipos que precisávamos para implementar o método *get* das Listas.

Também implementamos um método *strong\_get* na lista, onde se passa uma lista de inteiros, que representam posições na lista, e tem o retorno de quais itens ocupam tais posições. Isso nos permitiu que quando o usuário usa o método *get* da tabela, ele recebe tanto a posição da lista encadeada onde está a chave a ser buscada, quanto os atributos que acompanhavam essa chave no banco de dados original.

A partir daí fomos testar as funções e ver quais as vantagens e desvantagens de cada uma.

## ANÁLISE DOS RESULTADOS

Para a análise dos resultados, foram levados em consideração 3 fatores: o número de colisões, o tempo médio de inserção de um item e o tempo médio de busca de um item. Para cada função obtida, esses fatores foram analisados e comparados.

O tempo de *get* foi calculado automaticamente na inserção de itens na tabela, assim como o número de colisões. Porém, para calcular o tempo médio de busca, precisávamos que a busca ocorresse. Para isso, sorteamos aleatoriamente 100 palavras do banco de dados original e realizamos uma busca dessas palavras em 5 tabelas, cada uma com uma função diferente. Nossos resultados foram os seguintes:

	function	collisions	meantime_put	meantime_get
0	f1	11943	0.000006	0.000008
1	f2	9795	0.000003	0.000004
2	f3	30574	0.000004	0.000025
3	f4	10343	0.000006	0.000005
4	f5	11665	0.000003	0.000004

Plotamos ainda alguns gráficos para comparar como cada função desempenhou em relação a cada atributo.

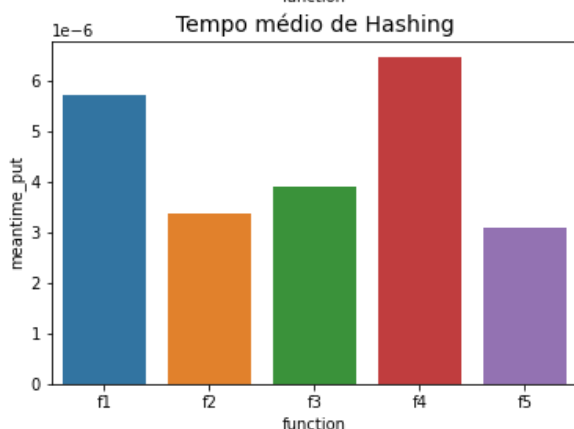
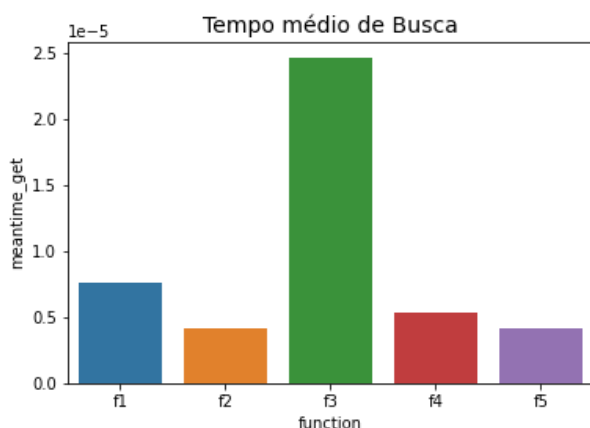
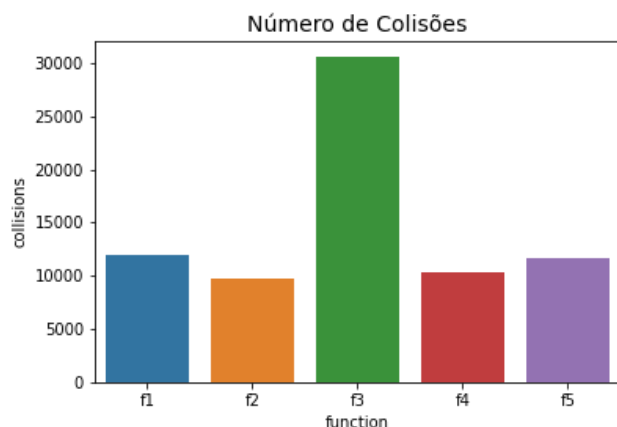
A função com menor número de colisões é a segunda, enquanto a com maior número

de colisões é a terceira. Pode-se observar que o número de colisões reflete no tempo médio de busca também. Uma vez que quando se tem muitas colisões, é mais fácil precisar percorrer uma lista para encontrar o item que está buscando. Notamos ainda que a quinta função teve o menor tempo de Hashing, o que significa que é mais rápido para essa função inserir dados na tabela. Outro fato curioso é que as funções 1 e 4 possuem um tempo de Hashing alto. Isso acontece porque tentamos 'brincar' com a implementação delas até que seus respectivos números de colisões fossem reduzidos. O custo dessa brincadeira foi ter funções menos eficientes pois elas requerem muito poder computacional para fazer contas que espalhem melhor as chaves.

## CONCLUSÃO

Levando em consideração a análise realizada, concluímos o trabalho com a constatação de que a f2 foi a melhor função desenvolvida, tendo obtido o menor número de colisões, o menor tempo médio de busca e o segundo menor tempo médio de hashing, logo, a função mais otimizada dentre as 5 desenvolvidas.

Em paralelo, a f3, com o maior número de colisões, o maior tempo médio de busca e o terceiro maior tempo médio de hashing, foi considerada a pior função dentre as propostas.



## REFERÊNCIAS

[1] BRUNET, João. Tabelas Hash. Estruturas de Dados e Algoritmos, 2019. Disponível em: <<https://joaoarthurbm.github.io/eda/posts/hashtable/#:~:text=Tabela%20hash%20%C3%A9%20uma%20estrutura,em%20tempo%20constante%20e%20uniforme>>. Acesso em: 13 de setembro, 2022.

[2] How to create a Linked List in Python. Educative Answers Team. Disponível em: <<https://www.educative.io/answers/how-to-create-a-linked-list-in-python>>. Acesso em: 15 de setembro, 2022.

[3] FEIJLOFF, Paulo. Hashing. IME-USP, 2019. Disponível em: <<https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html#:~:text=Outro%20exemplo%3A%20fun%C3%A7%C3%A3o%20de%20hashing,todos%20os%20d%C3%ADgitos%20do%20CPF>> . Acesso em: 16 de setembro, 2022.