Dicionários

PrepTech Google

Dicionários

Estruturas de dados que armazenam pares de chave-valor, onde cada chave é única e mapeia diretamente para um valor associado.

```
data= {
    "name": "Ana",
    "age" : 25
}
```

Dicionários

Estruturas de dados que armazenam pares de chave-valor, onde cada chave é única e mapeia diretamente para um valor associado.

```
data= {
    "name": "Ana",
    "age" : 25
}
```

"name" e "age" são chaves, enquanto "Ana" e 25 são os valores correspondentes.

Dicionários x Listas/Arrays

Listas/arrays permitem armazenar uma sequência de itens indexados numericamente, enquanto dicionários, em geral, permitem o uso de qualquer tipo imutável (como strings, inteiros, etc.) como índice.

Dicionários

Aplicações práticas

Lookup Tables
Usados para armazenar e recuperar informações rapidamente, como configurações de sistemas ou dados de referência.

Contagem de Frequência
Contagem de ocorrências de itens, como palavras em um texto (implementação de histogramas).

Grafos e Conjuntos
Representação de grafos (vértices e arestas) e operações em conjuntos (verificação de membros).

Caches
Implementação de caches para armazenar resultados de cálculos ou dados frequentemente acessados.

Tabelas de Símbolos
Em compiladores, dicionários são usados para armazenar variáveis e suas localizações de memória

Dicionários

Operações

Inserção
Adicionar um par chave-valor no dicionário.

Busca
Encontrar um valor a partir de uma chave.

Remoção
Remover um par chave-valor do dicionário.

Essas operações geralmente são realizadas em tempo médio constante `O(1)`, o que faz do dicionário uma estrutura de dados eficiente para acesso rápido.

Dicionários

```
class Dictionary:
    def __init__(self): #constructor

    def insert(self, key, value):

    def get(self, key):

    def remove(key):
```

Direct Access Table (DAT)

Dicionários

Direct-Address Tables

Chaves mapeiam diretamente para índices em uma tabela

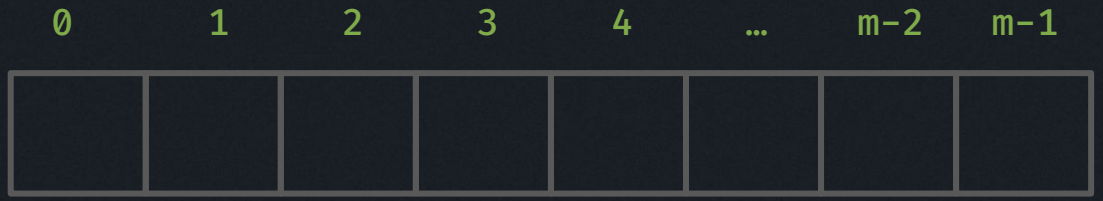
Acesso Direto
Como as chaves são usadas diretamente como índices, as operações de busca, inserção e remoção **sempre** têm complexidade constante O(1).

Pré-requisito
A chave deve ser um número inteiro (ou ser convertível para um inteiro), com um valor dentro de um intervalo conhecido e limitado.

Dicionários

Direct-Address Tables

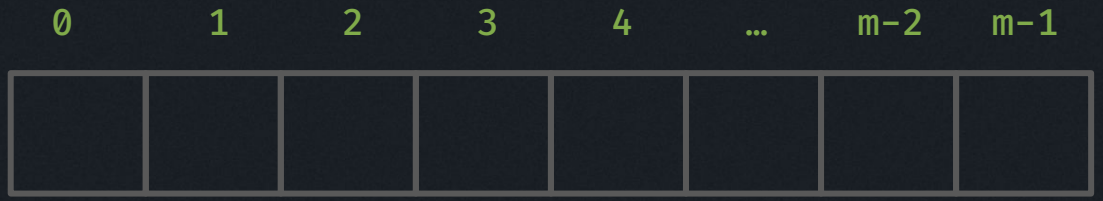
Tabela (array) de tamanho m



Dicionários

Direct-Address Tables

Tabela (array) de tamanho m

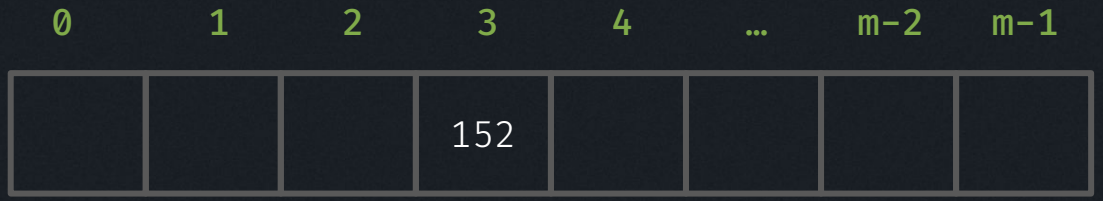



```
insert(3, 152)
```

Dicionários

Direct-Address Tables

Tabela (array) de tamanho m

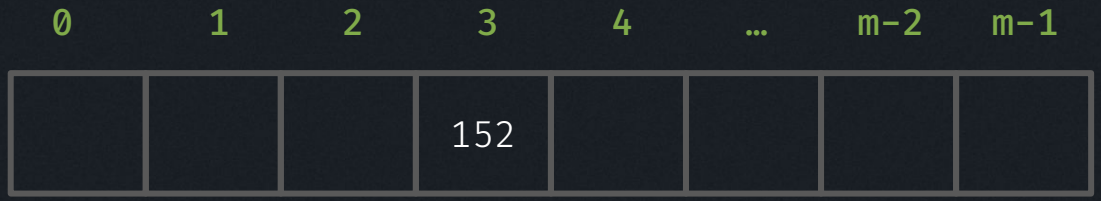



```
insert(3, 152)
```

Dicionários

Direct-Address Tables

Tabela (array) de tamanho m

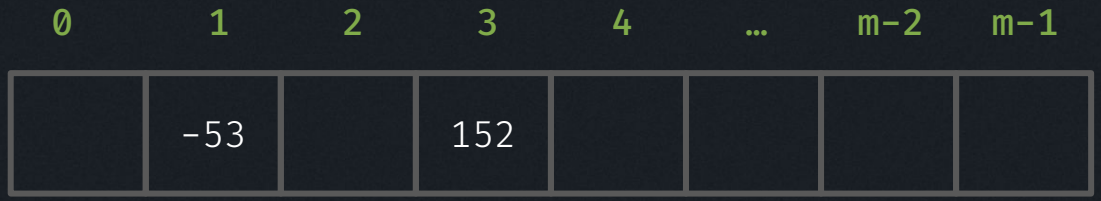



```
insert(3, 152)
insert(1, -53)
```

Dicionários

Direct-Address Tables

Tabela (array) de tamanho m

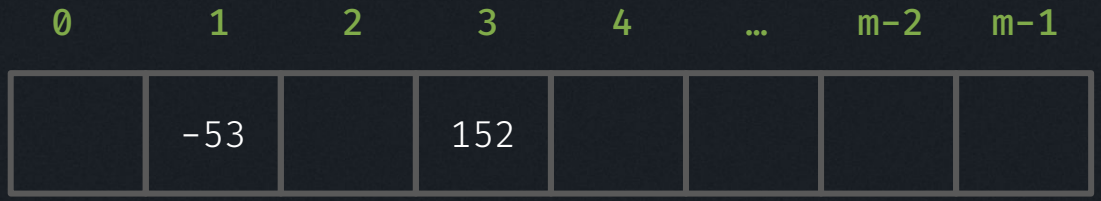



```
insert(3, 152)
insert(1, -53)
```

Dicionários

Direct-Address Tables

Tabela (array) de tamanho m

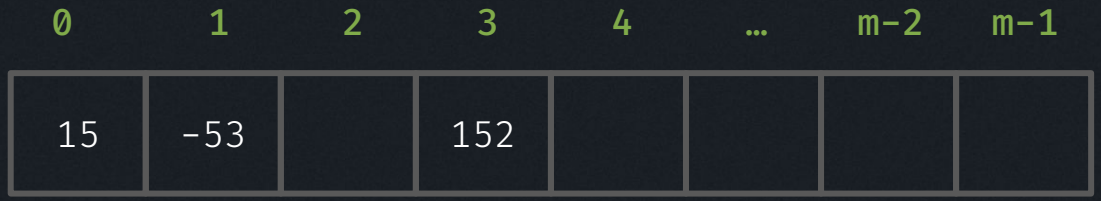



```
insert(3, 152)
insert(1, -53)
insert(0, 15)
```

Dicionários

Direct-Address Tables

Tabela (array) de tamanho m




```
insert(3, 152)
insert(1, -53)
insert(0, 15)
```

Dicionários

Direct-Address Tables

```
class DirectAddressTable:
    def __init__(self, size):
        self.table = [None] * size

    def insert(self, key, value):
        self.table[key] = value

    def get(self, key):
        return self.table[key]

    def remove(self, key):
        self.table[key] = None
```

Dicionários

Direct-Address Tables

Vantagens

Facilidade de implementação

Todas as operações são implementadas em tempo constante **O(1)** sem distinção de melhor, pior ou caso médio.

Desvantagens

Ineficiente em termos de espaço se o universo de chaves possíveis for muito grande comparado ao número de chaves armazenadas

Apenas prático quando se pode garantir que as chaves estarão dentro de um intervalo restrito.

Hash Tables

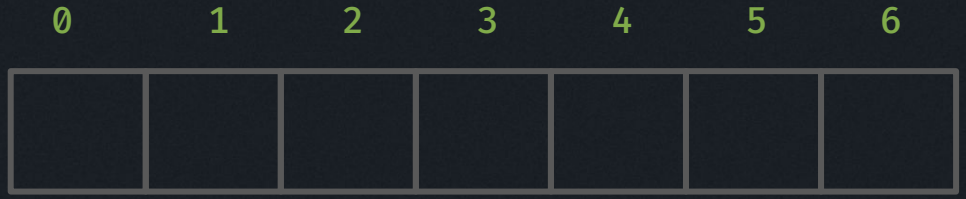
Hash Tables (Tabelas Hash) são estruturas de dados que mapeiam chaves a valores usando uma função hash para calcular um índice na tabela onde o valor será armazenado.

Função Hash: Uma função hash converte a chave em um índice na tabela, distribuindo as chaves **uniformemente** pelo espaço disponível.

Dicionários

Hash Tables

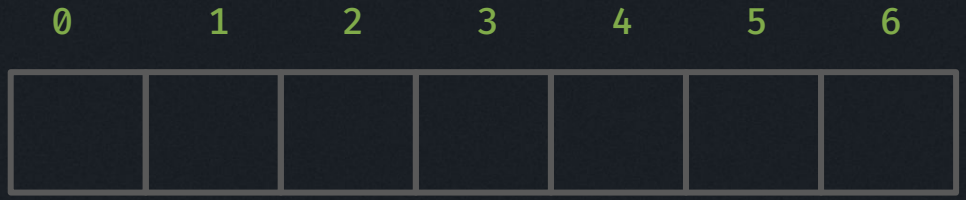
Tabela (array) de tamanho 7
Função hash: H(x) = x % 7



Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7

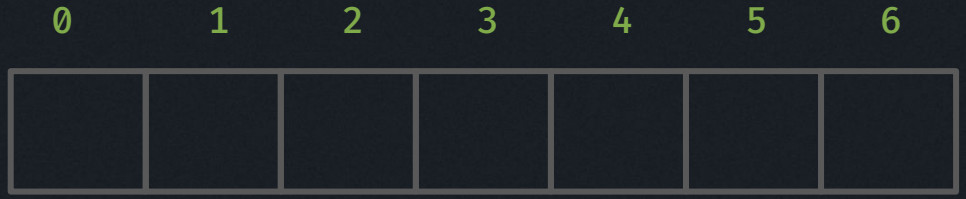



```
insert(3, 152)
```

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7




```
insert(3, 152)
```

H(3) = 3 % 7 = 3

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(3, 152)
```

H(3) = 3 % 7 = 3

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(9, -58)
```

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(9, -58)
```

H(9) = 9 % 7 = 2

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(9, -58)
```

H(9) = 9 % 7 = 2

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(1000, 75)
```

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(1000, 75)
```

H(1000) = 1000 % 7 = 6

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(1000, 75)
```

H(1000) = 1000 % 7 = 6

Dicionários

Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(10, 4)
```

Dicionários

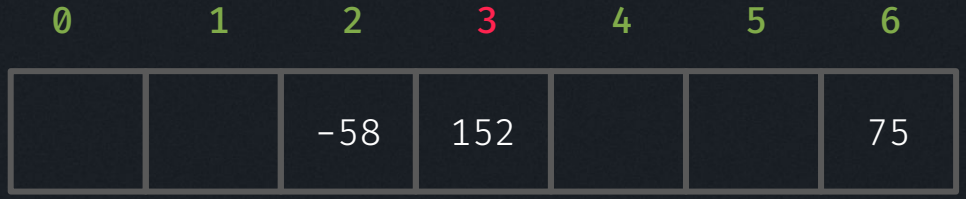
Hash Tables

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7


```
insert(10, 4)
```

H(1000) = 10 % 7 = 3

Tabela (array) de tamanho 7
Função hash: H(x) = x % 7



Colisões acontecem quando chaves diferentes (3 e 10 nesse exemplo) geram o mesmo índice na tabela

```
insert(10, 4)
```

H(1000) = 10 % 7 = 3

A função hash deve distribuir as chaves de forma uniforme para minimizar colisões. Elas devem ter as seguintes propriedades:

Determinismo: A mesma chave sempre gera o mesmo hash code.

Uniformidade: Hash codes devem ser distribuídos de maneira uniforme pelo espaço de índices.

Eficiência: A função hash deve ser rápida de calcular.

Exemplos de funções hash:

- **Hashing Simples:** hash(key) % size
- **Multiplicative Hashing:** floor(size * (key * A % 1)), onde A é uma constante fracionária.

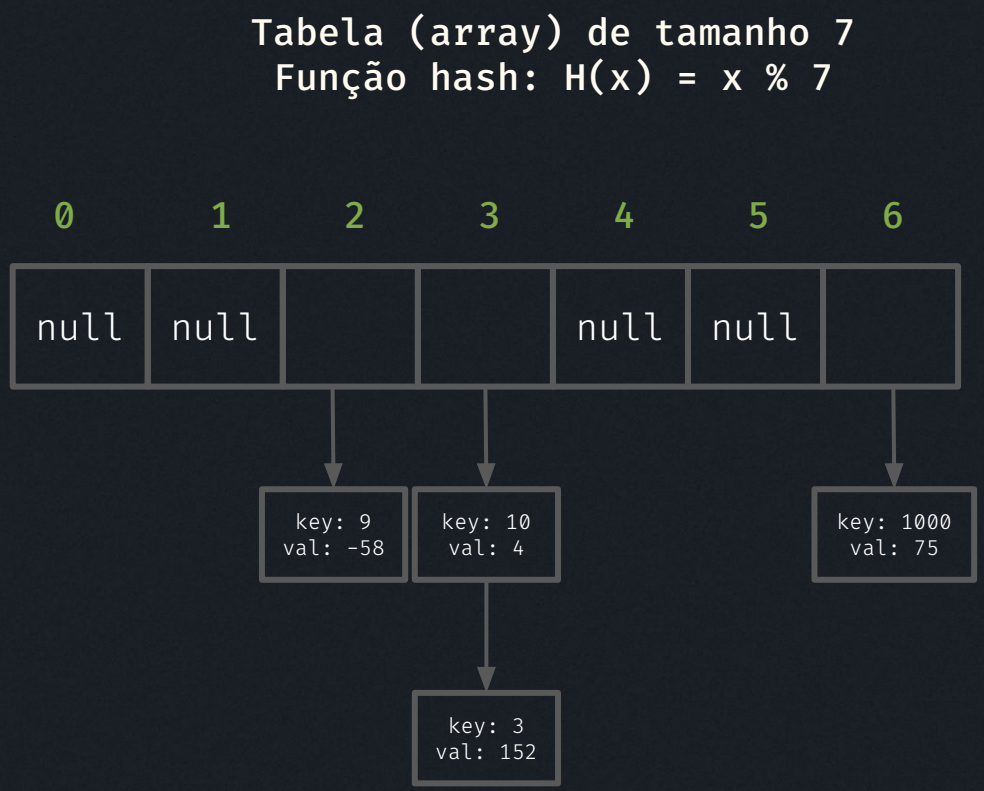
Por definição, colisões são **inevitáveis**, pois tabelas hash permitem mapear um domínio potencialmente infinito de chaves em uma tabela de tamanho finito.

Alguns métodos de tratamento de colisão:

- **Chaining**
- Open Addressing
- Duplo Hashing

Dicionários

Chaining usando Linked List



Dicionários

Chaining usando Linked List

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None


class HashTable:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key):

    def get(self, key):

    def remove(self, key):
```

Dicionários

Chaining usando Linked List

```
def insert(self, key, value):
    index = self._hash(key)
    new_node = Node(key, value)

    # Inserção no início da lista encadeada
    new_node.next = self.table[index]
    self.table[index] = new_node
```

Dicionários

Chaining usando Linked List

```
def get(self, key):
    index = self._hash(key)
    current = self.table[index]

    while current:
        if current.key == key:
            return current.value
        current = current.next

    return None
```

Dicionários

Chaining usando Linked List

```
    def remove(self, key):
        index = self._hash(key)
        current = self.table[index]
        prev = None

        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.table[index] = current.next
                return True
            prev = current
            current = current.next

        return False
```

Dicionários

Chaining usando Linked List - Complexidade (melhor e pior caso)

Inserção

Se sempre inserirmos no início da lista, a inserção sempre executará em tempo constante **O(1)** independente de haver ou não colisão

Busca

Pior caso: todas as chaves colidem no mesmo índice, então precisamos percorrer todos os elementos da tabela resultando em uma busca **O(n)**

Melhor caso: não há colisões, então todas as listas encadeadas terão tamanho 0 ou 1, resultando em uma busca **O(1)**

Remoção

A remoção envolve buscar o elemento a ser removido e retirá-lo da tabela. A análise de melhor e pior caso segue o mesmo racional da busca

A análise do **caso médio** assume que a função hash distribui as chaves uniformemente, ou seja, cada chave tem a mesma probabilidade de ser mapeada para qualquer índice.

Com essa suposição, o número esperado de elementos por índice é n/m, onde n é o número de chaves armazenadas e m é o tamanho da tabela.

Fator de Carga (α): O fator de carga é definido como α = n/m, que representa o número médio de elementos por índice.

Caso Médio da Busca: Com uma distribuição uniforme, a complexidade média das operações de busca é O(1 + α)

- O(1) para computar a função hash
- O(α) para fazer a busca

Dicionários

Chaining usando Linked List - Complexidade (caso médio)

Se o número de elementos inseridos for proporcional ao tamanho da tabela, então n será menor ou próximo de m. Por simplificação, vamos considerar $n \equiv m$. Nesse caso, $\alpha = n/m = m/m = 1$, resultando em uma busca em tempo constante **O(1)** no caso médio.

IMPORTANTE: Essa análise só é válida para casos de funções hash que sejam uniformes (chaves distintas colidem com probabilidade 1/m).

Open Addressing é uma estratégia de tratamento de colisão em que todos os elementos são armazenados diretamente dentro da tabela hash, sem listas encadeadas ou buckets adicionais.

Quando ocorre uma colisão, o algoritmo busca outro índice na tabela onde o elemento pode ser inserido. Possíveis estratégias: **Linear Probing, Quadratic Probing, Double Hashing**

Linear Probing é a técnica mais simples de open addressing. Quando ocorre uma colisão, o algoritmo tenta inserir o elemento na próxima posição disponível, percorrendo a tabela em passos lineares

Clustering: Uma das desvantagens do linear probing é o clustering. Quando várias chaves colidem, elas podem criar clusters de elementos, o que aumenta o tempo de busca e inserção.

Exemplo: Se a chave K1 mapeia para o índice 2, mas o índice 2 já está ocupado, o linear probing verificará o índice 3, depois o 4, e assim por diante, até encontrar um espaço vazio.

Quadratic Probing é uma variação do linear probing que evita clusters de colisões, verificando posições em saltos que aumentam quadraticamente.

Se o índice original para uma chave é i, o próximo índice a ser tentado é (i + 1^2) % m, depois (i + 2^2) % m, e assim por diante.

Vantagens: Reduz o problema de clustering primário (mas não elimina completamente o clustering secundário).

Problemas: Não garante que todos os índices serão visitados, o que pode dificultar a inserção se a tabela estiver quase cheia.

Double Hashing usa uma segunda função hash para determinar o passo de deslocamento (ou "salto") ao buscar um novo índice quando ocorre uma colisão.

Fórmula: Se o índice original para uma chave é i, o próximo índice é determinado por i + j * h2(key), onde h2(key) é a segunda função hash e j é o número de colisões ocorridas.

Vantagens: Double hashing minimiza os problemas de clustering primário e secundário, proporcionando uma distribuição mais uniforme.

Problemas: Requer o design de duas funções hash eficientes e independentes, o que pode ser desafiador.

A implementação de qualquer uma das estratégias de Open Addressing requer o uso de estruturas adicionais para gerenciar “buracos” na tabela, tornando a implementação mais complexa do que técnicas como Chaining.

Roman To Integer

https://leetcode.com/problems/roman-to-integer/

13. Roman to Integer

Solved

Easy | Topics | Companies | Hint

Roman numerals are represented by seven different symbols: `I`, `V`, `X`, `L`, `C`, `D` and `M`.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, `2` is written as `II` in Roman numeral, just two ones added together. `12` is written as `XII`, which is simply `X + II`. The number `27` is written as `XXVII`, which is `XX + V + II`.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not `IIII`. Instead, the number four is written as `IV`. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as `IX`. There are six instances where subtraction is used:

- `I` can be placed before `V` (5) and `X` (10) to make 4 and 9.
- `X` can be placed before `L` (50) and `C` (100) to make 40 and 90.
- `C` can be placed before `D` (500) and `M` (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2:

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.
```

Example 3:

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Roman To Integer

https://leetcode.com/problems/roman-to-integer/

```python
def romanToInt(self, s: str) -> int:
    table = {
        "I": 1,
        "V": 5,
        "X": 10,
        "L": 50,
        "C": 100,
        "D": 500,
        "M": 1000
    }

    skip_next = False
    output = 0
    for i in range(0, len(s)):
        if skip_next:
            skip_next = False
            continue

        c = s[i]
        value = table[c]

        if i + 1 < len(s):
            next_c = s[i + 1]
            next_value = table[next_c]

            if next_value > value:
                value = next_value - value
                skip_next = True

        output += value

    return output
```

Repeated DNA Sequences

https://leetcode.com/problems/repeated-dna-sequences

187. Repeated DNA Sequences

Solved

Medium | Topics | Companies

The **DNA sequence** is composed of a series of nucleotides abbreviated as `'A'`, `'C'`, `'G'`, and `'T'`.

- For example, `"ACGAATTCCG"` is a **DNA sequence**.

When studying **DNA**, it is useful to identify repeated sequences within the DNA.

Given a string `s` that represents a **DNA sequence**, return all the `10`**-letter-long** sequences (substrings) that occur more than once in a DNA molecule. You may return the answer in **any order**.

Example 1:

```
Input: s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT"
Output: ["AAAAACCCCC","CCCCCAAAAA"]
```

Example 2:

```
Input: s = "AAAAAAAAAAAAA"
Output: ["AAAAAAAAAA"]
```

Constraints:

- `1 <= s.length <= 10^5`
- `s[i]` is either `'A'`, `'C'`, `'G'`, or `'T'`.

ada

Repeated DNA Sequences

https://leetcode.com/problems/repeated-dna-sequences

```python
def findRepeatedDnaSequences(self, s: str) -> List[str]:
    if len(s) < 10: return []

    lookup = {}
    curr_substring = s[:10] # first 10 chars
    lookup[curr_substring] = 1

    for i in range (10, len(s)):
        c = s[i]
        curr_substring = curr_substring[1:10] + c

        if not curr_substring in lookup:
            lookup[curr_substring] = 0

        lookup[curr_substring] += 1

    output = []
    for key in lookup.keys():
        if lookup[key] > 1:
            output.append(key)

    return output
```

Longest Substring Without Repeating Characters

https://leetcode.com/problems/longest-substring-without-repeating-characters

3. Longest Substring Without Repeating Characters

Medium | Topics | Companies | Hint

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

> **Input:** s = "abcabcbb"
> **Output:** 3
> **Explanation:** The answer is "abc", with the length of 3.

Example 2:

> **Input:** s = "bbbbb"
> **Output:** 1
> **Explanation:** The answer is "b", with the length of 1.

Example 3:

> **Input:** s = "pwwkew"
> **Output:** 3
> **Explanation:** The answer is "wke", with the length of 3.
> Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

Longest Substring Without Repeating Characters

https://leetcode.com/problems/longest-substring-without-repeating-character

3. Longest Substring Without Repeating Characters

Medium | Topics | Companies | Hint

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

> **Input:** s = "abcabcbb"
> **Output:** 3
> **Explanation:** The answer is "abc", with the length of 3.

Example 2:

> **Input:** s = "bbbbb"
> **Output:** 1
> **Explanation:** The answer is "b", with the length of 1.

Example 3:

> **Input:** s = "pwwkew"
> **Output:** 3
> **Explanation:** The answer is "wke", with the length of 3.
> Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

Estratégia: iterar pela string e armazenar em um dicionário o último índice de cada caractere. Além disso, armazenamos o índice de onde começa a string atual sem repetição.

Quando encontramos um caractere que já existe o dicionário, **e o seu último índice for maior que o início da string atual**, atualizamos o início da string atual para aquela posição + 1.

Longest Substring Without Repeating Characters

https://leetcode.com/problems/longest-substring-without-repeating-character

3. Longest Substring Without Repeating Characters

Medium | Topics | Companies | Hint

Given a string `s`, find the length of the **longest substring** without repeating characters.

Example 1:

> **Input:** s = "abcabcbb"
> **Output:** 3
> **Explanation:** The answer is "abc", with the length of 3.

Example 2:

> **Input:** s = "bbbbb"
> **Output:** 1
> **Explanation:** The answer is "b", with the length of 1.

Example 3:

> **Input:** s = "pwwkew"
> **Output:** 3
> **Explanation:** The answer is "wke", with the length of 3.
> Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- `0 <= s.length <= 5 * 10^4`
- `s` consists of English letters, digits, symbols and spaces.

Longest Substring Without Repeating Characters

https://leetcode.com/problems/longest-substring-without-repeating-character

```python
def lengthOfLongestSubstring(self, s: str) -> int:
    table = dict()

    start = 0
    curr_length = 0
    ans = 0
    for i in range(0, len(s)):
        c = s[i]
        if c in table and table[c] >= start:
            ans = max(ans, curr_length)
            curr_length = i - table[c]
            start = table[c] + 1
        else:
            curr_length += 1

        table[c] = i

    ans = max(ans, curr_length)

    return ans
```

Obrigado