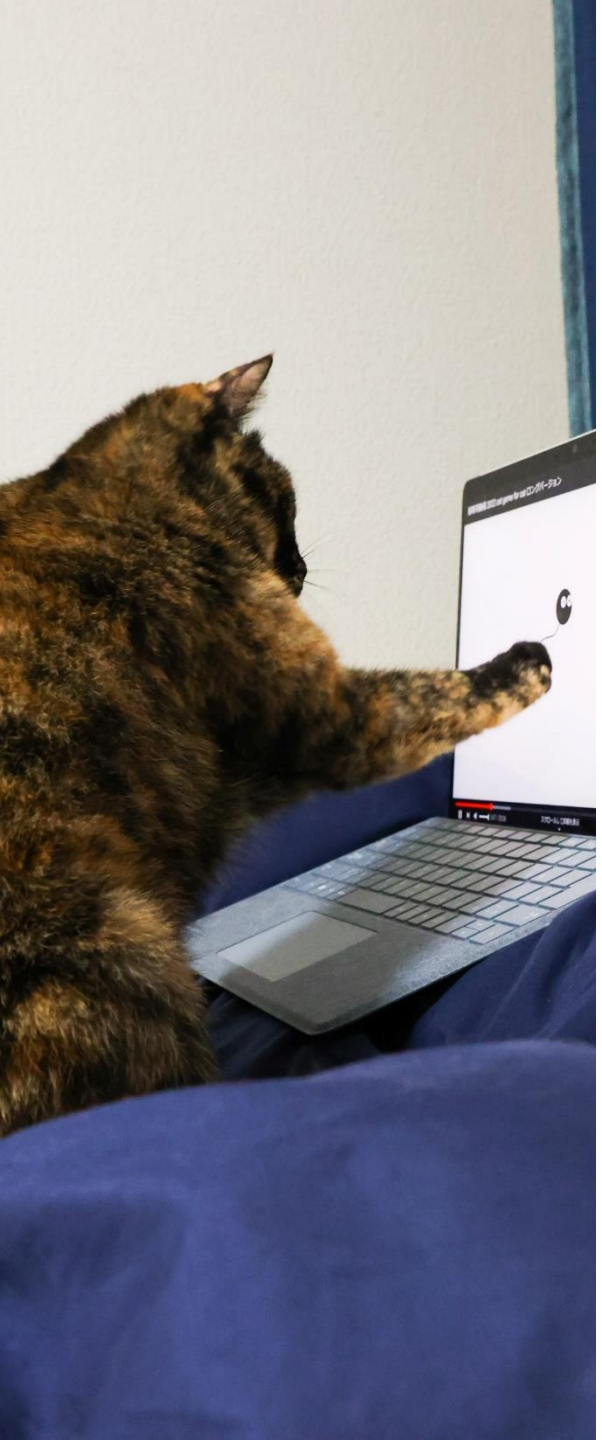
Performance em Sistemas Ciberfísicos

Aula 3 – Algoritmos de Cache

Frank Coelho de Alcantara – 2023 -1



O problema que o Cache Resolve

“Não existe esta coisa de bug de cache simples.”
Rob Pike

CACHE EM SISTEMAS COMPUTACIONAIS

1. **Desafio:** Diferença de velocidade entre CPU e RAM.

2. **Solução:** Memória cache - memória rápida e temporária.

3. **Objetivo:** Armazenar dados e instruções frequentemente utilizados.

4. **Resultado:** Aumento no desempenho e eficiência do sistema.

Sir. Maurice Wilkes - st John’s College Cambridge

Latência

Latência refere-se ao intervalo de tempo necessário para acessar e obter informações de uma localização específica de memória ou dispositivos de armazenamento.

Componente	Latência típica
Cache L1	0,5 - 2 nanossegundos
Cache L2	3 - 5 nanossegundos
Cache L3	10 - 40 nanossegundos
Memória Principal (RAM)	50 - 100 nanossegundos
HDD (Disco rígido)	5.000 - 10.000 nanossegundos (5 - 10 milissegundos)
SSD (Disco de estado sólido)	100 - 500 nanossegundos (0,1 - 0,5 milissegundos)

O Cache

- O cache é uma memória de alta velocidade, geralmente embutida no próprio processador, que armazena temporariamente dados e instruções frequentemente acessados. O sistema usa um algoritmo, como o LRU (*Least Recently Used*) ou LFU (*Least Frequently Used*), para determinar quais dados devem ser armazenados na cache.
- Quando a CPU precisa acessar um dado, ela verifica se o dado está presente na cache. Se o dado estiver presente (um "*cache hit*"), a CPU o acessa rapidamente. Caso contrário (um "*cache miss*“), a CPU precisa buscar o dado em outra memória.

Localidade

Localidade refere-se à propriedade dos programas que faz com que eles acessem regularmente um conjunto limitado de dados ou instruções em um determinado período de tempo.

Podemos dividir o conceito de localidade em em duas categorias: ***localidade espacial***, que se refere ao acesso repetido aos dados próximos em um espaço de endereço contíguo, e ***localidade temporal***, que se refere ao acesso repetido aos mesmos dados em diferentes momentos no tempo.

Código Eficiente Aproveita a Localidade

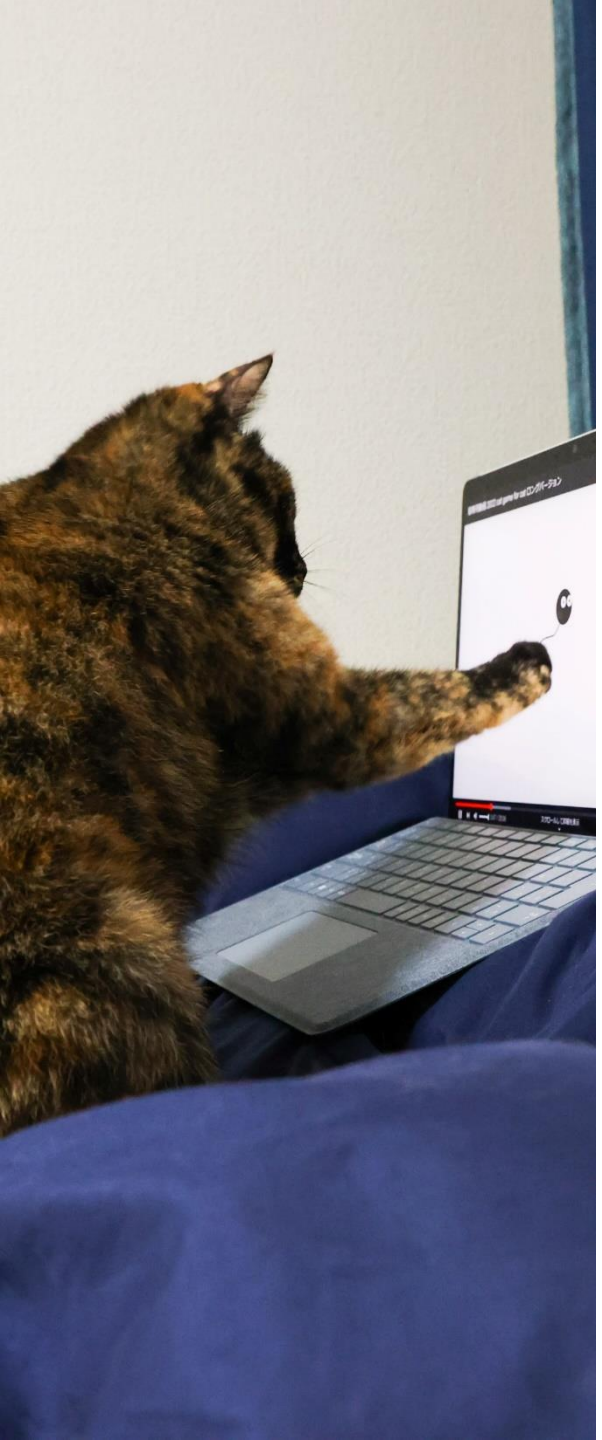
1. **Aproveite a localidade espacial**: tente agrupar as variáveis que são acessadas juntas em estruturas de dados adjacentes na memória. Dessa forma, quando você acessar uma variável, as outras estarão próximas na memória e serão acessadas mais rapidamente.
2. **Aproveite a localidade temporal**: tente evitar escrever código que faça com que você acesse repetidamente os mesmos dados em locais distantes da memória. Em vez disso, armazene os dados em variáveis locais ou em cache para reduzir o tempo de acesso.

Código Eficiente Aproveita a Localidade

3. **Reduza a complexidade do código**: quanto mais simples o código, mais fácil é para o compilador otimizá-lo e encontrar padrões de localidade. Evite criar estruturas de dados complexas ou usar muitas chamadas de função aninhadas que possam dificultar a análise de padrões de acesso.
4. **Otimize loops**: os loops são uma fonte comum de ineficiência em relação à localidade. Tente escrever loops que acessam dados próximos na memória e evite loops aninhados que podem causar saltos frequentes na memória.

Código Eficiente Aproveita a Localidade

5. **Evite acessos aleatórios na memória**: o acesso aleatório na memória é muito mais lento do que o acesso sequencial. Tente escrever código que acesse a memória de maneira sequencial ou pelo menos agrupe acessos relacionados próximos uns aos outros.
6. **Use algoritmos eficientes**: alguns algoritmos são mais propensos a ter boa localidade do que outros. Tente usar algoritmos que minimizem o acesso aleatório à memória, como os que usam matrizes de acesso sequencial.



Algoritmos de Cache

Algoritmos de substituição no Cache - LRU

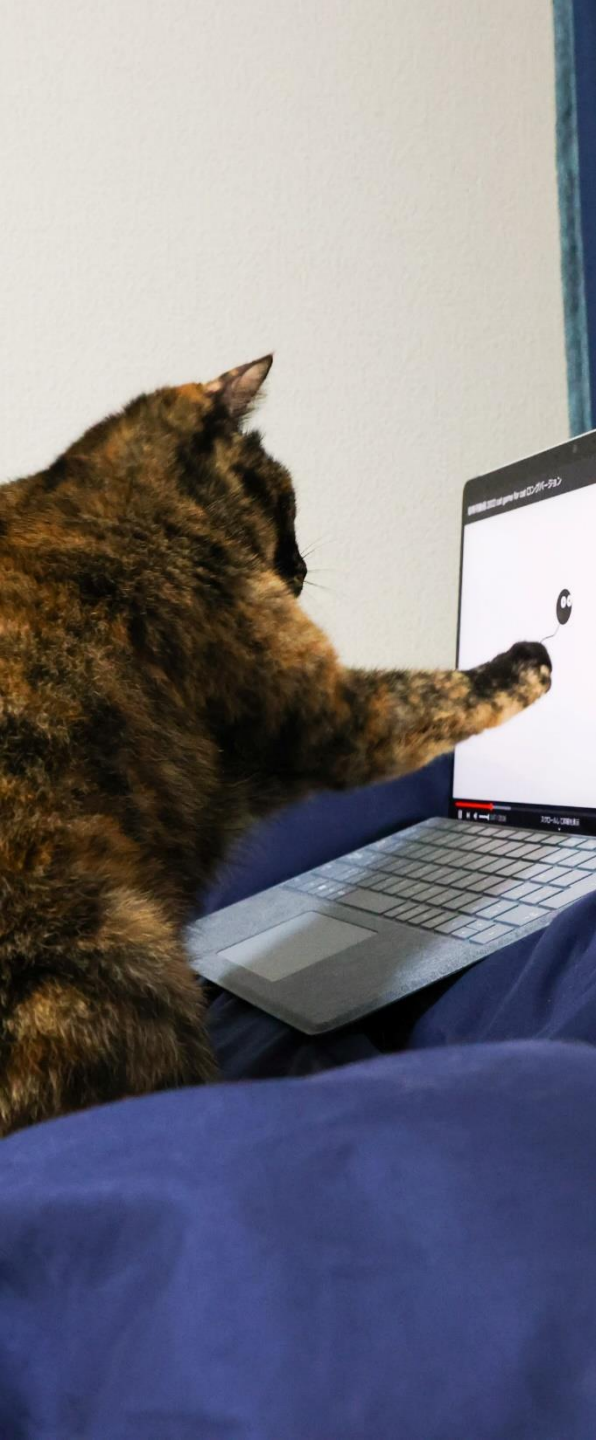
- ***Least Recently Used* (LRU):** remove do cache o item que não foi acessado por mais tempo. Este algoritmo é baseado na suposição de que itens acessados recentemente têm maior probabilidade de serem acessados novamente no futuro próximo. O LRU é bastante eficiente e pode ser implementado na prática usando ***listas duplamente encadeadas*** e ***tabelas hash*.**

Algoritmos de substituição no Cache - LFU

- ***Least Frequently Used* (LFU):** O LFU evita a remoção de itens que são frequentemente acessados, substituindo o item com a menor contagem de acessos. O algoritmo pode ser implementado com o uso de **tabelas *hash*** e estruturas de *heap*, mas pode ser menos eficiente do que o LRU em alguns cenários, já que itens frequentemente acessados no passado podem não ser necessários no futuro.

Algoritmos de substituição no Cache - LRU

- ***First-In, First-Out* (FIFO):** Este algoritmo simples remove o item que foi inserido primeiro na cache. O FIFO não considera a frequência ou a recenticidade dos acessos, o que pode levar a uma eficiência menor em comparação aos algoritmos LRU e LFU. No entanto, o FIFO é fácil de implementar usando apenas filas.



Estruturas de Dados

Estruturas de Dados

- Uma estrutura de dados é uma forma de organizar e armazenar dados em memória de forma eficiente. As estruturas de dados são projetadas para permitir que os dados sejam acessados, modificados e manipulados de maneiras diferentes e eficazes. Uma estrutura de dados deve conter, no mínimo: inserção, exclusão e busca como métodos de manipulação de dados.

Listas Duplamente Encadeadas

Uma lista duplamente encadeada é uma estrutura de dados linear que armazena uma coleção de elementos, chamados de vértices de forma ordenada. Cada vértice é composto de três componentes:

1. Um valor que armazena os dados do vértice.
2. Uma referência (ou ponteiro) para o vértice anterior na sequência.
3. Uma referência (ou ponteiro) para o próximo vértice na sequência.

A lista em si possui referências para o primeiro (cabeça) e o último (cauda) nó.

Listas Duplamente Encadeadas - Python

No Python, você pode utilizar a classe deque da biblioteca collections para implementar listas duplamente encadeadas. A classe deque oferece inserção e remoção eficientes de itens no início e no final da lista.

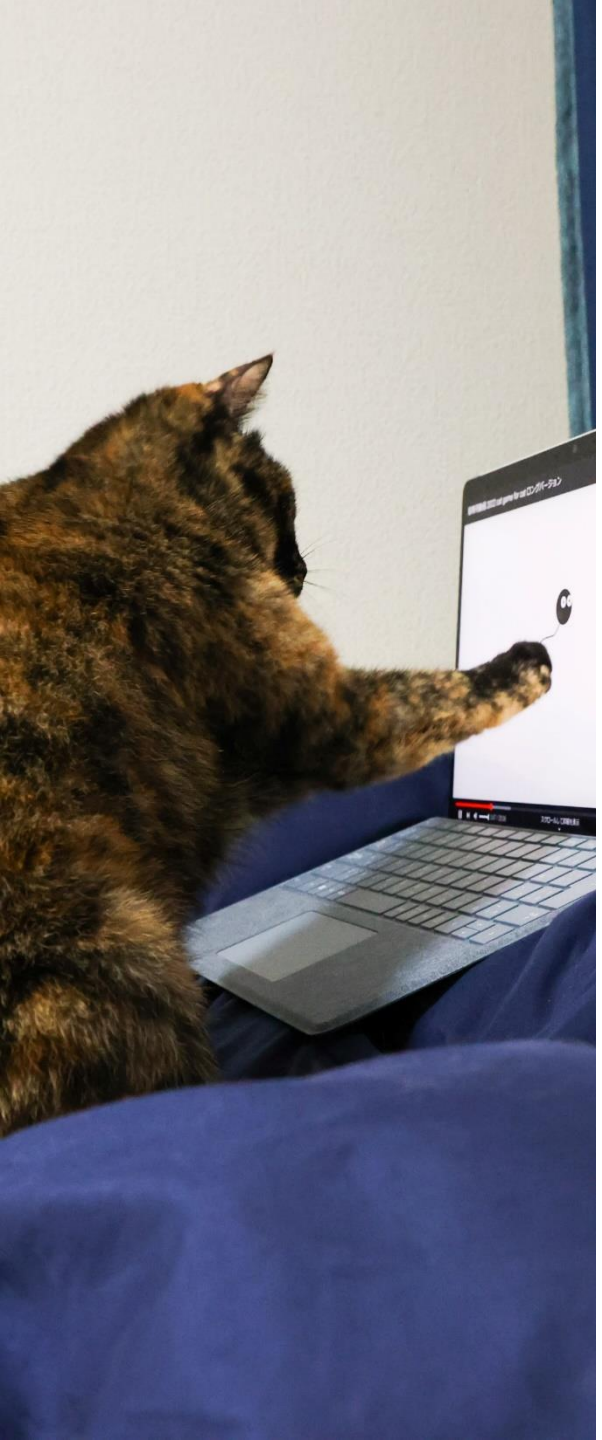
1. **Inserção:** Adicionar um novo elemento à lista. No início: ***deque_obj.appendleft(value);*** no final: ***deque_obj.append(value)***

2. **Remoção:** Excluir um elemento da lista. No início: ***deque_obj.popleft()***; no final: ***deque_obj.pop()***

Listas Duplamente Encadeadas - Python

3. **Busca:** Localizar um elemento na lista com base em seu valor ou posição. A busca pode ser feita percorrendo a lista a partir da cabeça ou da cauda, dependendo da posição do elemento desejado. Com a classe deque, você pode iterar sobre a lista usando um *loop for* ou acessar um elemento em uma posição específica usando indexação, como ***deque_obj[index]***.

4. **Travessia:** Percorrer a lista visitando cada elemento sequencialmente. A travessia pode ser feita na direção da cabeça para a cauda ou da cauda para a cabeça. Use um laço de repetição:



Exemplo de Lista Duplamente Encadeada em Python, Usando Deque

Exemplo usando números inteiros e strings

Hash Tables - Dictionaries

Uma *hash table* (ou tabela *hash*) é uma estrutura de dados que implementa um mapa de chaves para valores, permitindo uma busca, inserção e remoção eficientes de elementos. No Python, a classe dict (dicionário) é a implementação padrão de uma *hash table*.

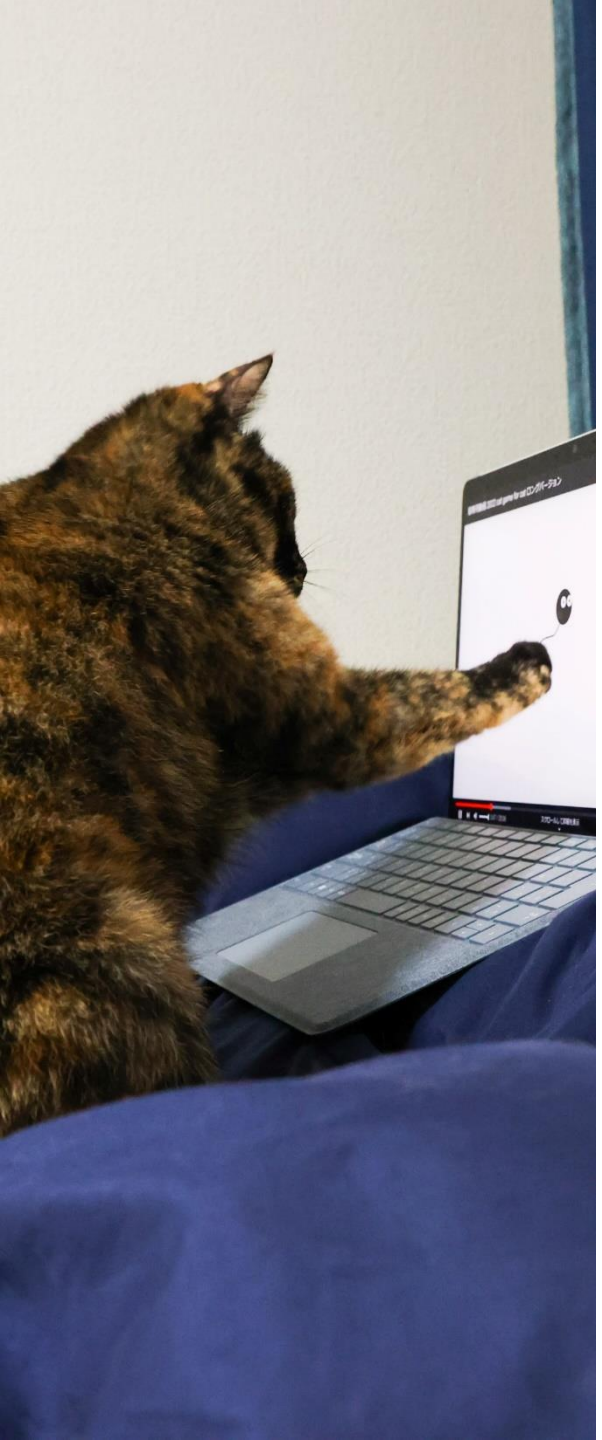
1. **Inserção:** Adicionar um novo par chave-valor à tabela hash: ***hashTable[key] = value***;

2. **Remoção:** Excluir um par chave-valor da tabela ***hash: del hashTable[key]***

Hash Tables - Dictionaries

3. **Busca:** Localizar e retornar o valor associado a uma chave na tabela hash: ***valor = hashTable[key]***

4. **Travessia*:*** Percorrer todos os pares chave-valor na tabela hash: use um laço;

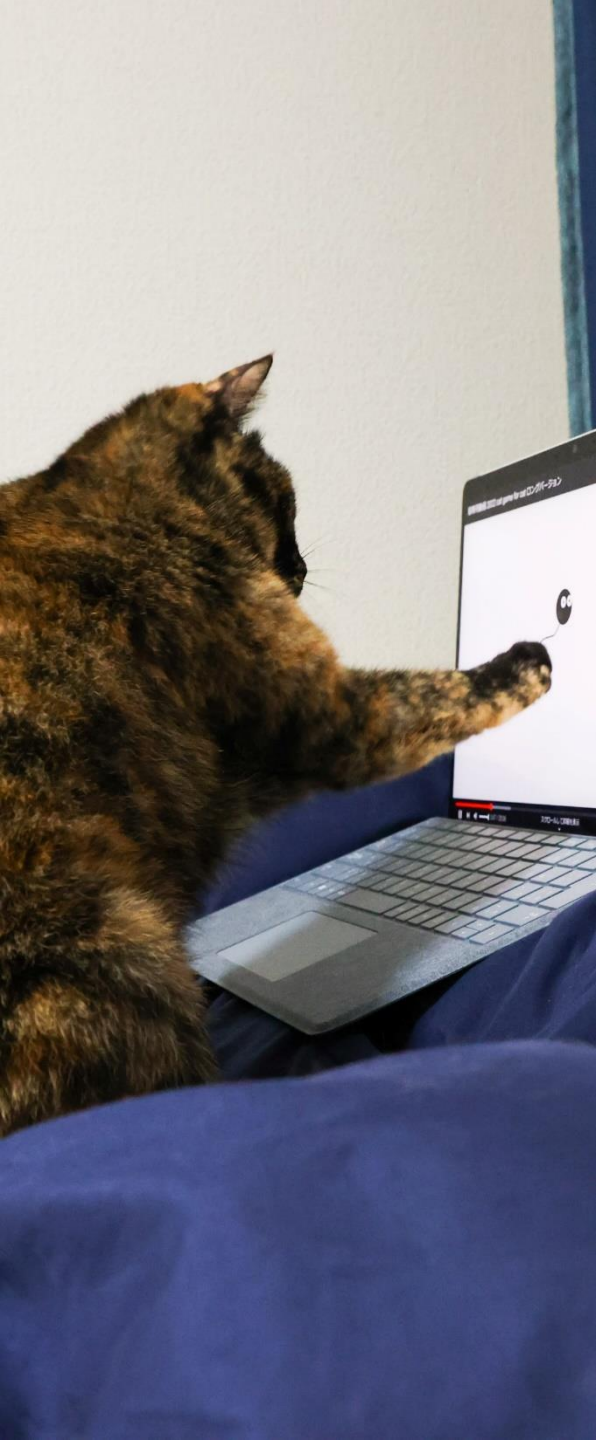


Exemplo de Hash Table em Python, Usando Dict

Exemplo usando números inteiros e strings

Filas

Uma fila é uma estrutura de dados que segue o princípio "primeiro a entrar, primeiro a sair" (FIFO - *First In, First Out*). A classe Queue, uma implementação de uma fila, permite que os elementos sejam adicionados e removidos da fila de acordo com o princípio FIFO. Inclui métodos para adicionar elementos ao final da fila (***put()***), remover elementos do início da fila (***get()***), e verificar se a fila está vazia (***empty()***).



Exemplo de Fila em Python, Usando Queue

Exemplo usando números inteiros e strings

Voltando ao LRU

1. Use a classe deque para criar uma lista duplamente encadeada, armazenando itens da cache.

2. Use um dicionário Python para criar uma tabela hash, armazenando chaves e referências aos nós da lista encadeada.

3. Ao acessar um item:

    Verifique se o item está presente na tabela hash.

        Se presente (cache hit):

            Remova o nó da lista encadeada.

            Insira o nó no início da lista encadeada.

Se não presente (cache miss):

Voltando ao LRU

3. Ao acessar um item:

Verifique se o item está presente na tabela hash.

Se presente (cache hit):

Remova o nó da lista encadeada.

Insira o nó no início da lista encadeada.

Se não presente (cache miss):

Busque o item no disco rígido.

Se cache cheia:

Remova o último item da lista encadeada e sua entrada na tabela hash.

Insira o novo item no início da lista encadeada.

Voltando ao LRU

4. Se cache cheia:

    Remova o último item da lista encadeada e sua entrada na tabela hash.

    Insira o novo item no início da lista encadeada.

    Adicione uma nova entrada na tabela hash para o novo item.

O LFU pode ser feito só com Hash

1.Use um dicionário Python para criar uma tabela *hash*, armazenando chaves e valores dos itens da cache, bem como suas frequências de acesso.

2. Use outro dicionário Python para manter um registro das frequências de acesso e uma lista de chaves correspondentes.

3. Ao acessar um item:

a) Verifique se o item está presente na tabela hash de chaves e valores.

b) Se presente (cache hit):

Atualize a frequência de acesso do item na tabela hash de chaves e valores.

O LFU pode ser feito só com Hash

3. Ao acessar um item:

a) Verifique se o item está presente na tabela *hash* de chaves e valores.

b) Se presente (*cache hit*):

Atualize a frequência de acesso do item na tabela *hash* de chaves e valores.

Atualize a tabela hash de frequências de acesso.

c) Se não presente (*cache miss*):

Busque o item no disco rígido.

O LFU pode ser feito só com Hash

c) Se a cache está cheia:

Encontre a menor frequência de acesso na tabela *hash* de frequências de acesso.

Remova a entrada com a menor frequência de acesso na tabela hash de frequências de acesso e na tabela *hash* de chaves e valores.

d) Adicione o novo item na tabela *hash* de chaves e valores com frequência de acesso igual a 1.

e) Adicione uma nova entrada na tabela *hash* de frequências de acesso para o novo item.