Hash Table Design Document

Overview

This program implements a hash table data structure from scratch in Python, without using Python's built-in dictionary or collections library. This hash table works with mostly O(1) without depending on the number of items in the hash table. It has operations such as insertion, deletion, retrieval and rehashing .

Components

1. Hash Function

The hash function used in this implementation is the FNV-1a hash function, which is a non-cryptographic hash function known for its good distribution and low collision probability.

- calculate_hash(key, seed):Implements the FNV-1a algorithm, taking a string key and an optional seed value as input, and returning a hash value.

2. Item Class

The Item class represents one key-value pair in the hash table. Each Item instance has three attributes:

- key: The key of the item, which must be a string.
- value: The value of the item, which can be any type.
- next: A reference to the next Item in the linked list for handling collisions.

3. HashTable Class

The HashTable class is the main component of the hash table implementation. It has the following attributes:

- bucket_size: The size of the bucket array, set to 97 by default (a prime number to reduce collisions).
- buckets: An array of buckets. Each bucket is a linked list that stores Item instances with the same hash value.
- item_count: The total number of items in the hash table.
- max_load_factor: The maximum load factor threshold, set to 0.7 by default.
- min_load_factor: The minimum load factor threshold, set to 0.3 by default.

The HashTable class provides the following methods:

- put(key, value): If the key already exists in the hash table then updates its value. Otherwise, if the key does not exist, inserts a new key-value pair into the hash table.
- get(key): Retrieves the value associated with the given key from the hash table.
- delete(key): Removes the key-value pair associated with the given key from the hash table.
- size(): Returns the total number of items in the hash table.
- rehash(): Adjusts the bucket size based on the load factor (item_count / bucket_size) to maintain efficiency. If the load factor exceeds the maximum threshold, the bucket size is doubled. If the load factor is lower than the minimum threshold and the bucket size is greater than 100, the bucket size is shrunk to half of the original. All the items from the old hash table need to be put into the new hash table, so the time complexity of the rehash function is O(n).
- rehash_put(key, value): The simplified version of the put function. A helper function used during the rehashing process to put an item in the new hash table.

4. Test Functions

The program includes two test functions:

- functional_test(): Tests the basic functionality of the hash table by performing various operations (insertion, retrieval, update, deletion).
- performance_test(): Tests the performance of the hash table by repeatedly inserting, retrieving, and deleting a large number of items. The execution time of each iteration is printed, and the goal is to achieve mostly O(1) performance.

Design Choices

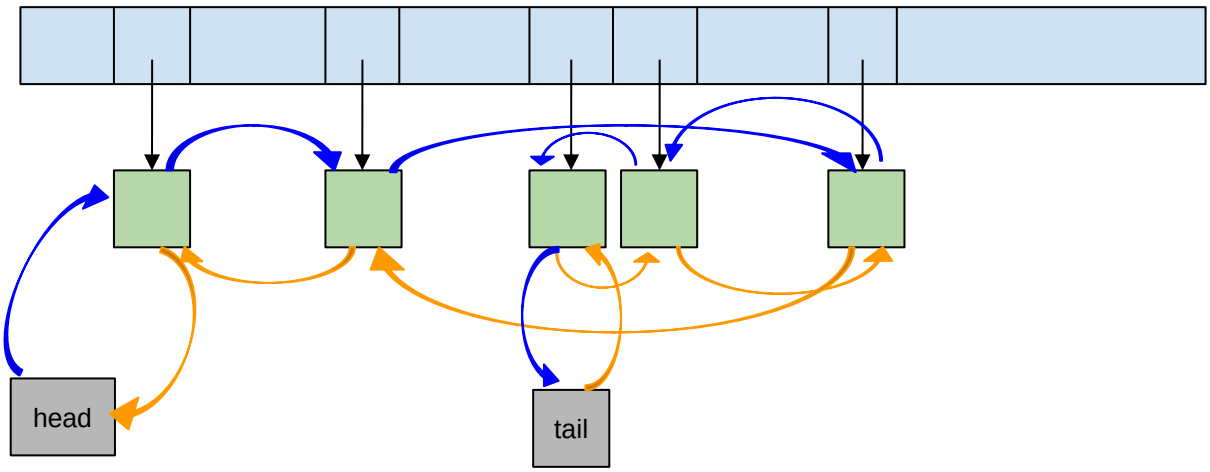
1. **Rehashing Strategy**: The rehashing strategy employed in this implementation is to double the bucket size when the load factor exceeds the maximum threshold (0.7) and halve the bucket size when the load factor is lower than the minimum threshold (0.3) and the bucket size is greater than 100. This approach aims to maintain a balanced load factor and keep the performance of the hash table within the desired O(1) time complexity.
2. **Hash Function**: The FNV-1a hash function is chosen for its good distribution and low collision probability.
3. **Initial Bucket Size**: The initial bucket size is set to 97, a prime number, to reduce the likelihood of collisions. However, this value can be adjusted based on the expected size and distribution of the input data.

Design a cache that achieves the following operations with mostly O(1)

- When a pair of is given, find if the given pair is contained in the cache or not
- If the pair is not found, insert the pair into the cache after evicting the least recently accessed pair

Use one hash table and one doubly linked list to achieve the constant-time insertion, retrieval, and deletion. The hash table is used to store the page data, and the doubly linked list is used to maintain the order of pages.

- A hash table to store the pages, using the URL as the key and the DoublyLinkedList node as the value.
- A doubly linked list to maintain the order of pages, each node is one single page, using the URL as the key and the page contents as the value.



Cache Design Document

Overview

This program implements a cache data structure that stores the most recently accessed N pages. The cache is designed to maintain the order of accessed pages, allowing the insertion and retrieval of the accessed pages work mostly O(1). The implementation does not rely on Python's built-in dictionary or collections library, ensuring that the data structure is implemented from scratch.

Components

1. DoublyLinkedList Class

The DoublyLinkedList class represents a node in a doubly linked list, which is a single page stored in the cache. Each node has the following attributes:

- key: The URL of the page.
- value: The contents of the page.
- prev:The previous node in the linked list.
- next: The next node in the linked list.

2. Cache Class

The Cache class is the main component of the cache implementation. It has the following attributes:

- size: The maximum size of the cache.
- page_counter: The number of pages currently stored in the cache.
- cache: A hash table (implemented separately) to store the pages, using the URL as the key and the DoublyLinkedList node as the value.
- head: The head of the doubly linked list. The next node of head is the most recently accessed page.
- tail: The tail of the doubly linked list. The previous node of tail is the least recently accessed page.

The Cache class provides the following methods:

- access_page(url, contents): Accesses a page and updates the cache to store the most recently accessed N pages. If the page is already in the cache, it moves the corresponding node to the head of the linked list. If the page is not in the cache, it creates a new node and adds it to the head of the linked list. After adding a new page, if the cache is full, it removes the least recently accessed page.
- get_pages(): Returns the URLs stored in the cache, ordered from most recently accessed to least recently accessed.
- addNode(node): Adds a new node to the head of the linked list.

- removeNode(node): Removes a node from the linked list.
- moveToHead(node): Moves a node to the head of the linked list.
- removeTail(): Removes the least recently accessed node from the linked list.

3. Test Function

The cache_test() function is provided to test the functionality of the cache implementation. It performs operations on the cache, such as accessing pages, removing least recently accessed pages, and verifying the order of the cached pages.

Design Choices

1. **Doubly Linked List**: The implementation uses a doubly linked list to maintain the order of the pages in the cache. This approach allows for efficient insertion and removal of nodes, which is crucial for maintaining the order of the most recently accessed pages.
2. **Hash Table**: A hash table implementation is used to store the pages, using the URL as the key and the DoublyLinkedList node as the value. This design choice allows for constant-time insertion, retrieval, and deletion of pages from the cache.
3. **Separate Cache and Linked List Components**: The cache implementation uses one hash table and one doubly linked list to achieve the constant-time insertion, retrieval, and deletion. The hash table is used to store the page data, and the doubly linked list is used to maintain the order of pages.
4. **Time Complexity**: The access_page and get_pages methods are designed to operate in mostly O(1) time complexity. The linked list operations (addNode, removeNode) are also performed in constant time.