Monday, November 4, 2019

Explanation

Problem 1: Least Recently Used Cache

Design:

I use two types of data structure to accomplish LRU Cache: dictionary and DoublyLinkedList.

Dictionary records every new node that is pushed by calling set_() function, and deletes the node that is pushed out. Nodes are stored in dictionary by its key value. The time complexity of searching, inserting, and deleting in dictionary is O(1) constant. The space complexity of storing n nodes is O(n).

DoublyLinkedList is used to construct the relationship between each node, such as:

self.head = (4,4) <-> (3,3) <-> (2,2) <-> (1,1) = self.tail

For any node that is called by get() function, that node will be moved out from its current position and added as before the current self.head. New relationships will be established by doubly linking its previous node to its next node (if the node is not self.tail, otherwise the next of previous node is None). The current self.head node becomes the next node of the called node, and called node becomes the self.head. The time complexity of searching, inserting, and deleting in DoublyLinkedList is O(1) constant. Since the get() function just accepts an integer each time, the space complexity is O(1) constant, regardless of the input size.

The limitation of current data structure is that, if we push the exiting node into LRU Cache, the it will be difficult for DoublyLinkedList to get the correct position of the node that has same key and value. Therefore, I current set this case as edge test and don't allow LRU Cache to accept any node that is already existing in Cache. Another two edge tests are empty input and string element input. For normal tests, the nodes that locate at head, mid, and tail are called to make sure LRU Cache is able to rearrange the node relationships in any cases.

Time complexity:

O(1) constant cost

Space complexity:

O(n) linear cost

Problem 2: File Recursion

Design:

File recursion is a tree traversal problem where files act as leaves and directories act as internal nodes. The time complexity of tree structure is averagely O(log(n)) for searching and O(n) in worst scenario. The space complexity is O(n).

Recursion technique is used to traverse all the nodes and leaves, until the target files are found. The time and space complexity of using recursion depends on the data structure inside and the method applied for searching. In our case, we traverse every single directory to the end before visit other directory. Therefore, the searching method is a Depth First Search algorithm that has time complexity is O(|E| + |V|), and space complexity is O(|V|), where V is vertices, or n, and E is edges.

Since the files have already been stored in directory, the only thing we care about is to traverse and search the target files. Since we apply DFS, the overall time complexity is O(|E|+|V|) and space complexity is O(|V|).

Time complexity:

O(|E| + |V|) by using DFS for searching, where V is vertices/target files, and E is edges

Space complexity:

O(|V|) linear cost, where V is vertices/target files, and E is edges

Problem 3: Huffman Code

Design:

In this problem I used three types of data structure: linked list, queue, and binary tree to accomplish Huffman code task. For any input string sentence, it will be first split into single string characters and stored in a dictionary with their appearing frequency. The dictionary will be stored from lowest to highest frequency before encoding. A Queue data structure is used to store all characters and their frequencies. In encoding process, two characters with lower frequency will be dequeued following LIFO rule at each time. A new node is formed and queued with None as key and the sum of frequencies of these two nodes. Two nodes will be assigned as left and right leaves of the new formed node. This process is continue until only root node with highest frequency left. After all nodes are merged into tree data structure, we will traverse the tree and assign binary code to the leaves of all node. Finally, a recursion technique is used to combine all binary code from top root node to the end leaf and assign the combined binary code to that string character. A new dictionary is created to store characters and their Huffman code for decoding process in the future. Huffman encoding process is complete.

In decoding process, since each character and its corresponding Huffman code are stored in dictionary. We can easily find the character by the inputed Huffman code.

Time complexity:

Encoding:

- Frequency sorting: O(n) for assigning element into dictionary, and O(n*log(n)) for sorting
- Merge to node: O(n) linear cost
- Traverse: O(n) linear cost
- Assign binary code: O(n) linear cost

# Decoding:

O(1) constant cost for finding character in dictionary, O(n) linear cost for decoding all Huffman code

# Overall: highest time complexity is O(n*log(n)) in frequency sorting function

Space complexity:

# Encoding:

- Frequency sorting: O(n) for assigning element into dictionary
- Merge to node: O(n)
- Traverse: O(n)
- Assign binary code: O(n)

# Decoding:

O(n) for storing all string characters

# Overall: highest space complexity is O(n)

Problem 4: Active Directory

Design:

It is similar to problem 2, Active Directory is a tree traversal problem and we are using DFS method to traverse all the stored node from directory tree.

For inserting new elements into tree data structure, the time complexity is O(log(n)) averagely and O(n) in worst scenario, and the space complexity is O(n). On the other hand, is_user_in_group() is a searching problem that is implemented by recursive technique. Since in the worst scenario, the first call won't finish executing until all lower level of sub-functions in that same level function complete, the searching method is a Depth-First-Search (DST), with time complexity of O(|E| + |V|) and space complexity of O(|V|). Where E is the edge and V is the number of vertices, or leaves.

Time complexity:

O(|E| + |V|) by using DFS for searching, where V is vertices/target files, and E is edges

Space complexity:

O(|V|) linear cost, where V is the target group, and E is edges

Problem 5: Blockchain

Design:

Blockchain is a linked list that has the time complexity of O(1) for insertion, deletion, and O(n) for searching. The space complexity is O(n) for n blocks. Each blockchain stores the information of transaction time, data, and hash value of previous blockchain. Every new node is attached as the tail of previous node. The search is conducted from tail to head until the certain data is found.

Time complexity:

O(1) constant cost for insertion and deletion

O(n) linear cost for searching, n is the number of block

Space complexity:

O(n) linear cost, n is the number of block

Problem 6: Union and Intersection of Two Linked Lists

Design:

In this problem the linked list structure is used that has the cost of O(1) for insertion, deletion and O(n) for searching. After two linked lists are created, we need to find the union and intersection group between two linked lists.

The logic for union group is that by first assigning all the unique data from linked list 1 into union group, only the data from linked list 2 that doesn't appear in union group will be assigned.

The logic for intersection group is that only the unique data from linked list 1 that can also be found in linked list 2 will be assigned to intersection group.

Time complexity:

O(n) linear cost for intersection function

O(n^2) quadratic cost for union function in worst scenario, since for every node in the linked list, it performs the search again in another list. Optimization can be done in the future to lower the cost

Space complexity:

O(n) linear cost for union function, where n is the total number of elements in both input lists at worst scenario when two lists are completely different

O(n) linear cost for intersection function also, where n is the total number of elements in short (relatively) input list. The worst scenario occurs when the all of the elements of short input list are also existing in long input list.