Nanodegree Data Structure and Algorithms

PROJECT 1- Data Structure Questions

Explanation LRU Cache

We know dictionary operations get/set are O(1) but it is needed the most Recently Used Element first. We know that add/remove from a linked list is O(1) but we cannot get in O(1) then:
I chose a dictionary for GET method O(1) and a Linked List to add/remove method to the head (most recently used) in O(1). But there is an important restriction to understand the value if this solution , is required to save every node inside dictionary. Then, move a node to the head is O(1) and get/set are O(1) too.

Complexity Time: O(1) to add_node/ remove_node /get /set
Complexity Memory: LRU_Cache (dictionary, double linked list), O(constant) == O(1)

Explanation Finding Files

The Recursive Solution used: identify a file with the particular suffix, path in the current directory and do the same with all subfolders.

Complexity Time: O(n) n: (folders+subfolders...) ,
Complexity Memory: O(constant) constant: size of the list

Explanation Huffman Coding

The solution was implemented according to the pseudoCode and some research following the next steps:

1. Take a string and determine the relevant frequencies of the characters, returning a dictionary with the frequencies.

    # frequencies(data: str) -> t.Dict[chr, int]:

    Complexity Time: O(n) n: size of data
    Complexity Memory: O(constant) c= size of dictionary

2. Build and sort a list of tuples from lowest to highest frequencies.
3. Build the Huffman Tree by assigning a binary code to each letter, using shorter codes for the more frequent letters. (This is the heart of the Huffman algorithm.)

    I used a sorted list to manage the priorities and gathering the trees

    # def build_priority(dict: t.Dict[chr, int]) -> t.List[Tree]:

    Complexity Time: O(n) n: size of dictionary<=size of data data
    Complexity Memory: O(constant) c=2*(size of list in mem)

    # def merge(tree_list: t.List[Tree]) -> Tree:

    Complexity Time: O(nlog(n)) → O(n) + O(nlog(n)) n: size of tree-1
    Complexity Memory: O(constant) c= size of Tree in Mem

4. Trim the Huffman Tree (remove the frequencies from the previously built tree).
5. Encode the text into its compressed form.

    Traverse the tree was needed to find the encode code
    # def traverse_tree(bt: Tree, elem: str, encode: str, output: t.List) -> str:
    # def huffman_encoding(data):

    Complexity Time: O(n) n: size of data (tree)
    Complexity Memory: O(1)

6. Decode the text from its compressed form.

    Traverse the tree was needed to determine every char from the encode and return the decoding string

    # def traverse_decode(copy_inmutable: Tree, data_list: t.List, output: t.List, parent: Tree):
    # def huffman_decoding(encode: str, tree: Tree):

    Complexity Time: O(n) n: size of data(tree)
    Complexity Memory: O(constant) c= size of list

Explanation Active Directory

.

The Recursive Solution used: to look up of whether the user is in a group I identify if the user belongs to the current group or belongs to the inners “groups.” If belongs at least one

then return true otherwise if its not present in main group and subgroups return false. The time complexity depends of groups

Complexity Time O(n*m) n: (groups+subgroups) , m:users
Complexity Memory: Group Structure (2 lists), O(2[size of the list in Mem]), O(constant)==O(1)

Explanation Blockchain

Having a block structure and a calc_hash "sha256()" encode what we need to do is create the blockchain list and a methods, example add a block to the chain and verify blockchain respectively to add an element referencing the previous and to verify if the blockchain was not modified by others. Then Blockchain structure is a linked list where every block contains their hash and previous hash.

Complexity Time add Element O(1) verify list O(n)
Complexity Memory O(constant) Block structure + Blockchain(list) constant: size of List
Add Element → O(1)
Verify list → O(1)

Explanation Union and Intersection

Union: Going through the list removing repeated for each list having a set and then merging the lists. Time complexity is O(n)/ Complexity Memory O(Node*num_Nodes)
Intersection: Going through the list removing repeated for each list having a set and then removing uniques from the lists. Time complexity is O(n)/ Complexity Memory O(Node*num_Nodes)