## **Blockchain:**

Single Linked List used for this problem. Each block has pointer ‘next’ which point to next node. Blockchain object created using BlockChain class, which represents single linked list. It has 2 pointers; one points to root/head node. Another one points to last/tail node. With the help of tail node pointer, we would able to set previous\_hash value for the current node

Time Complexity =O(n), where ‘n is number of node

Space Complexity=O(n\*l)=O(n), where ‘n’ is number of node and ‘l’ is size of the node

## **LRU Cache:**

Implement Least recently used cache. Cache is fixed size. Whenever new element is added or existing element access it should be considered as most recently item. Once cache size is full, new added element would be consider as most recently used and least recently used element should be removed from cache. All operations(get/set) should be O(1)

Time complexity , to add Element = O(1)

Time complexity , to get Element = O(1)

Space Complexity=O(n\*l)=O(n), where ‘n’ is maximum cache size and ‘l’ is size of the linked list node

Doubly linked list used to store elements, where ‘least recently used’ element would be head and ‘most recently used’ element would be tail. Two pointers (head/tail) maintain to access least/most recently used quickly.

|  |  |  |
| --- | --- | --- |
|  | 1 |  |

|  |  |  |
| --- | --- | --- |
|  | 2 |  |

|  |  |  |
| --- | --- | --- |
|  | 3 |  |

|  |  |  |
| --- | --- | --- |
|  | 4 |  |

|  |  |  |
| --- | --- | --- |
|  | 5 |  |

LRU\_Cache class used to implement ‘LRU\_Cache’ . following attributes are present in the class

MAX\_SIZE= maximum size of the cache

No\_elements= current size of the cache

Cache= dictionary object, where key if element to be added/retrieve and value should be corresponding node of the linked list. This dictionary used to implement all operations with O(1).

LRU- Linked list, which is explained above

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | |  |  |  | | --- | --- | --- | |  | 1 |  | |
| 2 | |  |  |  | | --- | --- | --- | |  | 2 |  | |
| 3 | |  |  |  | | --- | --- | --- | |  | 3 |  | |
| 4 | |  |  |  | | --- | --- | --- | |  | 4 |  | |
| 5 | |  |  |  | | --- | --- | --- | |  | 5 |  | |

**Set operation:**

While adding element,

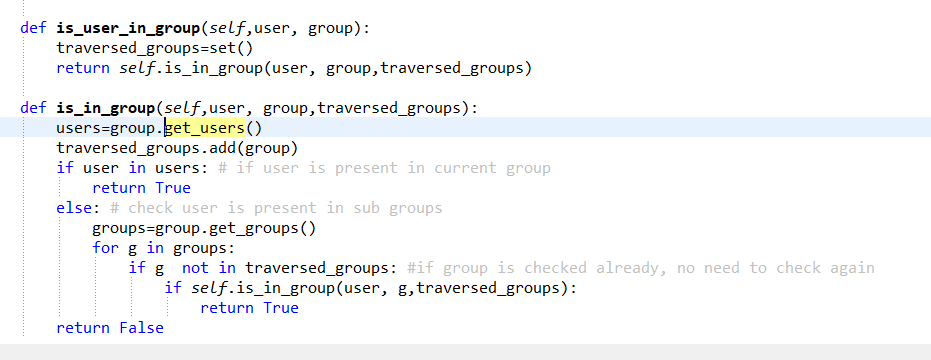
1. if element is already present (by checking in cache dictionary), move the element node into last in the linked list
2. If element is not present and maximum size is not reached. Create node and link the node as last node in linked list and add entry in dictionary with element value as ‘key’ and node as ‘value’. Increase the no\_elements by one
3. If element is not present and maximum size is reached,
4. Remove the first node from the linked list and it’s corresponding entry from cache dictionary. Mark second node in the linked list as recently used element.

ii ) Add new node at the end of the linked list and add that node in the cache dictionary

**Get Operation:**

1. If element is not present,
   1. Check cache dictionary has element ; if not, element is not present in LRU cache.
2. If element is most recently used,
   1. Element is present in dictionary and its last node in the linked list; no change in the linked list
3. If element is least recently used,
   1. Element is present in dictionary. Move the first node to last in the linked list.
4. If element is in the middle of the linked list,
   1. Element is present in dictionary. move that node to last node by changing the pointers

## **Active Directory:**



is\_user\_in\_group() method used to check whether user is present in group (or) its sub group.

is\_in\_group() called recursively to check whether user is present in group or not. traversed\_group set maintains the list of group checked, which avoids checking the same group again in our recursive calls

**Time Complexity= O(n), n is number of groups, as we traverse group**

**Space Complexity= O(d\*n), where d is number of users, n is number of groups**

## **Union And Intersection:**

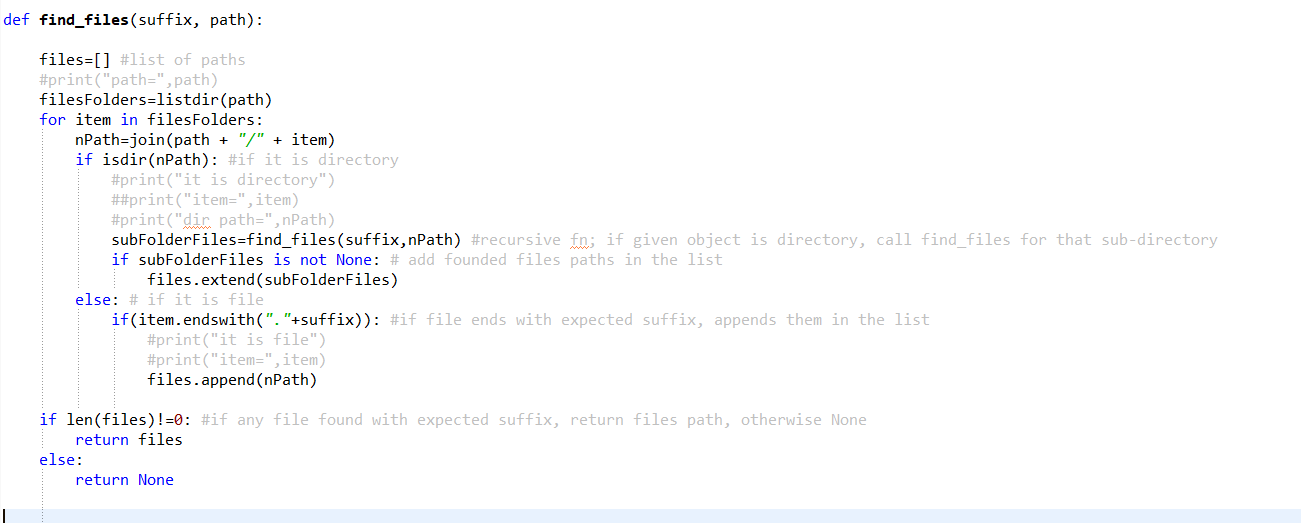
1. Create Linked list ‘List1’ & List2’
2. Convert them to set1, set2
3. Perform Union and intersection operation on set
4. Convert the resultant sets to linked list

**Time complexity for Union= O(n1+n2), where n1 is length of set1, and n2 is length of set2;**

**Time complexity for intersection= O(min(n1,n2)), where n1 is length of set1, and n2 is length of set2**

**Space complexity =O(l\*(n1+n2)) ), where n1 is length of set1, and n2 is length of set2; ‘l’ is size of element in set**

## **File Recursion**



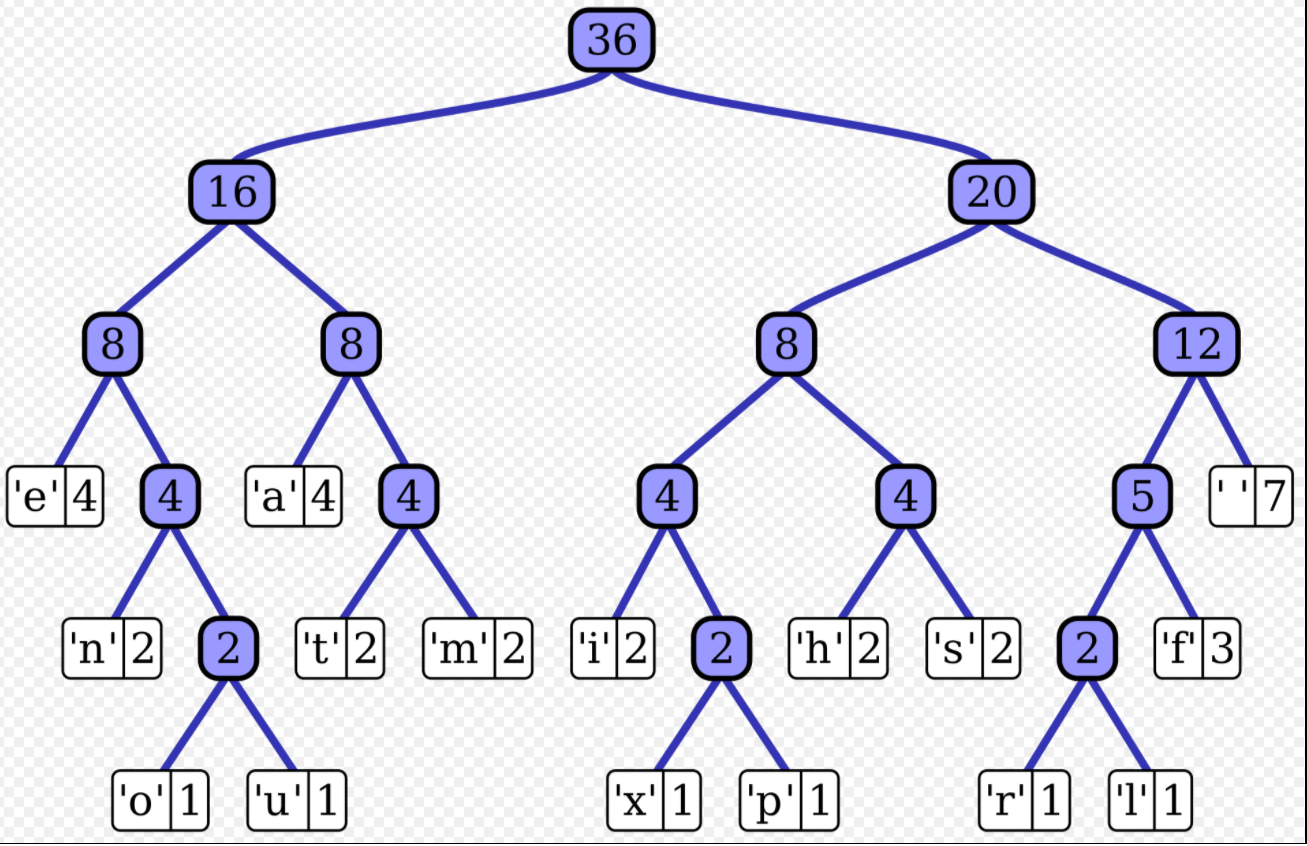
Steps followed:

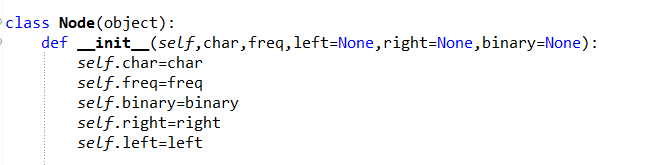
1. Call find\_files method with directory path and file extension as argument
2. Get all objects (files/folders) in the given path
3. If object type is file and extension of the file matches,
   1. add the complete path of the files in ‘files’ list
4. if object type is directory, construct ‘directory path’ . call find\_files method with directory path and file extension as argument (Go to Step 1)
5. return collected files path

**Time complexity for Union= O(n), where ‘n’ number of sub directories; find\_files method called ‘n’ times if ‘n’ sub directories are present in parent**

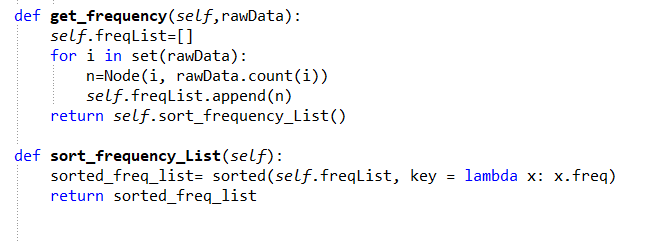
## **Huffman Coding:**

Example Sentence: this is an example of a huffman tree





Huffman tree is created using doubly linked list with above node structure.



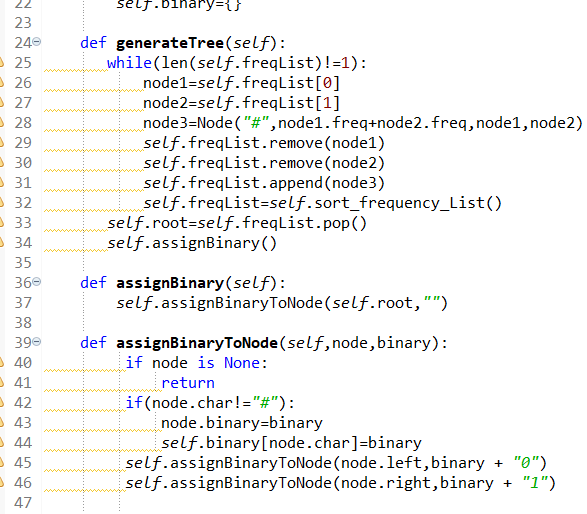
1. First calculate frequency of each character in the sentence and create node for each character and save the nodes in ‘freqList’ list, where node with least frequent chars at first. Refer above code

At the end of this steps, you have ‘n’ nodes where n is number of unique characters in sentence. They are saved in the ‘freqList’ in the ascending order of frequency

1. Huffman tree is generated using generateTree().
2. It takes first 2 items from freqList and construct parent node (node3) and assign char, left & right node and frequencies. (refer code below)
3. Remove the processed nodes from ‘FreqList’ and parent node (node3) in ‘freqList’; Sort the list again.
4. Perform above 2 steps (a& b) until freqList list has one element, which nothing but root node of the tree
5. At this point tree is generated. however binary value is not assigned to each node.



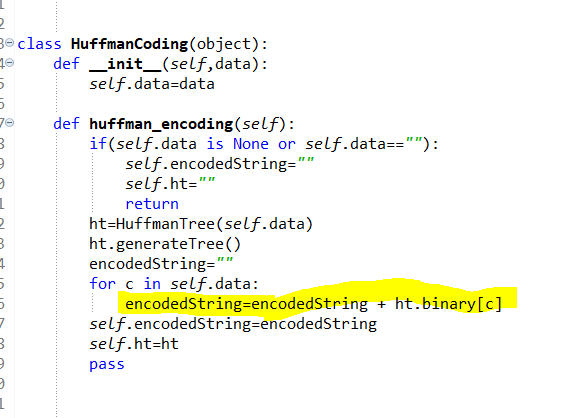
1. Assign Binary Value to nodes
2. assignBinary, assginBinaryToNode method used to assign binary value to node. if it is grouping node (where value is #) , don’t assign binary in the node otherwise assign binary value to node (only for the node with valid char/space has binary value).
3. Code uses preorder traversal (+AB) for traversing tree and append ‘0’ to left node and append ‘1’ to right node.
4. binary dictionary holds chars and its corresponding binary value



**To insert n elements(construct Huffman Tree), the time complexity will become O(n \* log(n)).**

1. encode String:

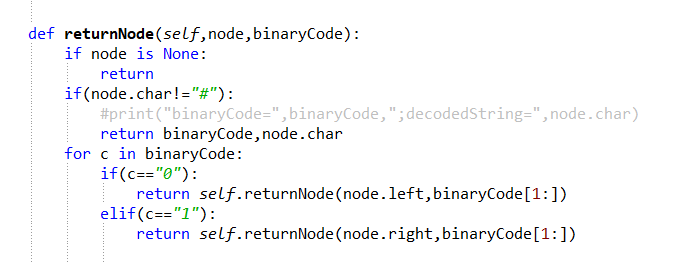
Below code construct the binary code for the given string. Binary dictionary holds character and its binary value.



**To encode string, the time complexity will be O(n \* log(n)). ; based on construction of huffman tree. Where n is number of unique characters**

1. Decode the String

Below code decode the binary code and gets the original sentence. Traverse through tree using given binary code and get corresponding characters

/\*

Time complexity of decode String , based on traverse tree to get correct characters. Edges are characters. Each character is used atleast once in a sentence and traverse is similar to DFS,

**Time complexity O(x\*(V+E)), where V is vertex and E is edge and ‘x’ is average frequency of chars**

