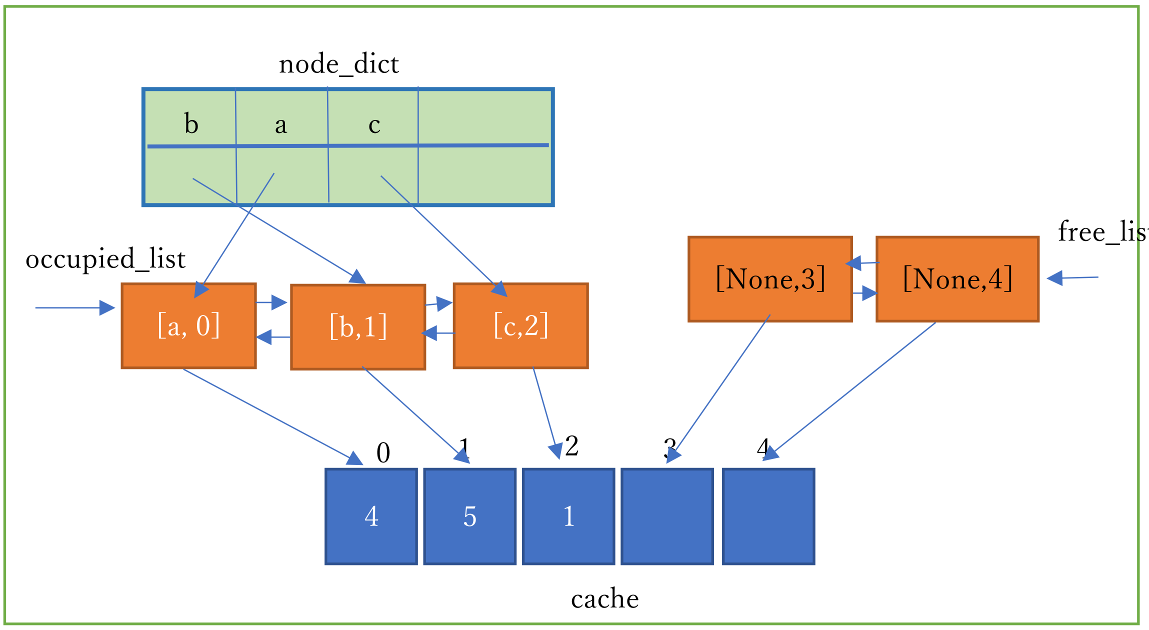
1. Problem 1 – LRU Cache

Three data structures are used here to solve this problem. An array list(cache) is used to save the values. A doubly linked list(occupied_list) is used to keep the element accessing order. Each node of saves [key, index]. A dictionary is used to hold the key : node pairs, so that each element can be accessed in consistent time. An auxiliary linked list(free_list) is used to store free nodes.



When we add new data to the cache(key is not in node_dict), if the cache is not full, we get a node from free_list. The data structure of the node is [key, index]. index is fixed for every node. Then we set node data to [key, index] and append it to tail of the occupied_list. Set cache[index]=value. Finally set node_dict[key]=node. If the cache is full, we pop a node from head of the occupied_list and do the same procedure. The time complexity of adding a new data is O(1).

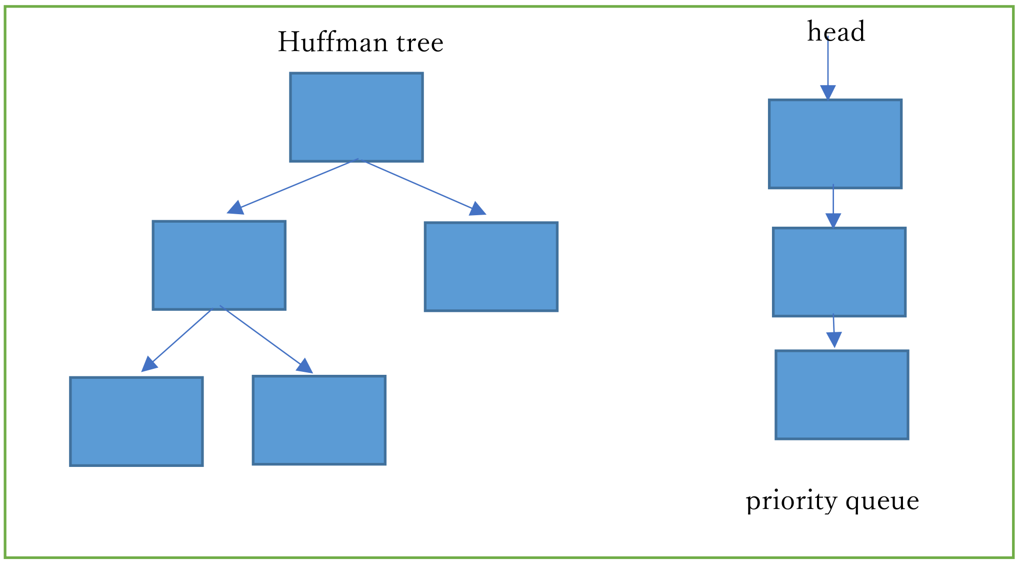
When we modify an existing data, we first search for the key in node_dict, and then get the node from occupied_list. As the node has the index to the cache, set cache[index]=value and put the node to the tail of the occupied_list. The time complexity is O(1).

When we get a data from cache, first we look for the key in node_dict, and then and then get the node from occupied_list. Finally get cache[index] and put the node to the tail of occupied_list. The time complexity is O(1).

Problem 2 – File Finding

A directory is a tree of subdirectories and files. In order to find the specified files, we should traverse through the directory tree in pre-order. As each file in the directory should be accessed once, the time complexity is O(file number).

Problem 3 – Huffman Coding



Huffman encoding:

1. Determine the frequency of each character in the message and save them in freq_map. Suppose the length of the input message is n, the time complexity of this step is O(m).
2. Create a priority queue whose type is MinHeap. MinHeap keeps the elements in ascending order. For each item in freq_map, create a node (character, frequency) and append them to priority queue. Suppose the number of different characters in the message is n, the time complexity of this step is O(n).
3. If priority queue has only one node, pop this node from the priority queue, then create a parent node, set this node as the left child of the parent node.
4. Else pop two nodes from head of the priority queue, and create a node whose node.freq=node1.freq+node2.freq. Then set the previous two nodes as the left and right child of the newly created node. Append the new node to priority queue. In the worst case we have to traverse through the whole linked list a append a node. As a result, the time complexity in worst case is O(n)
5. Iterate step 5 until 1 node exists in the priority queue. The last node is the root of Huffman tree. In the worst case, step 5 will be iterated for n times. So, the time complexity is O(n^2)
6. Calculate Huffman code for each leaf node in the Huffman tree and store them in a

dictionary (global variable codes). The time complexity in worst case is O(1+2+3+…+n) -> O(n^2)

7. Encode the input message character by character using the code dictionary. Time complexity is O(m)
8. return encoded message and Huffman tree. Total time complexity is O(m+n^2)

Huffman decoding:

1. Traverse the input from the start. set node=Huffman_tree.root. If current item is 0, move to left child. If current item is 1, move to right child. Iterate this step until current node is a leaf node. Append the character of the leaf node to the decoded string and set node=Huffman_tree.root.
2. The time complexity is O(m) where m is the length of the input code.

Problem 4 - Active Directory

Recursively traverse through the group tree. In the worst case, time complexity is O(depth of group tree)

Problem 5 – Block Chain

Block class consists of following members: timestamp, data, previous_hash, hash. The hash of a block is hash_function(timestamp+data+previous_hash). The block chain holds the hash of the last block.

The time complexity of adding a block to the block chain is O(1).
In the worst case, the time complexity of searching a block in the block chain is O(n), where n is the number of blocks.

In python, if there is no reference to a piece of data, the memory will be collected by the garbage collector. For this reason, I created a dictionary where the references to the blocks stored.

Problem 6 - Union and Intersection of Two Linked Lists

First, I defined a Linked List where elements are stored in ascending order. Appending an element to the linked list will take O(n) in the worst case, where n is the number of elements in the list.
The linked list keeps a reference to the tail. Member function insert_to_tail append a item to the tail of linked list in O(1) time complexity.

Union:

1. To get the union of two linked list, first define pointers to the head of linked list.
    node1=linkedlist1.head
    node2=linkedlist2.head
2. Get the min=min(node1.value,node2.value). Append it to the tail of union linked list using insert_to_tail function.
   While current value equals to min, move the pointer to next node. do this step until current value does not equal to the min.
3. Iterate step 2 until either of the pointer points to None.
4. Append the remaining value to union list. Skip the duplicated values.

The time complexity of solution is O(n1+n2), where n1 represents the length of first linked list, n2 represents the length of second linked list.

Insection:

1. To get the union of two linked list, first define pointers to the head of linked list.
    node1=linkedlist1.head
    node2=linkedlist2.head
2. If node1.value<node2.value, set node1 to node1.next. Else if node1.value>node2.value, set node2 to node2.next. Else if node.value=node.value, value=node1.value. Append it to the tail of insection linked list using insert_to_tail function.
   While current value equals to value, move the pointer to next node. do this step until current value does not equal to the min.
3. Iterate step 2 until either of the pointer points to None.

The time complexity of solution is O(n1+n2).