Laurent Luce's Blog

Technical blog on web technologies

Recent Posts

Least frequently used cache eviction scheme with complexity O(1) in Python
Cambridge city geospatial statistics
API to access the Cambridge city geospatial data
REST service + Python client to access geographic data
Massachusetts Census 2010 Towns maps and statistics using Python
Python, Twitter statistics and the 2012 French presidential election
Twitter sentiment analysis using Python and NLTK
Python dictionary implementation
Python string objects implementation
Python integer objects implementation

Search

GO

Meta

Log in
Entries RSS
Comments RSS
WordPress.org

Least frequently used cache eviction scheme with complexity O(1) in Python

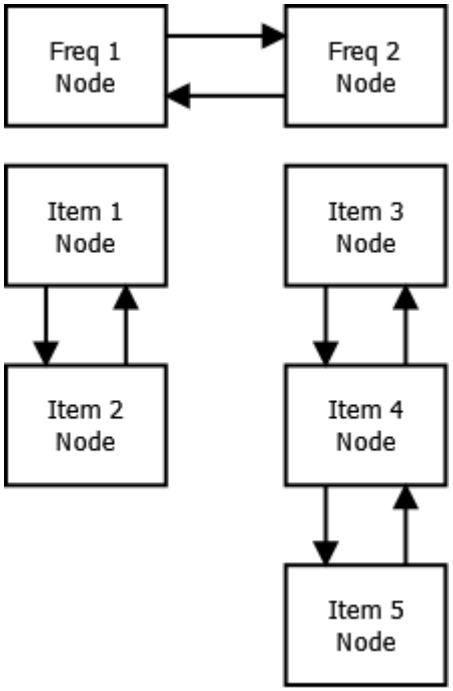
June 10, 2015

This post describes the implementation in Python of a "Least Frequently Used" (LFU) algorithm cache eviction scheme with complexity O(1). The algorithm is described in this paper written by Prof. Ketan Shah, Anirban Mitra and Dhruv Matani. The naming in the implementation follows the naming in the paper.

LFU cache eviction scheme is useful for an HTTP caching network proxy for example, where we want the least frequently used items to be removed from the cache.

The goal here is for the LFU cache algorithm to have a runtime complexity of O(1) for all of its operations, which include insertion, access and deletion (eviction).

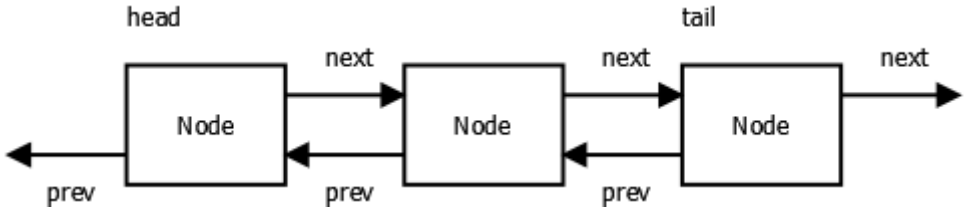
Doubly linked lists are used in this algorithm. One for the access frequency and each node in that list contains a list with the elements of same access frequency. Let say we have five elements in our cache. Two have been accessed one time and three have been accessed two times. In that case, the access frequency list has two nodes (frequency = 1 and frequency = 2). The first frequency node has two nodes in its list and the second frequency node has three nodes in its list.



How do we build that? The first object we need is a node:

```
1  class Node(object):
2      """Node containing data, pointers to previous and next node."""
3      def __init__(self, data):
4          self.data = data
5          self.prev = None
6          self.next = None
```

Next, our doubly linked list. Each node has a prev and next attribute equal to the previous node and next node respectively. The head is set to the first node and the tail to the last node.



We can define our doubly linked list with methods to add a node at the end of the list, insert a node, remove a node and get a list with the nodes data.

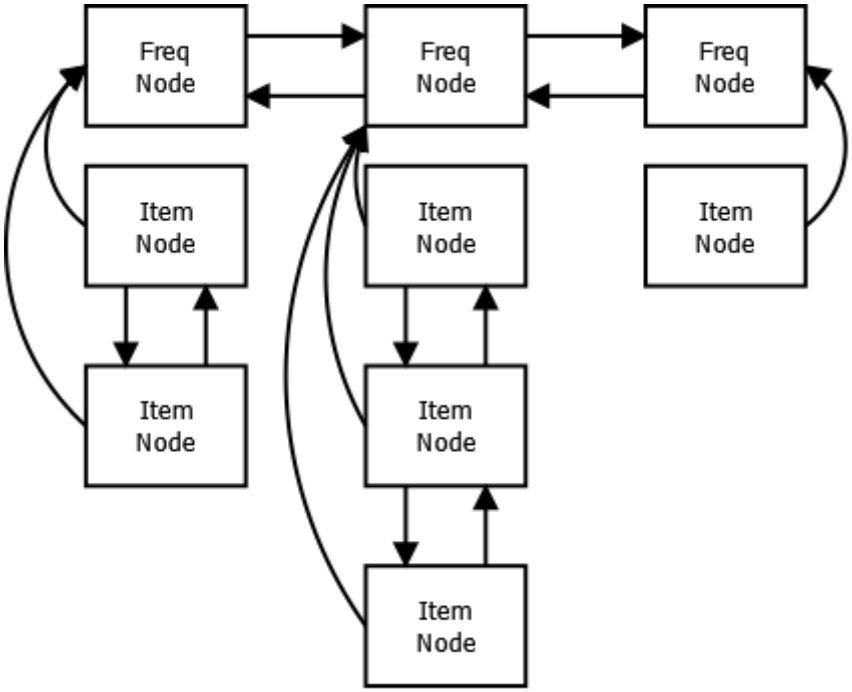
```
01 class DoublyLinkedList(object):
02     def __init__(self):
03         self.head = None
04         self.tail = None
05         # Number of nodes in list.
06         self.count = 0
07 
08     def add_node(self, cls, data):
09         """Add node instance of class cls."""
10         return self.insert_node(cls, data, self.tail, None)
11 
12     def insert_node(self, cls, data, prev, next):
13         """Insert node instance of class cls."""
14         node = cls(data)
15         node.prev = prev
16         node.next = next
17         if prev:
18             prev.next = node
19         if next:
20             next.prev = node
21         if not self.head or next is self.head:
22             self.head = node
```

```
        if not self.tail or prev is self.tail:
            self.tail = node
        self.count += 1
        return node

    def remove_node(self, node):
        if node is self.tail:
            self.tail = node.prev
        else:
            node.next.prev = node.prev
        if node is self.head:
            self.head = node.next
        else:
            node.prev.next = node.next
        self.count -= 1

    def get_nodes_data(self):
        """Return list nodes data as a list."""
        data = []
        node = self.head
        while node:
            data.append(node.data)
            node = node.next
        return data
```

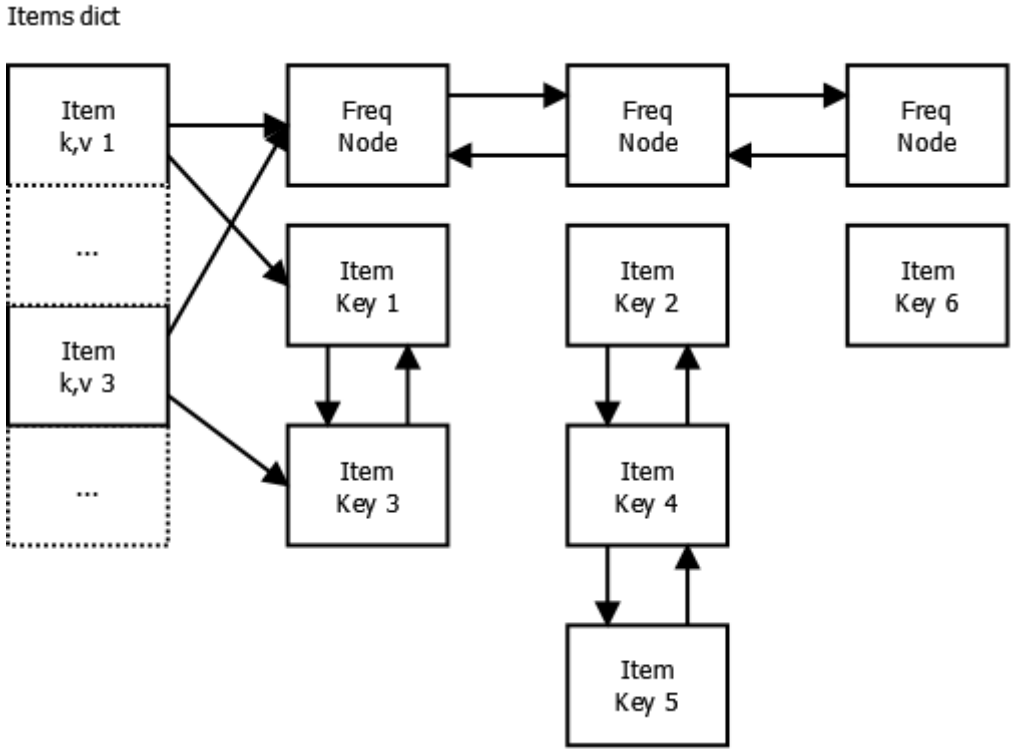
Each node in the access frequency doubly linked list is a frequency node (Freq Node on the diagram below). It is a node and also a doubly linked list containing the elements (Item nodes on the diagram below) of same frequency. Each item node has a pointer to its frequency node parent.




```
01 class FreqNode(DoublyLinkedList, Node):
02     """Frequency node containing linked list of item nodes with
03        same frequency."""
04     def __init__(self, data):
05         DoublyLinkedList.__init__(self)
06         Node.__init__(self, data)
07 
08     def add_item_node(self, data):
09         node = self.add_node(ItemNode, data)
10         node.parent = self
11         return node
12 
13     def insert_item_node(self, data, prev, next):
14         node = self.insert_node(ItemNode, data, prev, next)
15         node.parent = self
16         return node
17 
18     def remove_item_node(self, node):
19         self.remove_node(node)
20 
21 
22 class ItemNode(Node):
```

```
23      def __init__(self, data):
24          Node.__init__(self, data)
25          self.parent = None
```

The item node data is equal to the key of the element we are storing, an HTTP request could be the key. The content itself (HTTP response for example) is stored in a dictionary. Each value in this dictionary is of type LfuItem where “data” is the content cached, “parent” is a pointer to the frequency node and “node” is a pointer to the item node under the frequency node.




```
1  class LfuItem(object):
2      def __init__(self, data, parent, node):
3          self.data = data
4          self.parent = parent
5          self.node = node
```

We have defined our data objects classes, now we can define our cache object class. It has a doubly linked list (access frequency list) and a dictionary to contain the LFU items (LfuItem above). We defined two methods: one to insert a frequency node and one to remove a frequency node.

```
01  class Cache(DoublyLinkedList):
02      def __init__(self):
03          DoublyLinkedList.__init__(self)
```

```
        self.items = dict()

    def insert_freq_node(self, data, prev, next):
        return self.insert_node(FreqNode, data, prev, next)

    def remove_freq_node(self, node):
        self.remove_node(node)
```

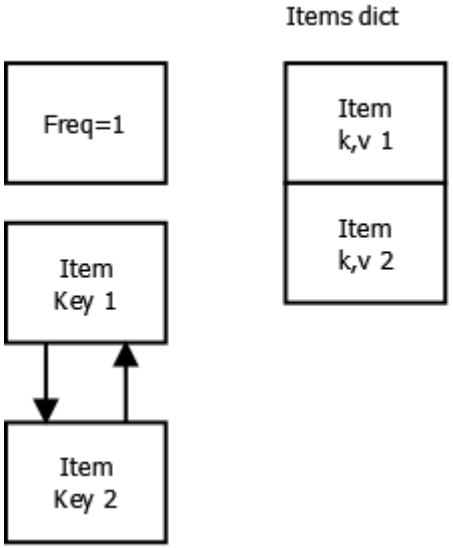
Next step is to define methods to insert to the cache, access the cache and delete from the cache.

Let's look at the insert method logic. It takes a key and a value, for example HTTP request and response. If the frequency node with frequency one does not exist, it is inserted at the beginning of the access frequency linked list. An item node is added to the frequency node items linked list. The key and value are added to the dictionary. Complexity is O(1).

```
def insert(self, key, value):
    if key in self.items:
        raise DuplicateException('Key exists')
    freq_node = self.head
    if not freq_node or freq_node.data != 1:
        freq_node = self.insert_freq_node(1, None, freq_node)

    freq_node.add_item_node(key)
    self.items[key] = LfuItem(value, freq_node)
```

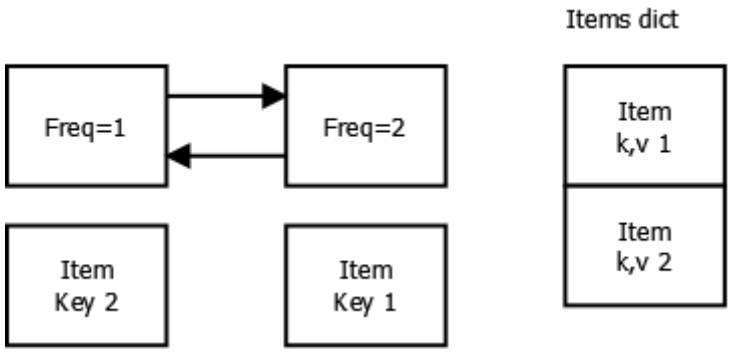
We insert two elements in our cache, we end up with:



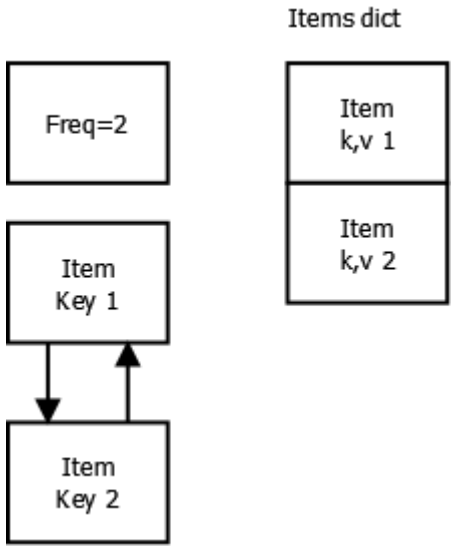
Let's look at the access method logic. If the key does not exist, we raise an exception. If the key exists, we move the item node to the frequency node list with frequency + 1 (adding the frequency node if it does not exist). Complexity is O(1).

```
01 def access(self, key):
02     try:
03         tmp = self.items[key]
04     except KeyError:
05         raise NotFoundException('Key not found')
06
07     freq_node = tmp.parent
08     next_freq_node = freq_node.next
09
10     if not next_freq_node or next_freq_node.data != freq_node.data + 1:
11         next_freq_node = self.insert_freq_node(freq_node.data + 1,
12             freq_node, next_freq_node)
13     item_node = next_freq_node.add_item_node(key)
14     tmp.parent = next_freq_node
15
16     freq_node.remove_item_node(tmp.node)
17     if freq_node.count == 0:
18         self.remove_freq_node(freq_node)
19
20     tmp.node = item_node
21     return tmp.data
```

If we access the item with Key 1, the item node with data Key 1 is moved to the frequency node with frequency equal to 2. We end up with:



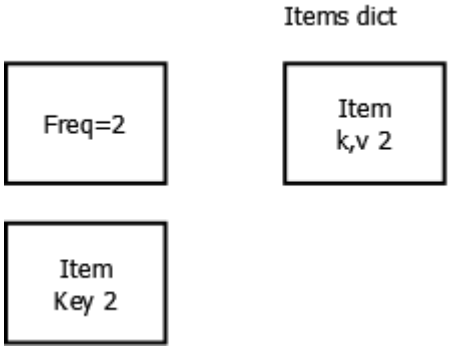
If we access the item with Key 2, the item node with data Key 2 is moved to the frequency node with frequency equal to 2. The frequency node 1 is removed. We end up with:



Let's look at the delete_lfu method. It removes the least frequently used item from the cache. To do that, it removes the first item node from the first frequency node and also the LFUItem object from the dictionary. If after this operation, the frequency node list is empty, it is removed.

```
01 def delete_lfu(self):
02     """Remove the first item node from the first frequency node.
03     Remove the item from the dictionary.
04     """
05     if not self.head:
06         raise NotFoundException('No frequency nodes found')
07     freq_node = self.head
08     item_node = freq_node.head
09     del self.items[item_node.data]
10     freq_node.remove_item_node(item_node)
11     if freq_node.count == 0:
12         self.remove_freq_node(freq_node)
```

If we call delete_lfu on our cache, the item node with data equal to Key 1 is removed and its LFUItem too. We end up with:



Github repo for the complete implementation.

tags: Python
posted in Uncategorized by Laurent Luce

Follow comments via the RSS Feed | Leave a comment | Trackback URL

Leave Your Comment

Name (required)

Mail (will not be published) (required)

Website

Post Comment

Powered by Wordpress and MySQL. Theme by Shlomi Noach, openark.org