When and Why to use a Least Frequently Used (LFU) cache with an implementation in Golang

February 27, 2019 · 16 min · Ilija Eftimov

▶ Table of Contents

Over the years, people involved in computer science and engineering have worked really hard on optimisations of various natures. Given that we live in a world with limited resources, humanity has always worked on ways to optimise the costs and speed literally everything.

In software engineering, I would argue, the most popular approach to performance improvement is caching. While there are various applications of caching, depending on the area of software engineering, the idea behind caching is quite simple: store data that is often needed/used in fast structure/storage so it can be retrieved very fast.

In fact, caches have to be fast along two dimensions:

1. Ensuring that as many of the requests for files go to it (cache hit), not over the network or to main memory (cache miss);
2. The overhead of using it should be small: testing membership and deciding when to replace a file should be as fast as possible.

In this article we will focus on the second part: taking a specific implementation approach to Least Frequently Used cache and making its membership test and eviction algorithms performant. Also, we'll also cover the basics and explore where such a caching scheme can be useful.

The basics

LFU is a caching algorithm in which the Least Frequently Used item in the cache is removed whenever the cache's capacity limit is reached. This means that for every item in our cache we have to keep track of how frequently it is used. Once the capacity is exceeded of the cache, an eviction algorithm will be run which has to pick and expire (remove) items from the cache.

If you have ever implemented an LFU cache you've probably looked into using a min-heap data structure because it handles insertion, deletion, and update in logarithmic time complexity. In this article, we will look at another approach to implementing it.

But before we go into the implementation, let's see in what cases LFU is better than the alternatives.

Where LFU shines

Imagine an asset cache on a CDN, where the assets are cached based on the usage patterns. So, when users request some images for the web page they are requesting, this CDN will add it to its cache for other users to get it even faster.

For example, one such image (asset) is the logo of the website. Can you imagine how many times a day Google's logo is requested across all of their products? I'd really like to find that number out, but for now, we can probably agree that the number is **HUGE**.

Such asset caches are the perfect use-case for LFU caches. An LFU cache eviction algorithm will never evict frequently accessed assets. In fact, in such a cache Google's logo will be cached virtually forever. In contrast, if there are any images that are going to be accessed due to a new landing page of a new product that's on front page of Reddit, Slashdot & Hackernews, once the hype-storm passes the assets will be evicted faster because the access frequency will drop dramatically, although in the past few days they have been accessed many times.

As you might already be noticing, this approach to cache eviction is very efficient in cases where the access patterns of the cached objects do not change often. While LRU caches will evict the assets that would not be accessed recently, the LFU eviction approach would evict the assets that are not needed any more after the hype has settled.

Implementing an LFU cache

Now, let's get to the meat of it. As we said before, instead of looking at a min-heap as a possible data structure that would back our LFU cache, we're going to look at a better approach.

In fact, in 2010, a group of researchers Prof. Ketan Shah, Anirban Mitra & Dhruv Matani published a paper titled "An O(1) algorithm for implementing the LFU cache eviction scheme" (you can check it here) in which they explain an implementation of an LFU cache that has a runtime complexity of `O(1)` for all of its operations, which include insertion, access and deletion (eviction).

Here, I'll show you how we can implement this cache and walk you through the implementation.

The data structure(s)

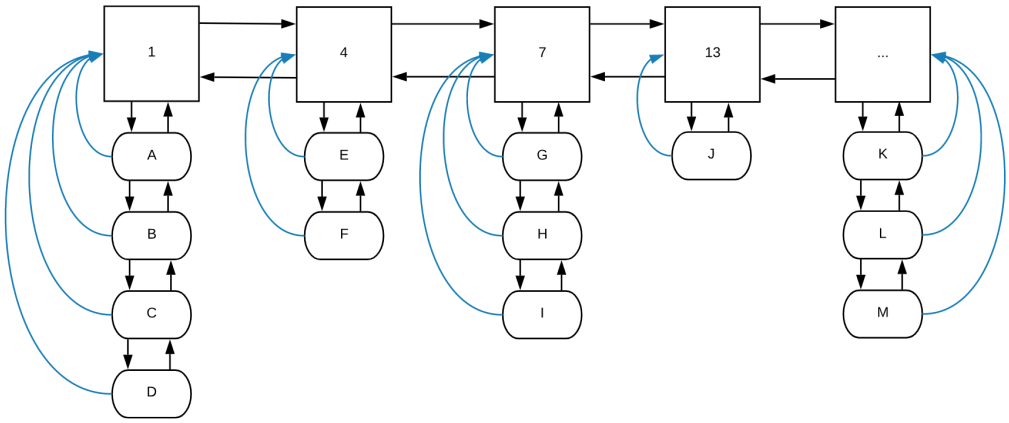
Nope, it's not going to be some sort of a Frankenstein red-black tree. It's, in fact, two doubly-linked lists and a hash table. Yes, that's all.

To be able to understand the fundamentals of this implementation of LFU, let's look at the linked list and hash table as graphics. Before we look at the actual graphics, we need to understand how the hash table and linked lists will be used.

The hash table will store all of the items with a key that is processed through a hashing algorithm (for our purpose we can keep it simple) and the value will be the actual item:

Key	Item
1221fa3d48178bcd16e5dee81302c21d	A
314f044e9b390a1c51698b3e5cd830a4	B
0521a78fffed8ed19891dd534f0df6c4	F
b27bb5876512cdb4591695551d00bab4	H
dbc35736633103720889a315568c1a41	D
4eb125794facce1eadde6824fe714a4a	G
be16485146790d53063bdd0a2263476a	C
b56abd93655ffa2eb9b7d1e19314d77b	E
fb401598525b0b822b3756d6a0a8904f	L
e9b9387ecc5173bfd5fe050521dbb277	K
d59beff405f6f171765608fb21cb8623	I
dc77be8c9ce980d51011f3310cd386e0	J
f252976b344702de796b0cf49014aae1	M

The linked lists are a bit more complicated. The first one will be the "frequency list", which will have all of the frequencies of access. Each of the nodes in this list will have an item list, which will contain all of the items that have been accessed with the corresponding frequency. Additionally, each of the items in the item list will have a pointer to their ancestor in the frequency list:



If we look at our graphical example above, we can notice that the items `A`, `B`, `C` and `D` have been accessed 1 time. The items `E` and `F` have been accessed 4 times and so on. The blue lines are the pointers that each of the items in the item lists has to their ancestor in the frequency list.

So, what would happen if item `E` gets accessed one more time? Let's go through the steps:

1. Retrieving the item from the hash table is easy (and scales well, `O(1)`)
2. We would access the item's `frequencyParent` pointer, from which we can check what is the next frequency in the list
3. If the new frequency is present (e.g. `8`), we will push it as the first item of the item list under frequency node `8`.
4. If the new frequency is not present, we will create the frequency node `8` and will add `E` to its item list

That's it. Retrieving the item and refreshing the frequency of the item is `O(1)`. Before we go into implementing the access algorithm, let's first establish the basic types that we would need to make this work.

Types

As we said earlier, we need to model the types required that will be the backbone of our cache.

The first `struct` will be the `CacheItem`. This will be the actual item that will be stored in the cache:

```
type CacheItem struct {
    key         string        // Key of entry
    value       interface{}   // Value of item
```

```
    frequencyParent *list.Element // Pointer to parent in cacheList
}
```

It contains the `key` by which we can look it up in the hash table, the `value` which is the actual cached item and a `frequencyParent` pointer to the pointer in the frequency list.

The next `struct` is the `FrequencyItem`, which represents each of the items in the frequency list. It contains a set of entries which will be a set of `CacheItem` pointers. We will use a `map` to store it so we can treat it as a set, which contains only unique items:

```
type FrequencyItem struct {
    entries map[*CacheItem]byte // Set of entries
    freq    int                 // Access frequency
}
```

The last `struct` that we will need to have a smooth running cache is, well, the `Cache` itself:

```
type Cache struct {
    bykey       map[string]*CacheItem // Hashmap containing *CacheItems for O(1) acc
    freqs       *list.List            // Linked list of frequencies
    capacity    int                   // Max number of items
    size        int                   // Current size of cache
}
```

The `Cache` will contain the hashmap, called `bykey` (the naming comes from the paper linked above), the frequency list called `freqs`, the maximum capacity of the cache called `capacity` and the `size` of the cache which represents the count of items cached at any given moment.

New, set & get

Let's look at the first three functions needed to make our cache work. The first one is a little constructor function:

```
func New() *Cache {
    cache := new(Cache)
    cache.bykey = make(map[string]*CacheItem)
    cache.freqs = list.New()
    cache.size = 0
    cache.capacity = 100

    return &c
}
```

The constructor `New` will create a new `Cache` struct and will set all the defaults to it. In case you're wondering how `list.New()` works: for the frequency list, we will use Go's `container/list` package, which contains a neat linked-list implementation. You can check its documentation for more details.

The second function, which will be implemented on the `Cache`, is the `Set` function:

```
func (cache *Cache) Set(key string, value interface{}) {
    if item, ok := cache.bykey[key]; ok {
        item.value = value
        // Increment item access frequency here
    } else {
        item := new(CacheItem)
        item.key = key
        item.value = value
        cache.bykey[key] = item
        cache.size++
        // Eviction, if needed
        // Increment item access frequency
    }
}
```

The function will take the cache key and the actual value/item to be cached as arguments. Then, it checks if the item is already cached or not. If it is cached, it will just update the value on the item. Otherwise, it will create a new `CacheItem` which will encapsulate the actual value, it will set the `key`, it will add the item to the `bykey` hashtable and it will increment the `size` of the cache.

Now, in both logical branches I have added some comments for the missing pieces:

1. `Cache` will have to know how to increment the access frequency for a `CacheItem`, but we are yet to implement that;
2. `Cache` will have to know how to evict an item based on the access frequency if the `size` reaches the `capacity`.

We will keep these comments in until we implement the `increment` and `evict` functions.

The third function that `Cache` will receive is `Get` - accessing the item by the key from the hashtable and returning it:

```
func (cache *Cache) Get(key string) interface{} {
    if e, ok := cache.bykey[key]; ok {
        // Increment acess frequency here
        return e.value
    }

    return nil
}
```

There's no magic here as well - we check if the `bykey` hashtable contains a value with the `key` argument and we return it if present. Otherwise, we return `nil`. Here, just like in `Set`, we will leave the placeholder comment where we have to add the frequency increment function call once we implement it.

Updating the access frequency

As we already saw, with every **access** action to the cache we have to update the access frequency of the accessed item.

Let's look at the steps that our `Increment` function would have to take. First, for the item to be expired we will have to decide if this item is already a part of the hash table and the frequency list or not. If it is, we will have to find out its new frequency value and its next frequency position (node) in the frequency list.

Second, we will have to figure out if for the new frequency there's already a node in the frequency list or not. If there is one, we will have to add the item to its list of `entries` and assign its new access frequency (which is the current access frequency + 1). If there's none, we will have to create a new frequency node in the frequency list (and set all of its sane defaults) and then add the item to its list of `entries`

Third, once we have a `FrequencyParent` detected, our function will have to set the new parent to the `item` that's being incremented and add it to the parent's list of `entries` .

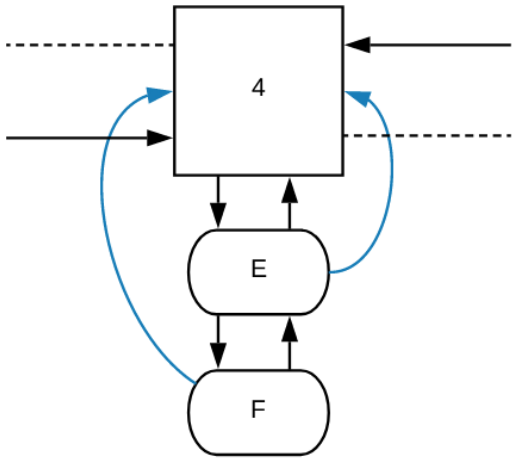
As the final step, the `increment` function will remove the item from the `entries` of the old frequency node (`frequencyParent`).

Here's the code in Golang:

```
func (cache *Cache) increment(item *CacheItem) {
    currentFrequency := item.frequencyParent
    var nextFrequencyAmount int
    var nextFrequency *list.Element

    if currentFrequency == nil {
        nextFrequencyAmount = 1
        nextFrequency = cache.freqs.Front()
    } else {
        nextFrequencyAmount = currentFrequency.Value.(*FrequencyItem).freq + 1
        nextFrequency = currentFrequency.Next()
    }

    if nextFrequency == nil || nextFrequency.Value.(*FrequencyItem).freq != nextFreq
        newFrequencyItem := new(FrequencyItem)
        newFrequencyItem.freq = nextFrequencyAmount
        newFrequencyItem.entries = make(map[*CacheItem]byte)
        if currentFrequency == nil {
            nextFrequency = cache.freqs.PushFront(newFrequencyItem)
        } else {
            nextFrequency = cache.freqs.InsertAfter(newFrequencyItem, currentFrequen
        }
    }

    item.frequencyParent = nextFrequency
    nextFrequency.Value.(*FrequencyItem).entries[item] = 1
    if currentFrequency != nil {
        cache.remove(currentFrequency, item)
```

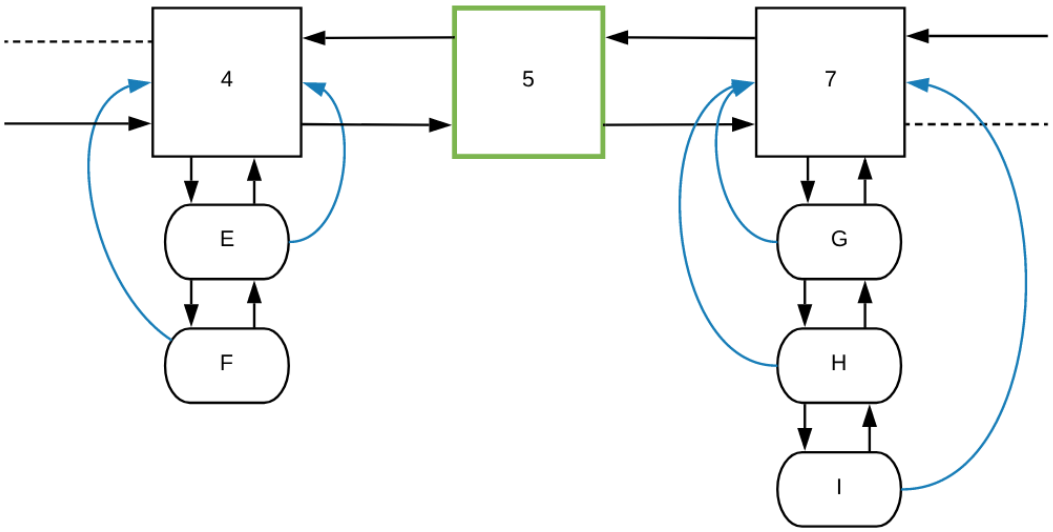
```
        }
    }
```

Let's refer back to our original diagram of the frequency and entries lists and walk through incrementing the `E` item.

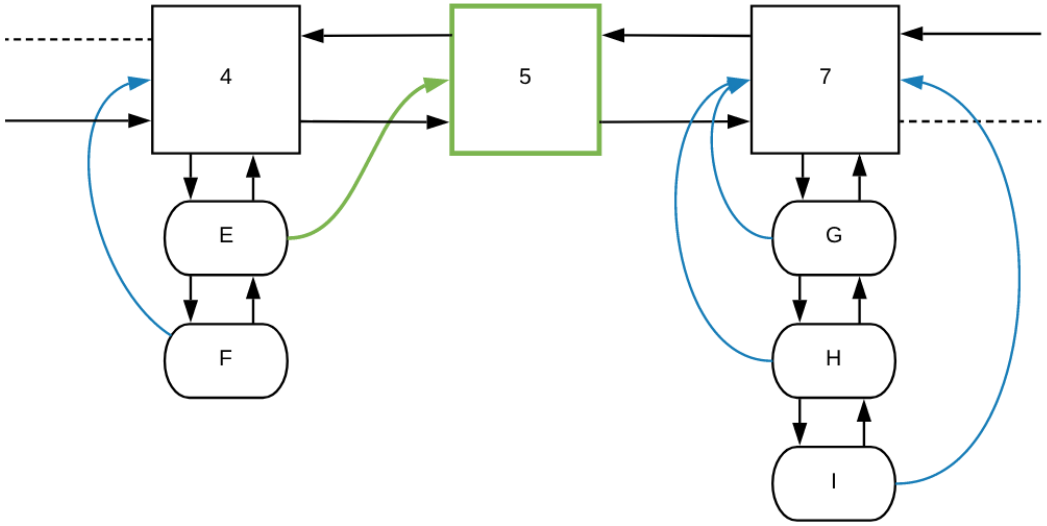


The first steps that our `increment` function will take are to allocate a pointer to node `4` (the `frequencyParent`) and its `value` (which is `4`). Since node `4` is present in the list, it will find the next node in the frequency list, which in our case is node `7`.

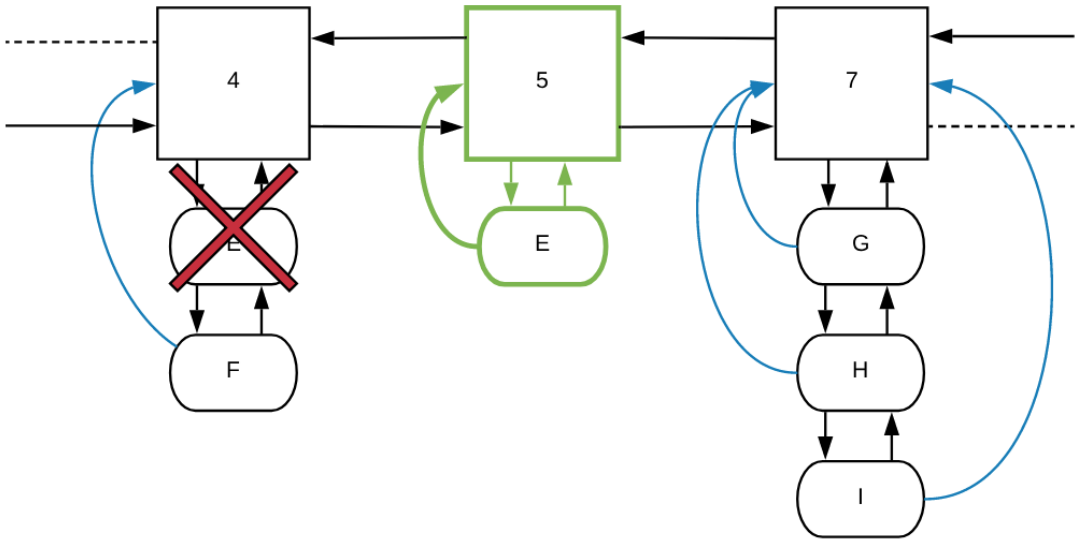
Once it figures out that the new frequency for the `E` node should be `5` and not `7`, it will append a new frequency node in the list, between nodes `4` and `7`:



Once the `5` node is added to the list, the function will set the defaults needed for the node to function properly. Then it will set `E`'s pointer to the new `frequencyParent` (the `5` node):



As the last step it will take the `item`, which has a `*CacheItem` type, and will add it to the `entries` list while deleting it from the `entries` list from the previous `frequencyParent`:



Let's look at what are the steps to remove a `CacheItem` from a `FrequencyItem`'s `entries` list.

Removing entries

Once we know the node in the list from which we want to remove it, we can just remove the item from the `entries` list and also completely remove the `FrequencyItem` from the frequency list if the `entries` become empty:

```
func (cache *Cache) Remove(listItem *list.Element, item *CacheItem) {
    frequencyItem := listItem.Value.(*FrequencyItem)
    delete(frequencyItem.entries, item)
    if len(frequencyItem.entries) == 0 {
        cache.freqs.Remove(listItem)
    }
}
```

Eviction

The last piece of the puzzle is eviction, or in other words, removing the least frequently used items once the cache reaches its maximum capacity.

We have to know how many items we want to evict. Usually, our cache would have a low and high bound, so when the high bound is reached we will remove items until the low bound. In our case, we will evict an arbitrary number of items, that the `Evict` function will take as an argument.

The function will start to "walk through" the frequency list from the beginning towards the end. Since the frequency list is in ascending order, it will start to remove the `entries` from the first frequency node onwards, until it removes as many items as the arbitrary number passed in.

If a frequency node contains no `entries` due to eviction, the `Evict` function will have to remove the frequency node from the frequency list as well. It will do that by invoking the `Remove` function. That way, the eviction will not leave any garbage behind.

Here's the code of what we described above:

```
func (cache *Cache) Evict(count int) {
    for i := 0; i < count; {
        if item := cache.freqs.Front(); item != nil {
            for entry, _ := range item.Value.(*FrequencyItem).entries {
                if i < count {
                    delete(cache.bykey, entry.key)
                    cache.Remove(item, entry)
                    cache.size--
                    i++
                }
            }
        }
    }
}
```

Back to Set and Get

At the beginning of this article we implemented the `Set` and `Get` functions. One thing that we didn't have available back then is the `Evict` and `increment` functions, so we can use them accordingly. Let's add their invocation.

Incrementing access frequency

In the `Get` function, if we find an item in the `bykey` hash table, we need to increment it's access frequency before we proceed to return its `value`:

```
func (cache *Cache) Get(key string) interface{} {
    if e, ok := cache.bykey[key]; ok {
        cache.increment(e)
        return e.value
    }

    return nil
}
```

With this change, the `Cache` will increment the frequency of that particular item before we return it. But, are we forgetting something? Also, the `Set` function makes access to the cached items when it actually caches them. This means that when an item is cached it will immediately be added to the frequency list, under the node with value `1`:

```
func (cache *Cache) Set(key string, value interface{}) {
    if item, ok := cache.bykey[key]; ok {
        item.value = value
                cache.increment(item)
    } else {
        item := new(CacheItem)
        item.key = key
        item.value = value
        cache.bykey[key] = item
        cache.size++
        // Eviction, if needed
                cache.increment(item)
    }
}
```

Eviction after addition

The `Set` function allows users of our LFU `Cache` to cache more items in it. A key component of any cache is that it should know how to evict items (free up space) when new items are added to the cache. For an LFU cache, the least frequently used items need to be removed when the cache is at capacity.

Let's first add a function that will return a `bool` if the cache has reached its maximum capacity:

```
func (cache *Cache) atCapacity() bool {
    return cache.size >= cache.capacity
}
```

The function is simple: checks if the current `size` of the `Cache` is bigger or equals than the `capacity`.

Now, let's put this into use in the `Set` function. Once we have a new item set in the cache, we have to check if the cache has reached its capacity and then evict a number of items from it.

For simplicity, we will remove only 10 items every time we reach max capacity:

```
func (cache *Cache) Set(key string, value interface{}) {
    if item, ok := cache.bykey[key]; ok {
        item.value = value
        cache.increment(item)
    } else {
        item := new(CacheItem)
        item.key = key
        item.value = value
        cache.bykey[key] = item
        cache.size++
        if cache.atCapacity() {
            cache.Evict(10)
        }
        cache.increment(item)
    }
}
```

With this change, if at any point adding an item reaches the capacity of the cache, the cache will evict the least frequently used items.

If you would like to see the whole code for this article, you can check out this gist.

Comments on scaling and time complexity

LFU is an interesting eviction scheme, especially when compared to LRU, in my opinion, due to its unconventional nature. While its application is limited, the algorithm and the backing data structures explained in the paper used in this article are fascinating due to the scaling ability of the approach.

If we revisit the paper that we mentioned at the beginning of the article, we will see that while LFU is not news, but it was traditionally implemented using a min-heap, which has a logarithmic time for insert, lookup and deletion. Interestingly in this paper, the authors explain that the approach they propose has an `O(1)` time complexity for each of the operations (insertion, lookup and deletion) because the operations are based on a hash table. Additionally, the linked lists do not add any time complexity because we do not traverse the lists at any point - we merely add or remove a node in them if needed (which is an O(1) operation).

In closing

In this article, we went through the basics of LFU caches. We established what are the most important performance metrics (hit to miss ratio, membership & eviction speed). We saw that while it's not the most widely used caching scheme, it sure can be very performant in some use cases.

Then we went on to implement it, using an approach that scales quite well in terms of time complexity. We saw the complexities of implementing the eviction and frequency incrementation algorithms. In the end, we explored some more how the approach we used to implement it scales.

If you would like to read more on the topic, here are a couple of links that will enrich your knowledge of LFU caches and caching in general:

- "An O(1) algorithm for implementing the LFU cache eviction scheme" - Prof. Ketan Shah, Anirban Mitra, Dhruv Matani
- "Caching in theory and practice" - Pavel Panchekha
- "LFU (Least Frequently Used) Cache Implementation"
  - Geeks for Geeks

lfu

« PREV PAGE
Barebones model of Spotify's 'Recently Played' screen using a Least Recently Used (LRU) cache in Golang

NEXT PAGE »
Golang Datastructures: Trees