ECGR 4090/5090 Cloud Native Application Architecture
Lab 2: Go Interfaces

Interfaces are used to separate behavior of a type from its concrete implementation. In C++ abstract classes serve this purpose. In Go, interfaces may be implemented by an arbitrary number of types without explicit inheritance (like in C++). And conversely, a type may implement an arbitrary number of interfaces.
Every type implements the empty interface (*interface{}*)

Example use of interface (From Tour of Go)

MyFloat and *Vertex* both implicitly implement the *Abser* interface since they both have the *Abs()* method

```
type Abser interface {
        Abs() float64
}

type MyFloat float64
// Note: Adding methods to predefined types such as float64

func (f MyFloat) Abs() float64 {
        if f < 0 {
                return float64(-f)
        }
        return float64(f)
}

type Vertex struct {
        X, Y float64
}

func (v *Vertex) Abs() float64 {
        return math.Sqrt(v.X*v.X + v.Y*v.Y)
}


func main() {
        var a Abser
        f := MyFloat(-math.Sqrt2)
        v := Vertex{3, 4}

        a = f  // a MyFloat implements Abser
        a = &v // a *Vertex implements Abser
```

```
        // Error: In the following line, vertex (the value type) doesn't implement Abser because
the Abs method is defined only on *Vertex (the pointer type).
        a = v
        fmt.Println(a.Abs())
}
```

Let's take a look at the use of interfaces to define abstract behavior of a cache. Here *Cacher* is an interface that defines two methods *Get()* and *Put()*. These use Go generics (introduced after Go 1.18; similar to C++ templates) to take in arguments of "comparable" type (i.e. constraint that comparisons can be performed), and any type (no constraints). See Generics in Go By Example.

lruCache is a concrete implementation of a cache. As long as *lruCache* implements the *Get()* and *Put()* methods, it satisfies the *Cacher* interface, and can be used anywhere where *Cacher* is used. Note that you could have another cache concrete type (perhaps, with different replacement policy) with the same methods that also satisfies the *Cacher* interface. The concrete implementation is thus separated from the behavior allowing the implementation to be changed without affecting the application that uses it (decoupling of implementation and use).

```
package cache

import "errors"

type Cacher[K comparable, V any] interface {
   Get(key K) (value V, err error)
   Put(key K, value V) (err error)
}

// Concrete LRU cache
type lruCache[K comparable, V any] struct {
   size int
   remaining int
   cache map[K]V
   queue []K
}

// Constructor
func NewCacher[K comparable, V any](size int) Cacher[K, V] {
   return &lruCache[K, V]{size: size, remaining: size, cache: make(map[K]V), queue: make([]K,
0)}
}

func (c *lruCache[K, V]) Get(key K) (value V, err error) {
```

```
        // Your code here ...

    // Hint - Move key to tail of queue (mark as recently used)

}

func (c *lruCache[K, V]) Get(key K) (value V, err error) {
        // Your code here ...

    // Hint - Move key to tail of queue (mark as recently used)

}

func (c *lruCache[K, V]) Put(key K, value V) (err error) {
        // Your code here ...

    // Hint - Check if key already exists


        // Hint - Check capacity, and evict if needed


    // Hint - Add new key-value pair


}

// Helper method to delete all occurrences of a key from the queue
func (c *lruCache[K, V]) deleteFromQueue(key K) {
    newQueue := make([]K, 0, c.size)
    for _, k := range c.queue {
        if k != key {
            newQueue = append(newQueue, k)
        }
    }
    c.queue = newQueue
}
```

To do-

Implement the *Get()* and *Put()* methods for *lruCache*.
Please find the following on Canvas - *lru.go lru_test.go*
Hint 1: Anytime you put or get an element from the map, it needs to be appended to the tail of the queue

Hint 2: The head of the queue is the last recently used element
Hint 3: When the cache is full, the element at the head of the queue is the victim that needs to be evicted from the cache, and also from the queue. It is possible that multiple copies of the element exist in the queue (due to multiple operations on this element), and so all these need to be deleted as well. I have provided an unexported helper method for deleting the victim element from the queue.