CSC 385
Assignment 5. Implement an LRU cache using Linked List

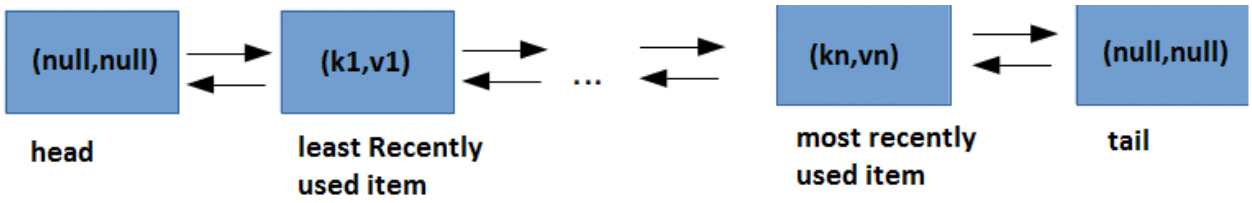
20 points

Instructions

When accessing a large amount of data from a slow medium such as disk or network, a common speed up technique is to keep a smaller amount of data into a more accessible location known as **cache**. LRU (Least Recently Used cache is a widely used cache algorithm for memory management). For example, a database system may keep data cached in memory so that it doesn't have to read the hard drive. Or a web browser might keep a cache of web pages on the local machine so that it doesn't have to download them over the network.

In general, a cache is much too small to hold all the data you might possibly need, so at some point you are going to have to remove something from the cache in order to make room for new data. The goal is to retain those items that are more likely to be retrieved again soon. This requires a sensible algorithm for selecting what to remove from the cache. One simple but effective algorithm is the Least Recently Used, or LRU, algorithm. When performing LRU caching, you always throw out the data that was least recently used. The LRU cache is based on the idea that items that are most recently used are more likely to be used again in future.

In this assignment, you are to implement a very basic Least Recently Used (LRU) cache using LinkedList. The Linked list has two dummy nodes: head and tail which hold references to the beginning and end of the list, respectively.



Every node in the linked list represent a cached item which consists of a key and a value. Duplicate keys are not allowed in the cache. The nodes are accessed by their keys. The first node (after head) is the node that is least recently accessed and the last node (before tail) is the node that is most recently accessed. When a user accesses a key, the node which contains that key is moved to the end of the list (before tail).

A cache has a limited capacity. When a new entry is added to the cache it is added to the end of the list. If the cache is full, the least recently used item (the first node after head) must be discarded (removed) from the list to make room for the new entry.

I have provided the framework of the class you need to implement "LRULinkedCache.java". This class has the following attributes:

- **theSize**: The current number of items in the cache
- **capacity:** The capacity of the cache (the maximum number of items which can be stored in the cache)
- **head**: reference to the head node
- **Tail**: reference to the tail node

All you need to do is implementing three parts of this class:

1. **The nested `CacheNode<K,V>` class**: This class encapsulates the building block of a cache node. It contains a key and value, as well as a reference to both the next and previous nodes in the list. K is a parametric (generic) type for the key and V is a parametric type for the value.
2. **LRUGet(K key):** This method returns the value for a given key in the cache and moves the node which contains the key to the end of the list (because it is most recently accessed). The method returns null if there is no node with the given key.

3. **LRUPut(K key, V value)**: This method adds a new node with the given key and value to the end of the list. If the cache is full, the least recently used node (the first node after head) is removed to make room for new node. Since duplicate keys are not allowed, the method must first check to see if a node with a given key already exists in the cache and if so, it updates its value and move the node to the end of the list.

Please follow these guidelines when implementing your class:

1. Do not extend any of the data structure classes or interfaces I provided in the "source code" (doing so will be more trouble than it is worth). You do not need to use any other file beyond the provided LRULinkedCache.java file.

2. You can add any private helper method you want; but your code will be graded based on the LRUGet and LRUPut methods.

3. The LRULinkedCache.java in its current form is not compiled as its missing the implementations requested above. After you add your implementations you can compile and test your class. For example, if you use the following main method:

```
public static void main(String[] args)
{
    LRULinkedCache<Integer, Integer> cache = new LRULinkedCache<Integer,Integer>(4);
    cache.LRUPut(1,5);
    System.out.println("cache after calling LRUPUT(1,5): "+ cache.toString());
    cache.LRUPut(2, 2);
    System.out.println("cache after calling LRUPUT(2,2): "+ cache.toString());
    cache.LRUPut(3, 7);
    System.out.println("cache after calling LRUPUT(3,7): "+ cache.toString());
    cache.LRUPut(4, 9);
    System.out.println("cache after calling LRUPUT(4,9): "+ cache.toString());
    cache.LRUPut(1,9);
    System.out.println("cache after calling LRUPUT(1,9): "+ cache.toString());
    System.out.println("LRUGET(3) returned: " + cache.LRUGet(3));
    System.out.println("cache after calling LRUGET(3): "+ cache.toString());
    cache.LRUPut(5, 10);
    System.out.println("cache after calling LRUPUT(5,10): "+ cache.toString());
    cache.LRUGet(4);
    System.out.println("LRUGET(4) returned: " + cache.LRUGet(4));
    System.out.println("cache after calling LRUGet(4): "+ cache.toString());
    cache.LRUGet(10);
    System.out.println("cache after calling LRUGet(10): "+ cache.toString());

}
```

Your program must produce the following output:

```
cache after calling LRUPUT(1,5): (1,5)
cache after calling LRUPUT(2,2): (1,5),(2,2)
cache after calling LRUPUT(3,7): (1,5),(2,2),(3,7)
cache after calling LRUPUT(4,9): (1,5),(2,2),(3,7),(4,9)
cache after calling LRUPUT(1,9): (2,2),(3,7),(4,9),(1,9)
LRUGET(3) returned: 7
cache after calling LRUGET(3): (2,2),(4,9),(1,9),(3,7)
cache after calling LRUPUT(5,10): (4,9),(1,9),(3,7),(5,10)
LRUGET(4) returned: 9
cache after calling LRUGet(4): (1,9),(3,7),(5,10),(4,9)
cache after calling LRUGet(10): (1,9),(3,7),(5,10),(4,9)
```

Notice when LRUPput(1,9) is called, since a node with key=1 already exists in the cache, its value is updated to 9 and the node (1,9) is moved to the end of the list. When LRUGet(3) is called the value for key=3 is returned and the node (3,7) is moved to the end of the list.

When LRUPut(5,10) is called the cache has reached its capacity, so the least recently used node (2,2) is removed from the cache and (5,10) is added to the end of the list. When LRUGet(10) is called the cache is remained unchanged as there is no node with key=10 in the cache.

What you need to turn in:

1. Turn in your `LRULinkedCache.java` containing the implementations described above In addition, you should submit an extra document explaining (Big-Oh) the time efficiency of the LRUPut and LRUGet method.

Grading Rubric

CacheNode<K,V> inner class	3
LRUGet(K key) method	7
LRUPut(K key, V value) method	7
Explaining the time efficiency of LRUPut and LRUGet methods	3
Total:	**20**