(3/3/2025)

LRU - Least Recently Used

~ Het Jasani
2nd Year B. Tech

* LRU - Least Recently Used

- mainly used on cache Replacement
- removes the ~~latest~~ least recently used item when cache reaches the capacity

* Working :

-> Cache stores a fixed number of item
-> when a new item is accessed
  - if already exist
    then move to mast recent
  - if not
    then add
  - if cache is full [least]
    remove the recent used

* Example (capacity = 3)

sequence; 1, 2, 3, 4, 2, 5, 1

Step		Cache	Exp
1		1	Ins 1
2		2 -> 1	Ins 2
3		3 -> 2 -> 1	Ins 3
4		4 -> 3 -> 2	Ins 4
			Rem 1
5		2 -> 4 -> 3	Moved 2
6		5 -> 2 -> 4	Rem 3
			Ins 5
7		1 -> 5 -> 2	Rem 4
			Ins 1

* LRUCache.Java

```
import java.util.HashMap;

class LRUCache {
    private class Node {
        int key, value;
        Node prev, next;

        Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private final int capacity;
    private final HashMap<Integer, Node> cache;
    private final Node head, tail;

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
        this.head = new Node(-1, -1);
        this.tail = new Node(-1, -1);

        head.next = tail;
        tail.prev = head;

    }
```

```
public int get (int key) {
    if (!cache.containsKey(key))
        return -1;
    else {
        Node node = cache.get(key);
        moveToHead(node);
        return node.value;
    }
}

public void put(int key, int value){
    if( cache.containsKey(key)) {
        Node node = cache.get(key);
        node.value = value;
        moveToHead(node);
    }
    else {
        Node newNode = new Node(key, value);

        cache.put(key, newNode);
        addToHead(newNode);

        if( cache.size() > capacity) {
            removeLRU();
        }
    }
}
```

```
private void moveToHead(Node node) {
    removeNode(node);
    addToHead(node);
}

private void addToHead(Node node){
    node.next = head.next;
    node.prev = head
    head.next.prev= node
    head.nex = node;
}

private void removeNode(Node node){
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void removeLRU() {
    Node lru = tail.prev;
    removeNode(lru);
    cache.remove(lru.key);
}

public static void main(String[] args){
    LRUCache lru = new LRU(cache(3);
    lru.put(1,10);
    lru.put(2,20);
    lru.put(3,30);
    lru.print Cache();
    lru.get(1);
    lru.print Cache();
    lru.put(4,40);
    lru.print Cache();
}
```

```
public void printCache() {
    Node current = head.next;
    System.out.print("Cache");
    while (current != tail) {
        System.out.print(
            current.key + ":" + current.value
        );
        current = current.next;
    }
    System.out.println();
}

}
```

* Class Structure :

Node
int capacity
HashMap cache
Node head, tail
} Data Members

LRUCache() } constructor

get()
put()
printCache()
} public functions

moveToHead()
removeNode()
removeLRU()
addToHead()
} Private functions