LRU Cache:

Time Complexity: 0(1)

Space Complexity: 0(n)

Run on leet-code: No, implmented in jupyter-notebook with certain base-cases missing

In [19]:
```python
# create a node
class Node:

    def __init__(self,key,value):
        self.key = key
        self.value = value
        self.next = None
        self.previous = None
```

In [95]:
```python
# create a doublyLinkedListCache
class dllCache:

    def __init__(self):
        self.head = None
        self.tail = None
        self.cacheDict = {}

    # insert into cache
    def insertCache(self,key,value):

        # 1. create an obj of class node
        objNewNode = Node(key,value)

        # 2. key-node pair into the cacheDict
        self.cacheDict[key] = objNewNode

        # 3. chk if its the first node
        if self.head == None and self.tail == None:
            self.head = objNewNode
            self.tail = objNewNode
            return

        # 4. else case -- insert the node from the head
        objNewNode.previous = self.head
        self.head.next = objNewNode
        self.head = objNewNode

    # update into the cache
    def updateCache(self,key,val):

        '''To check for certain base cases i.e. related with tail and head ptr's (di

        # 1. get the rfr from the cache
        objNewNode = self.cacheDict[key]

        # 2. update the node.value
        objNewNode.value = val

        # 3. update the reference

        # 3.1. update the rfr's for the head
        self.head.next = objNewNode
        self.head.previous = objNewNode.previous
```

```
            # 3.2. update the rfr's for objNewNode.previous
            objNewNode.previous.next = self.head

            # 3.3. update objNewNode rfr's
            objNewNode.next = None
            objNewNode.previous = self.head

            # 3.4. update the head
            self.head = objNewNode

        # delete LRU from the cache -- delete tail
        def deleteCache(self):

            '''To check for certain base cases i.e. related with tail and head ptr's (di

            # 1. get the key for the tail
            key = self.tail.key

            # 2. delete from the cacheDict
            del self.cacheDict[key]

            # 3. delete the node from the dll
            objDelNode = self.tail

            self.tail = self.tail.next
            self.tail.previous = None

            objDelNode.next = None
            objDelNode = None
```

In [96]:

```
# implement class LRU cache
class LRUCache:

    def __init__(self, capacity):
        # initialize capacity
        self.capacity = capacity
        self.count = 0

        # initlize dllCache
        self.lruCache = dllCache()

    def get(self, key):

        # 1. get the value from the cacheDict
        value = self.lruCache.cacheDict[key].value

        # 2. perform the update of node position
        self.lruCache.updateCache(key,value)

        # 3. return the calue
        return value

    def put(self, key, value):

        # A. fresh-node
        if key not in self.lruCache.cacheDict:

            # Insert into LRU cache
            self.lruCache.insertCache(key,value)
            self.count += 1

            # check for capcity
            if self.count > self.capacity:
                # delete the least recnetly used ---> remove from tail
                self.lruCache.deleteCache()
                self.count -= 1
            return

        # B. non-fresh-node
        else:

            # Update into LRU cache
            self.lruCache.updateCache(key,value)
            # count won't be changed
            return
```

In [97]:
```
lru = LRUCache(5)
```

Upsert into LRU Cache

In [98]:
```
lru.put(1,1)
lru.put(2,2)
lru.put(3,3)
lru.put(4,4)
lru.put(5,5)
```

In [99]:
```
# print the node
lru.lruCache.cacheDict
```

Out[99]:
```
{1: <__main__.Node at 0x23f7924ae20>,
 2: <__main__.Node at 0x23f7924a040>,
 3: <__main__.Node at 0x23f7924a1c0>,
 4: <__main__.Node at 0x23f7924abe0>,
 5: <__main__.Node at 0x23f7924a550>}
```

In [100…
```
lru.put(3,33)
```

In [101…
```
# print the node
lru.lruCache.cacheDict
```

Out[101…
```
{1: <__main__.Node at 0x23f7924ae20>,
 2: <__main__.Node at 0x23f7924a040>,
 3: <__main__.Node at 0x23f7924a1c0>,
 4: <__main__.Node at 0x23f7924abe0>,
 5: <__main__.Node at 0x23f7924a550>}
```

In [102…
```
lru.lruCache.cacheDict[3].value
```

Out[102…
```
33
```

In [103…
```
lru.put(6,6)
```

In [104…
```
# print the node
lru.lruCache.cacheDict
```

Out[104…
```
{2: <__main__.Node at 0x23f7924a040>,
 3: <__main__.Node at 0x23f7924a1c0>,
 4: <__main__.Node at 0x23f7924abe0>,
 5: <__main__.Node at 0x23f7924a550>,
 6: <__main__.Node at 0x23f7924a4f0>}
```

Get the value from LRU Cache

In [105…

```
lru.get(4)
```

Out[105…

```
4
```

In [106…

```
# print the node
lru.lruCache.cacheDict
```

Out[106…

```
{2: <__main__.Node at 0x23f7924a040>,
 3: <__main__.Node at 0x23f7924a1c0>,
 4: <__main__.Node at 0x23f7924abe0>,
 5: <__main__.Node at 0x23f7924a550>,
 6: <__main__.Node at 0x23f7924a4f0>}
```

In [107…

```
vars(lru.lruCache.head)
```

Out[107…

```
{'key': 4,
 'value': 4,
 'next': None,
 'previous': <__main__.Node at 0x23f7924a4f0>}
```