Design LRU (Least Recently Used) Cache :

public int get(key)

TimeComplexity : O(1)

public void add(key, value)

LRUCache size is fixed, if cache reaches the capacity, we would need to remove the "least recently used element" while adding the new element to the Cache.

public LRUCache(int capacity) :

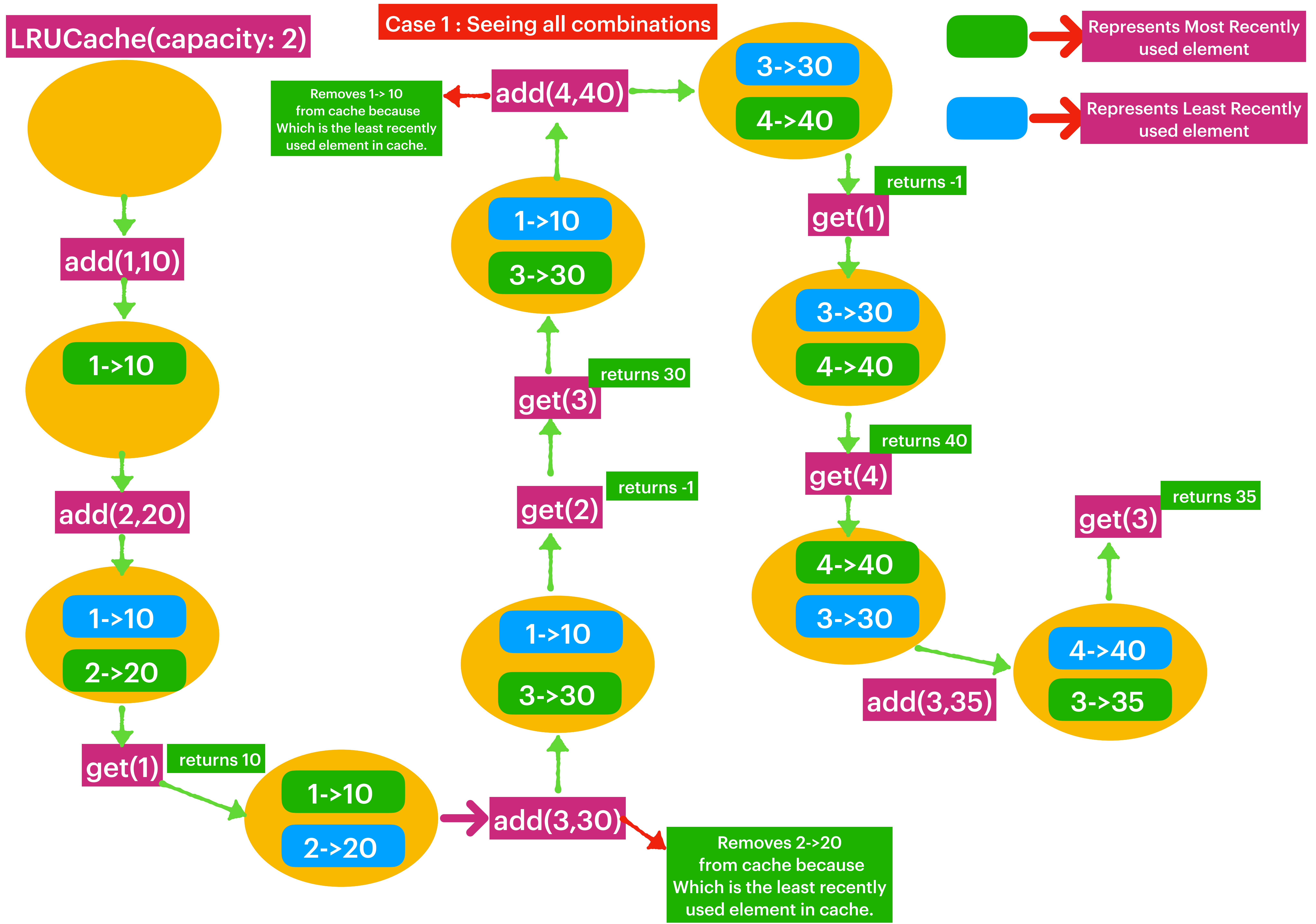
LRUCache has the fixed capacity

public void add(int key, int value) :

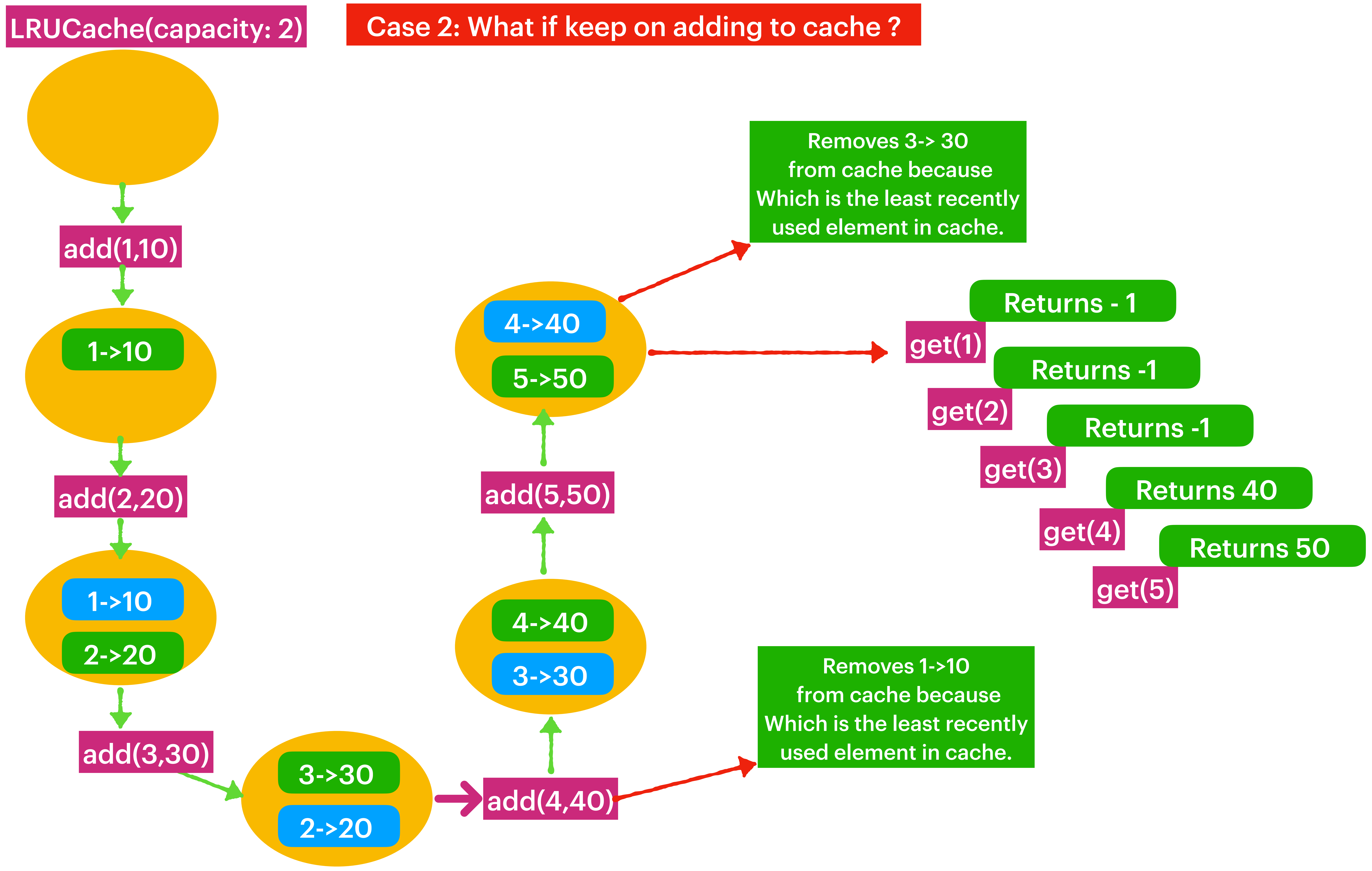
Adds the element to LRUCache.

public int get(int key) :

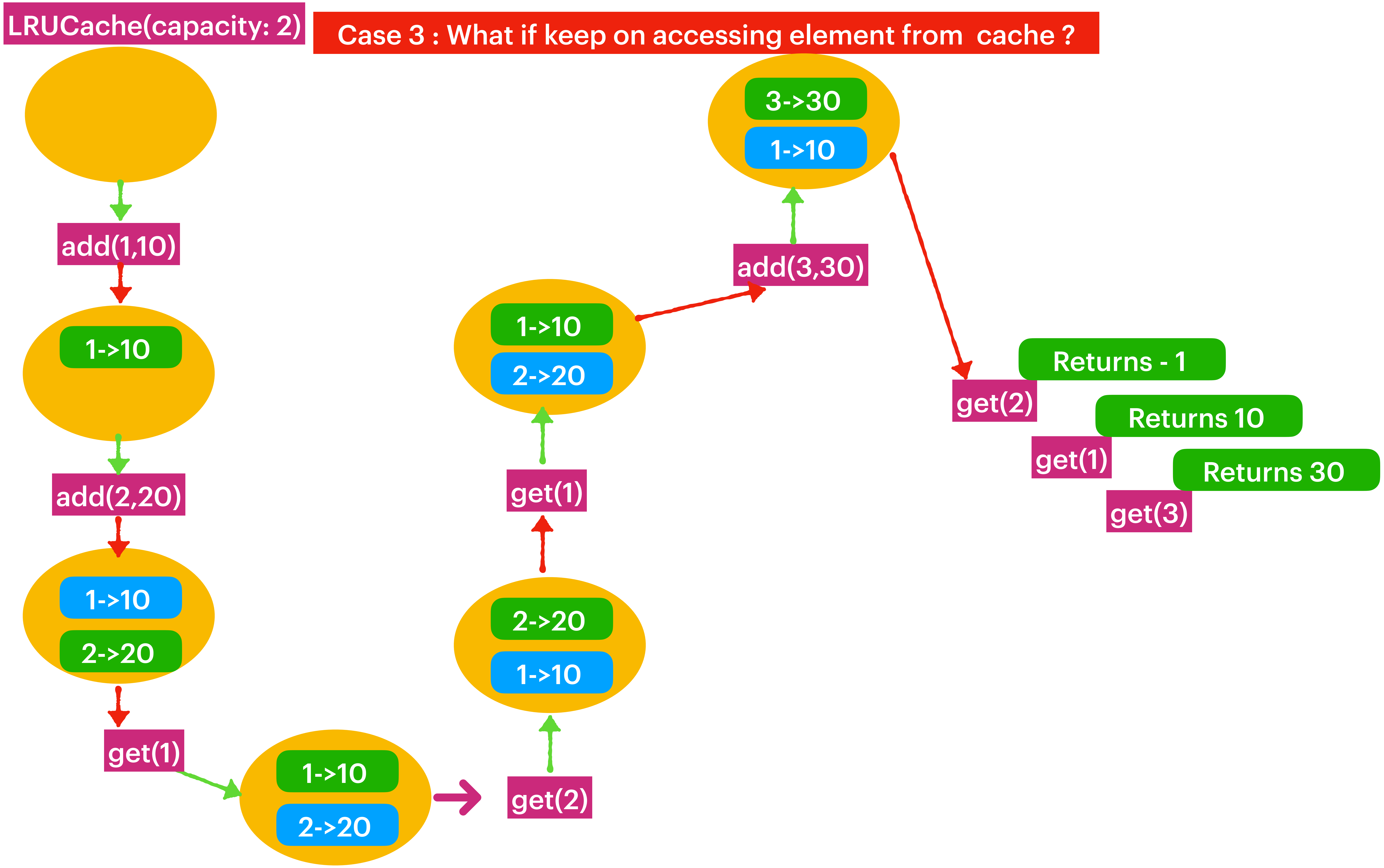
Returns value if the key presents otherwise returns -1


LRUCache(capacity: 2)
Case 1 : Seeing all combinations
Represents Most Recently used element
Represents Least Recently used element
add(1,10)
1->10
add(2,20)
1->10
2->20
get(1)
returns 10
1->10
2->20
add(3,30)
Removes 2->20 from cache because Which is the least recently used element in cache.
1->10
3->30
get(2)
returns -1
get(3)
returns 30
1->10
3->30
add(4,40)
Removes 1-> 10 from cache because Which is the least recently used element in cache.
3->30
4->40
get(1)
returns -1
3->30
4->40
get(4)
returns 40
4->40
3->30
add(3,35)
4->40
3->35
get(3)
returns 35

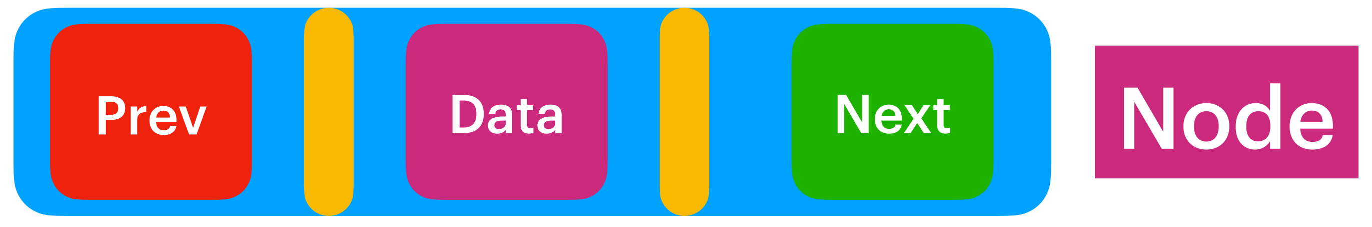
Case 2: What if keep on adding to cache ?



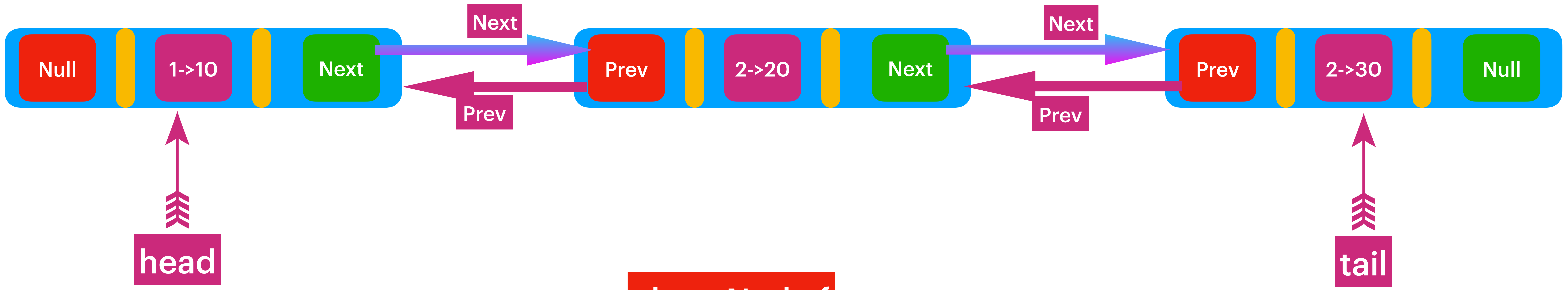
Case 3 : What if keep on accessing element from cache ?



Double Linked List



Double Linked List has the reference of nextNode and its previous Node. So that we can traverse both in forward and reverse directions. Double Linked List simply fees the insert & delete operations.



```
class  Node {
int data,
int value,
Node next;
Node prev;
}
```

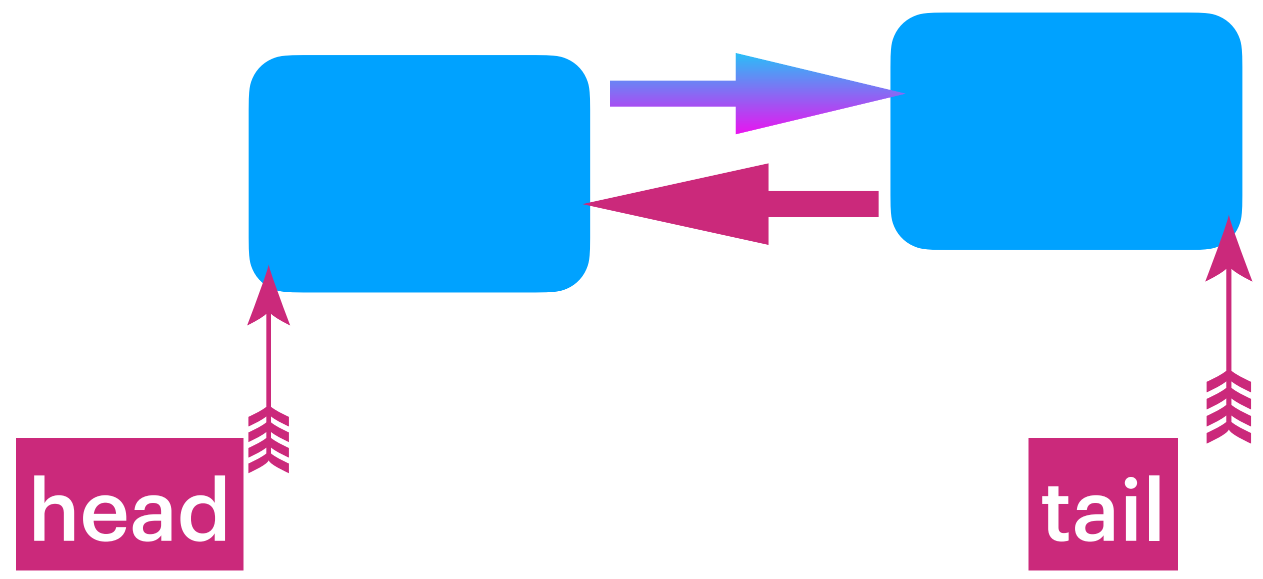
To avoid null checks maintain , take head & tail dummy nodes !!!

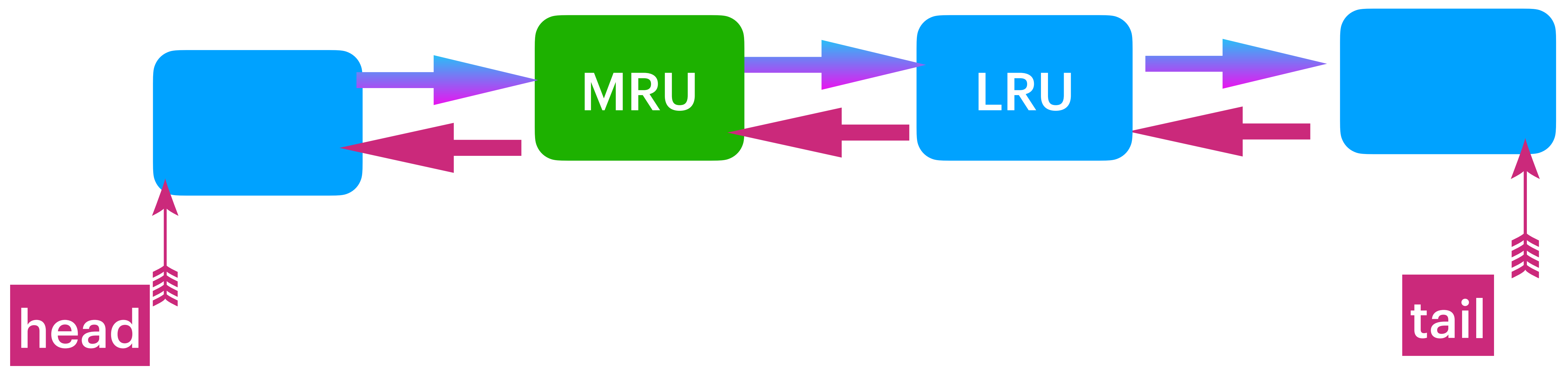
MRU : Most Recently Used Element

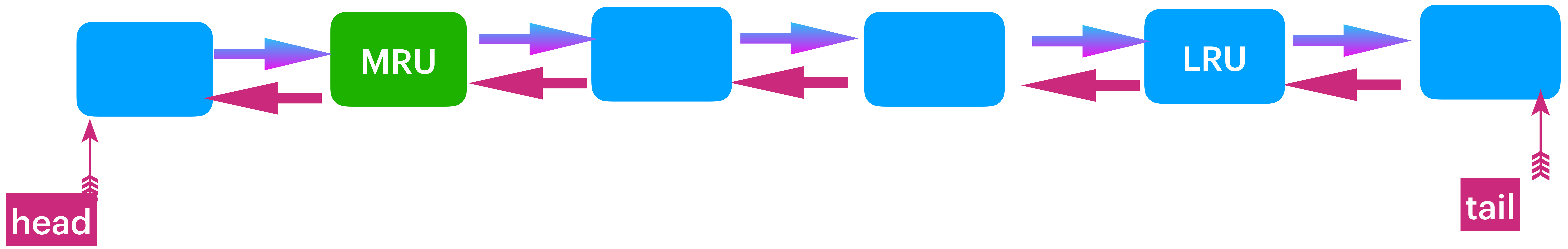
LRU : Least Recently Used Element

To maintain O(1) TimeComplexity
Use Map<key,Node> with DoubleLinkedList
for add and get

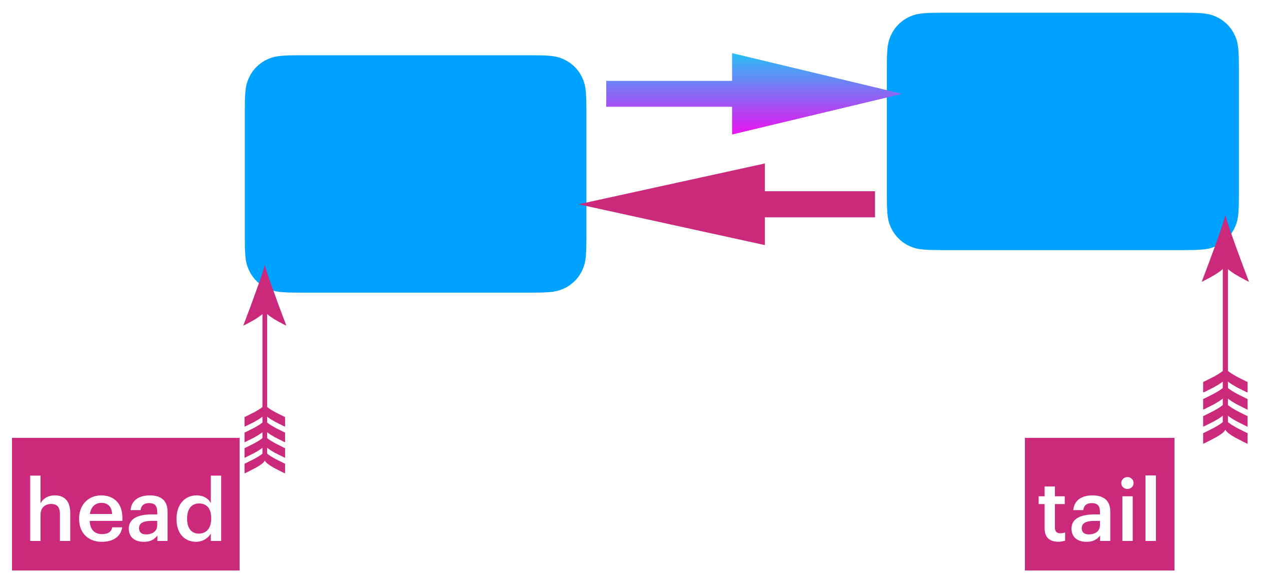
Always add new node to right after head !!!
It represents most recently used element.







To avoid null checks maintain , take head & tail dummy nodes !!!



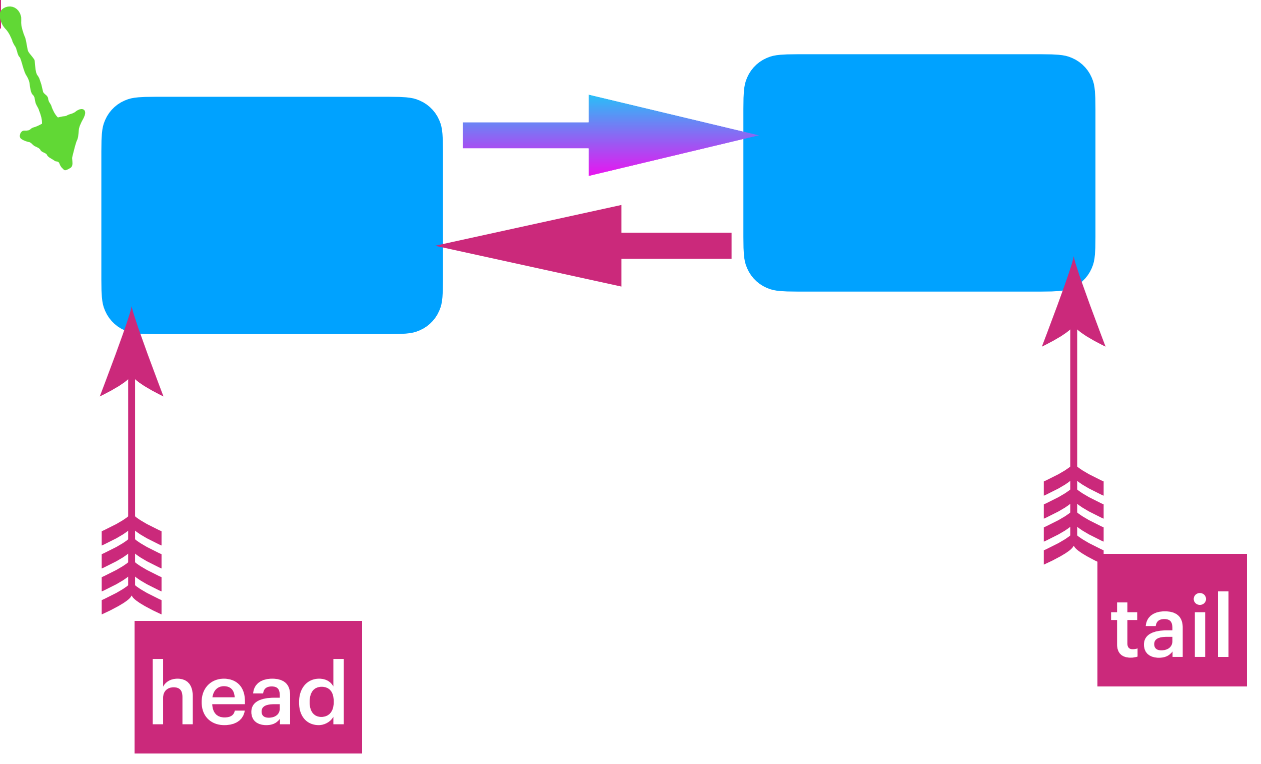
To maintain O(1) TimeComplexity
Use Map<key,Node> with DoubleLinkedList
for add and get

Always add new node to right after head !!!
It represents most recently used element.

add(1,10)

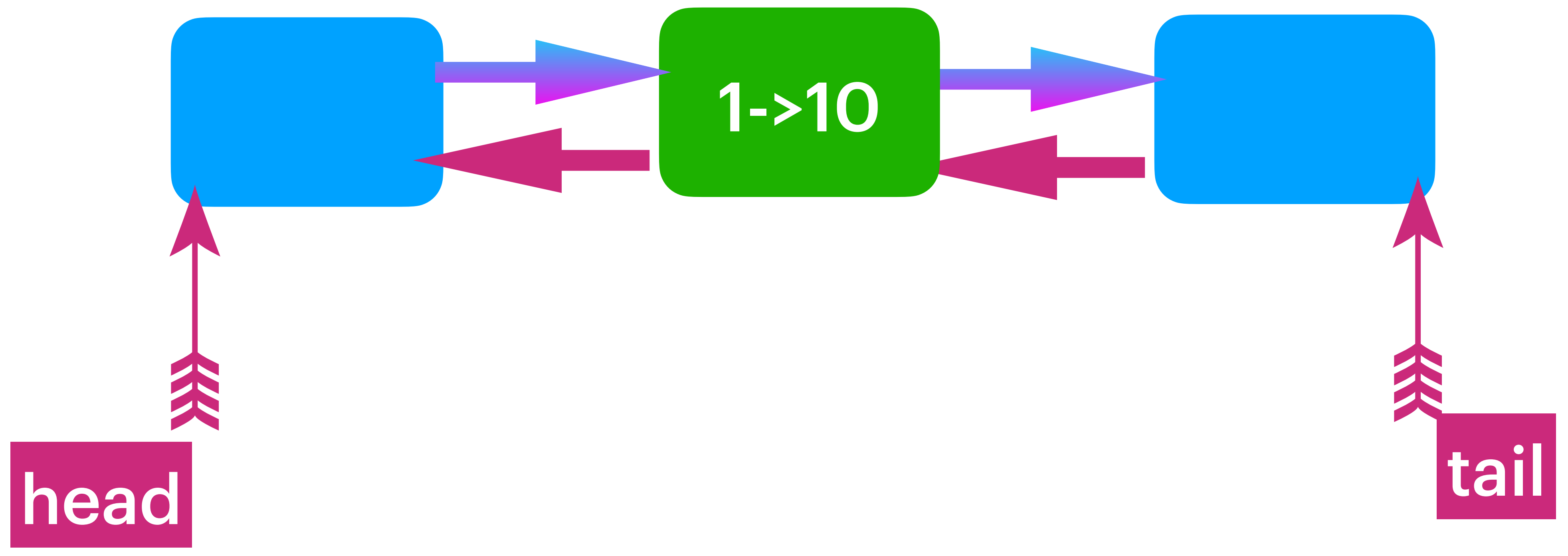
Case 1: If the node does exist
addToHead & after add If the size > capacity,
remove tail.prev.

Case 2: If the node is present then
Update the value moveToHead.



LRUCache : Capacity(2)

Node(1->10) does not exist so add right to the head

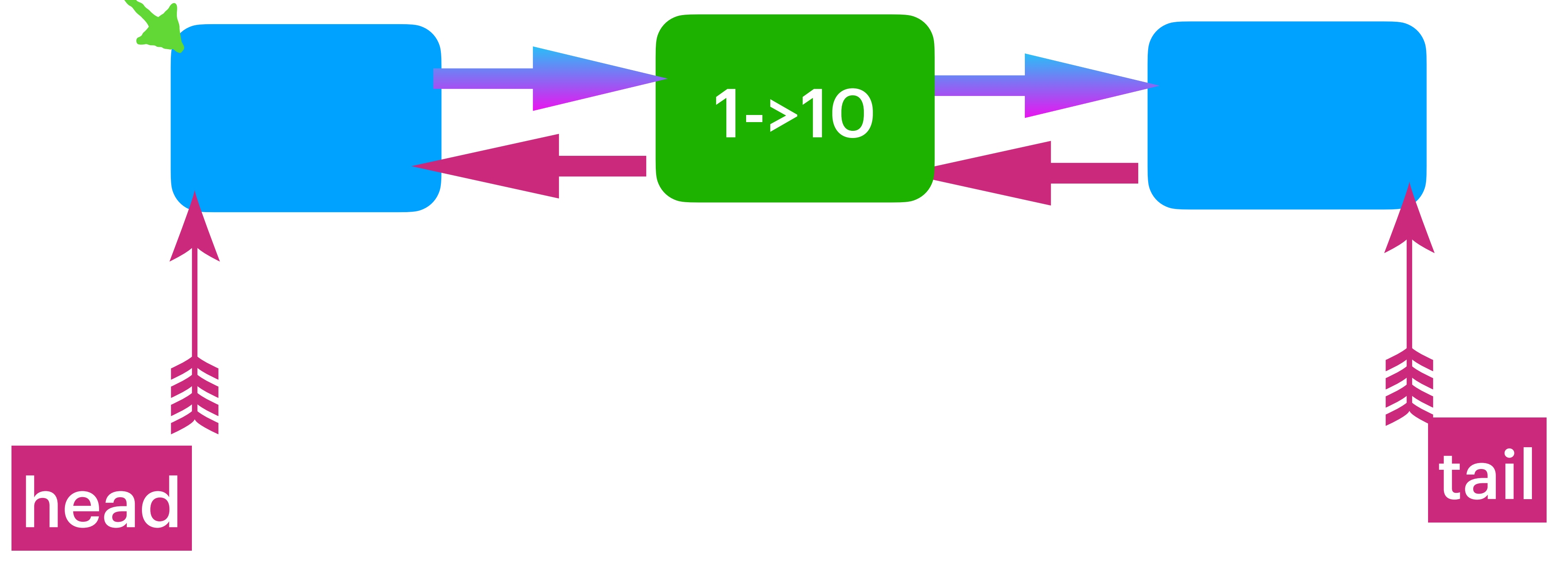


LRUCache : Capacity(2)

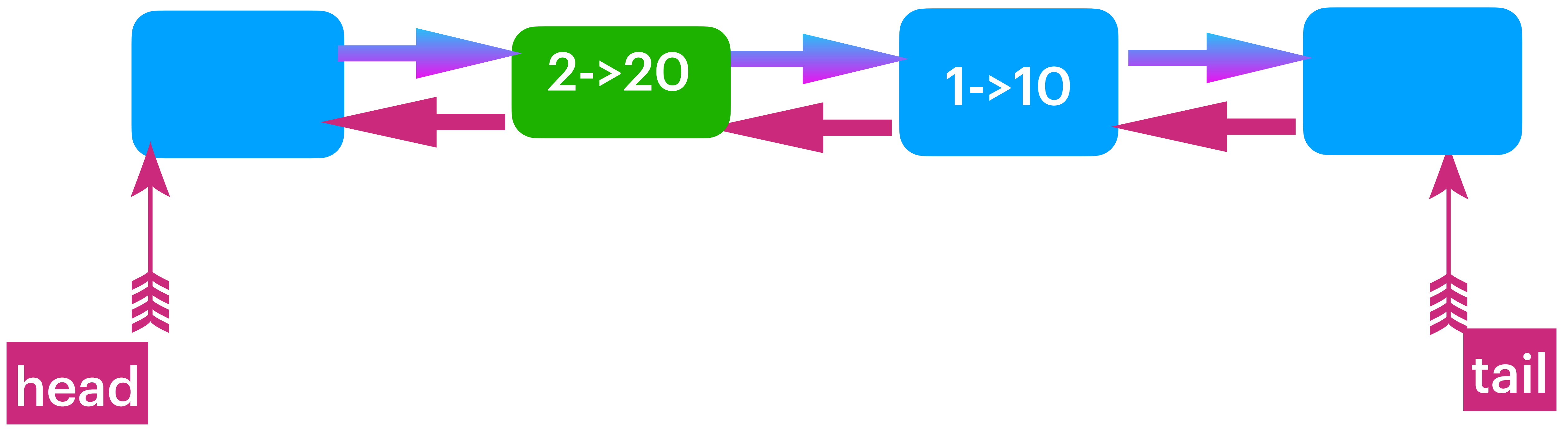
add(2,20)

Case 1: If the node does exist addToHead & If the size > capacity, remove tail.prev.

Case 2: If the node is present then Update the value moveToHead.



Node(2->20) does not exist so add to Right to the head

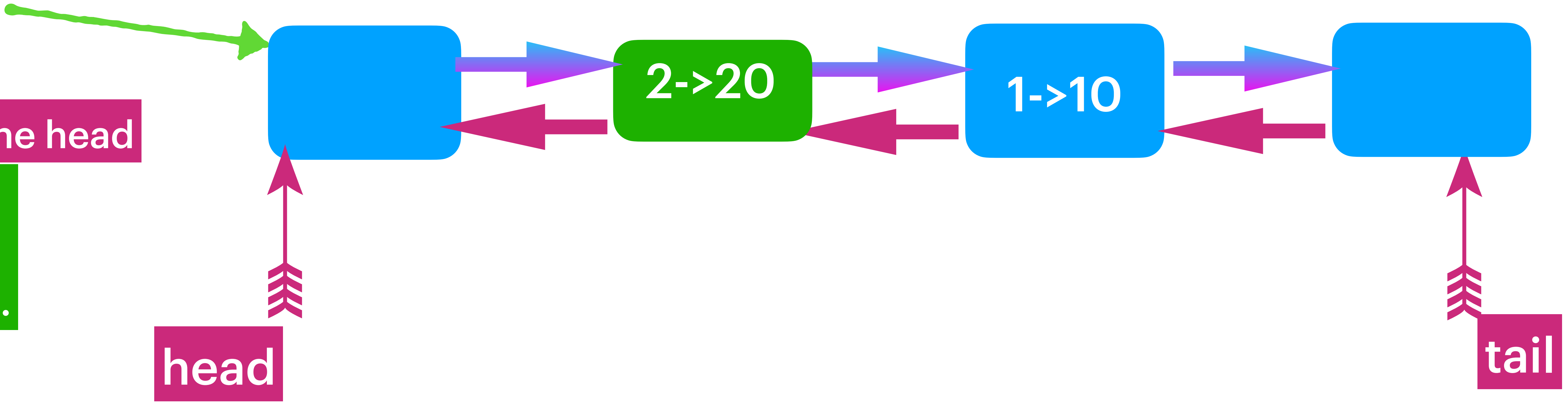


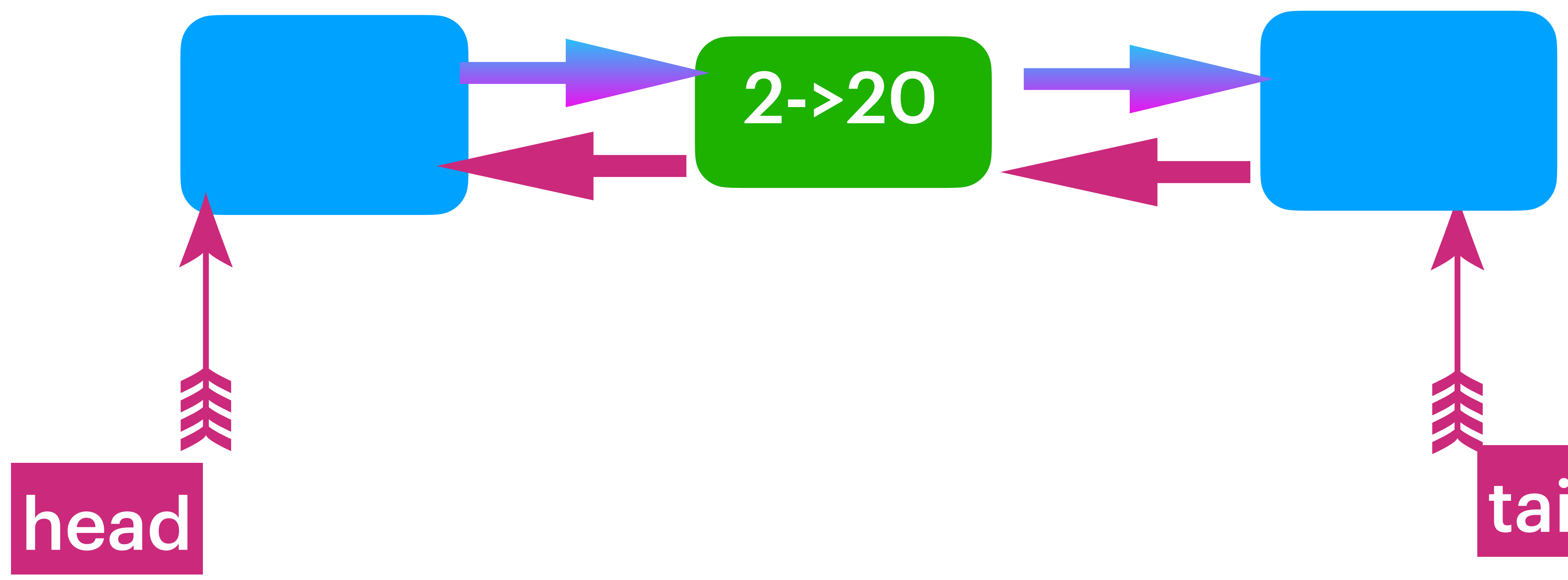
LRUCache : Capacity(2)

get(1)

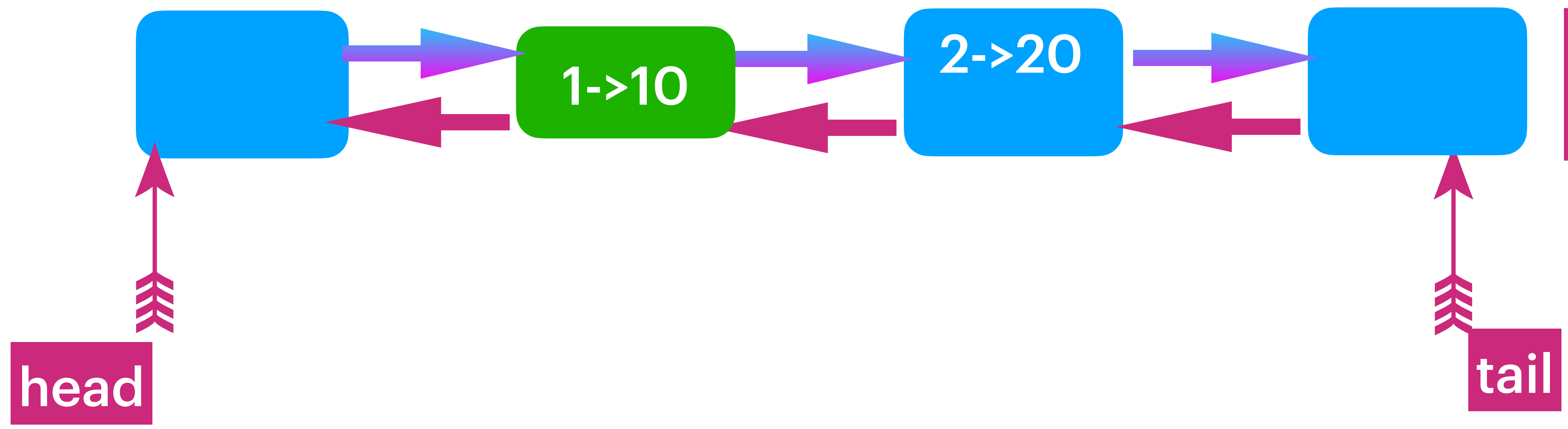
Move the accessed Node to the head

1) Remove currentNode
From LinkedList.
2) Then Add the currentNode.





Remove currentNode(1)



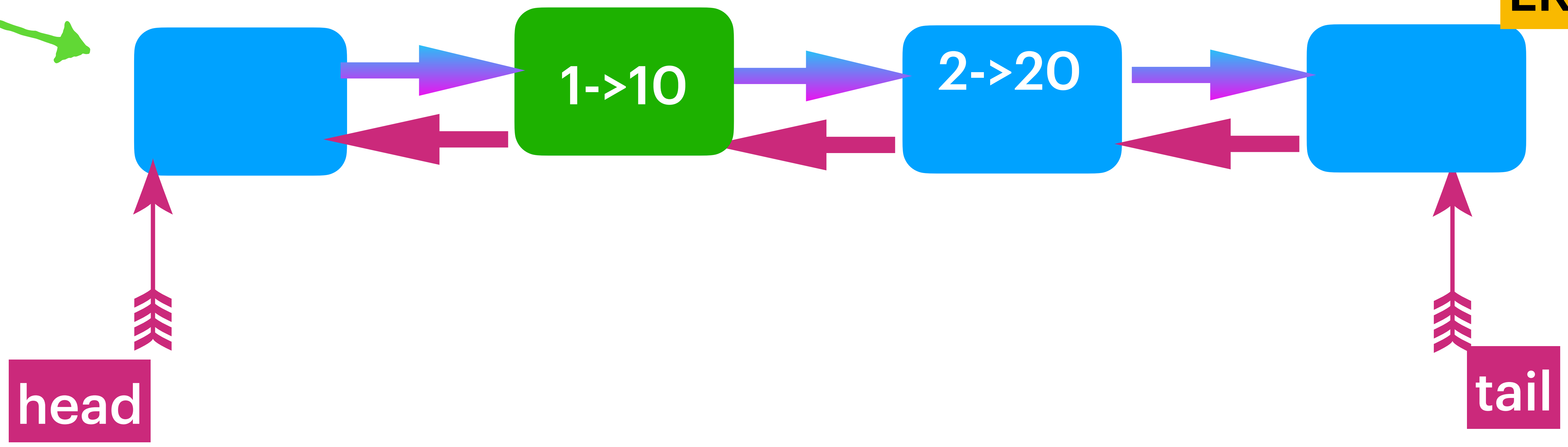
Then add currentNode(1)
Right to the head.

LRUCache : Capacity(2)

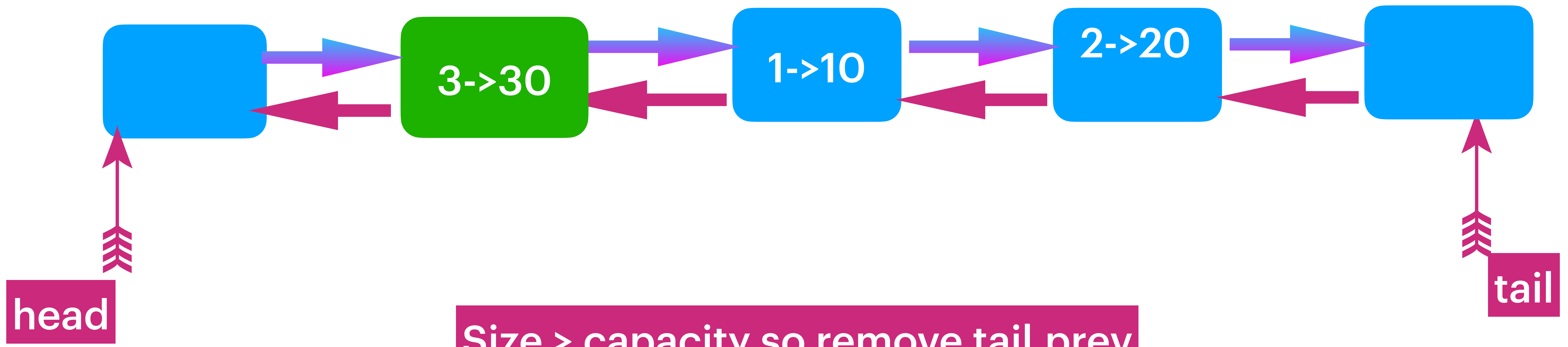
add(3,30)

Case 1: If the node does exist addToHead & If the size > capacity, remove tail.prev.

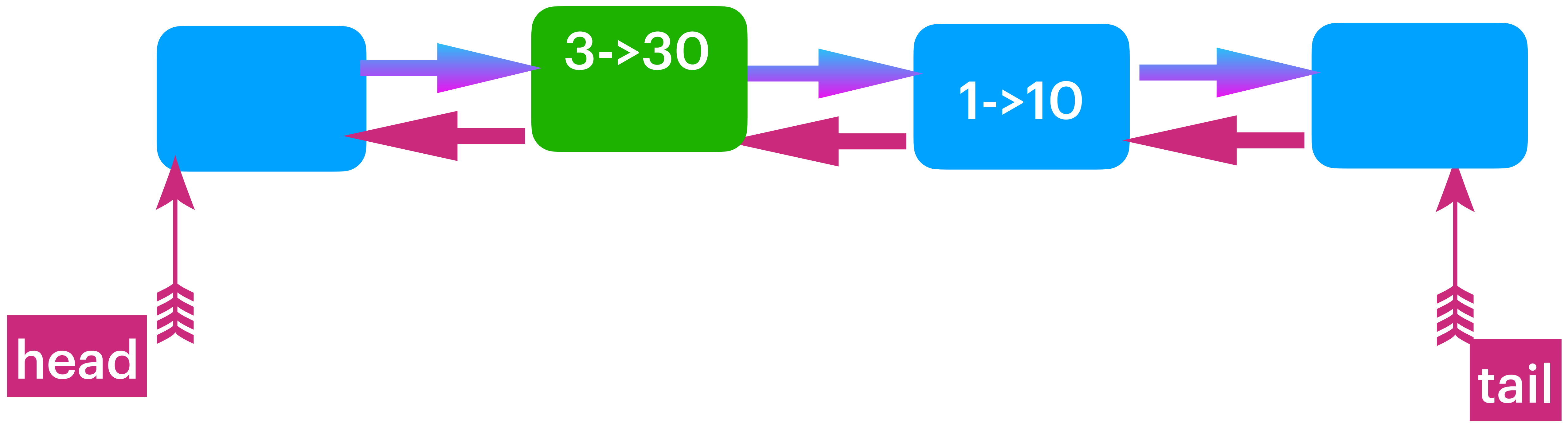
Case 2: If the node is present then Update the value moveToHead.



Node(3->30) does not exist so add right to the Head

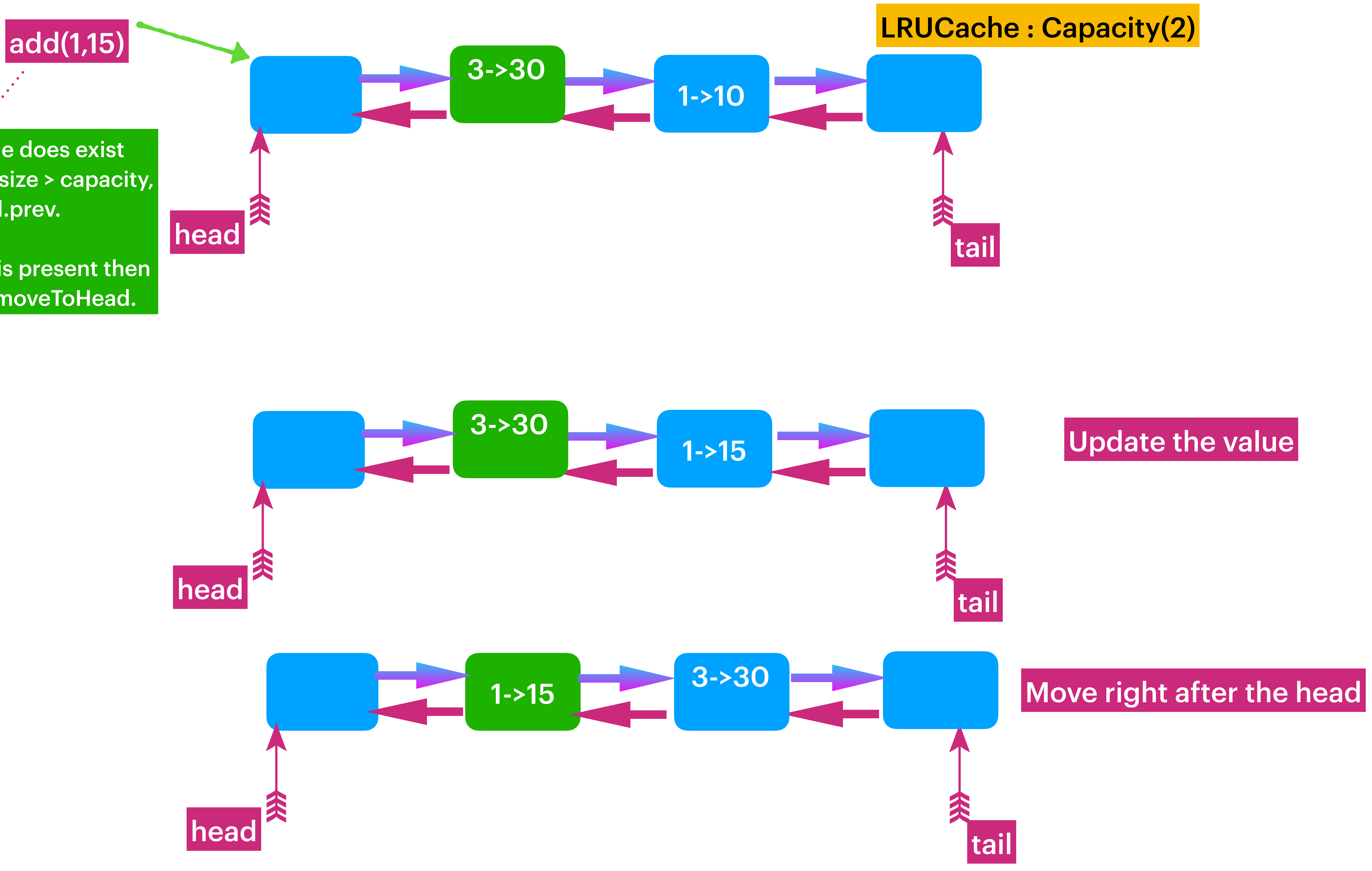


Size > capacity so remove tail.prev



Case 1: If the node does exist addToHead & If the size > capacity, remove tail.prev.

Case 2: If the node is present then Update the value moveToHead.


add(1,15)
LRUCache : Capacity(2)
3->30
1->10
head
tail
3->30
1->15
head
tail
Update the value
1->15
3->30
head
tail
Move right after the head