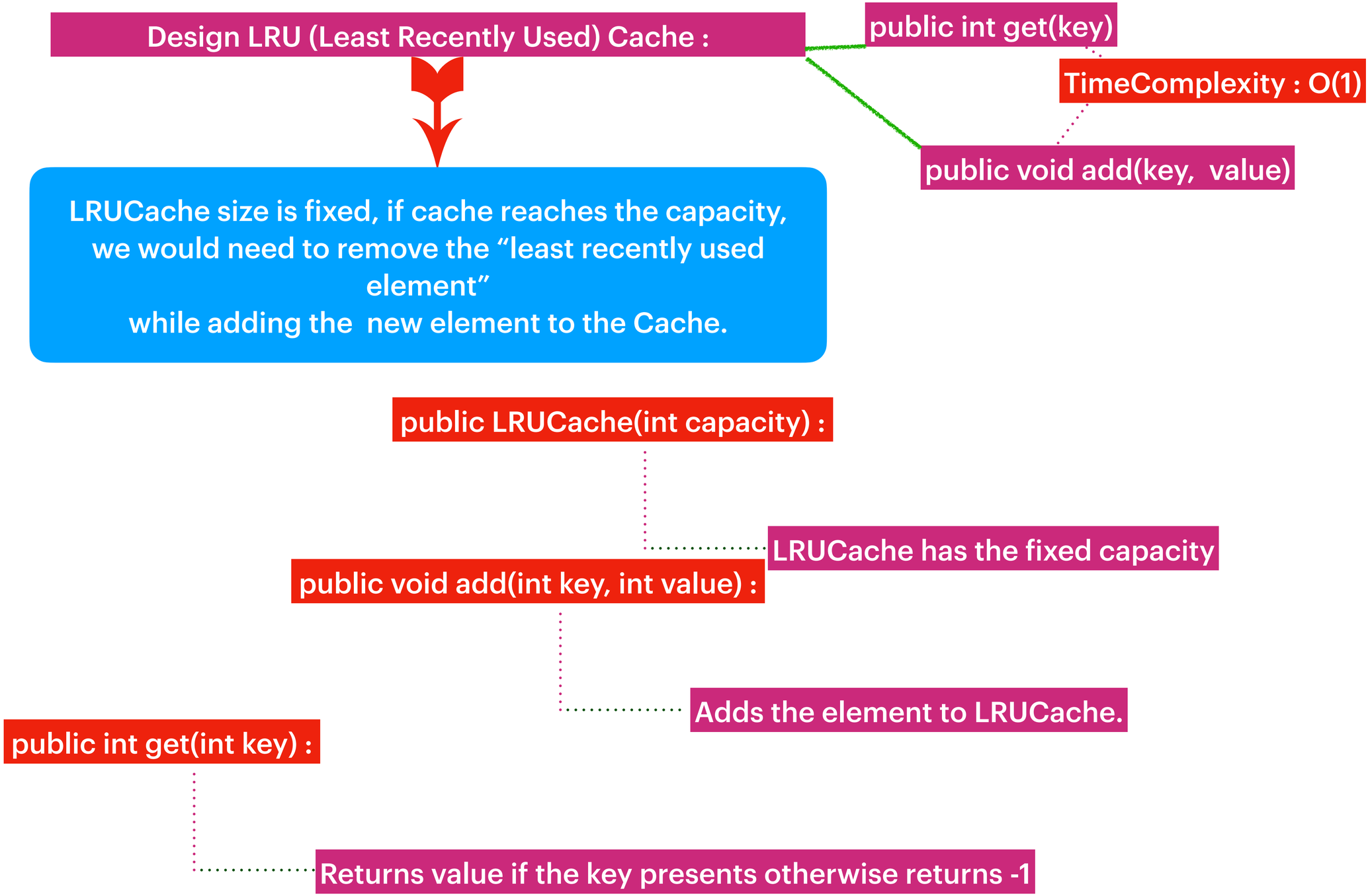
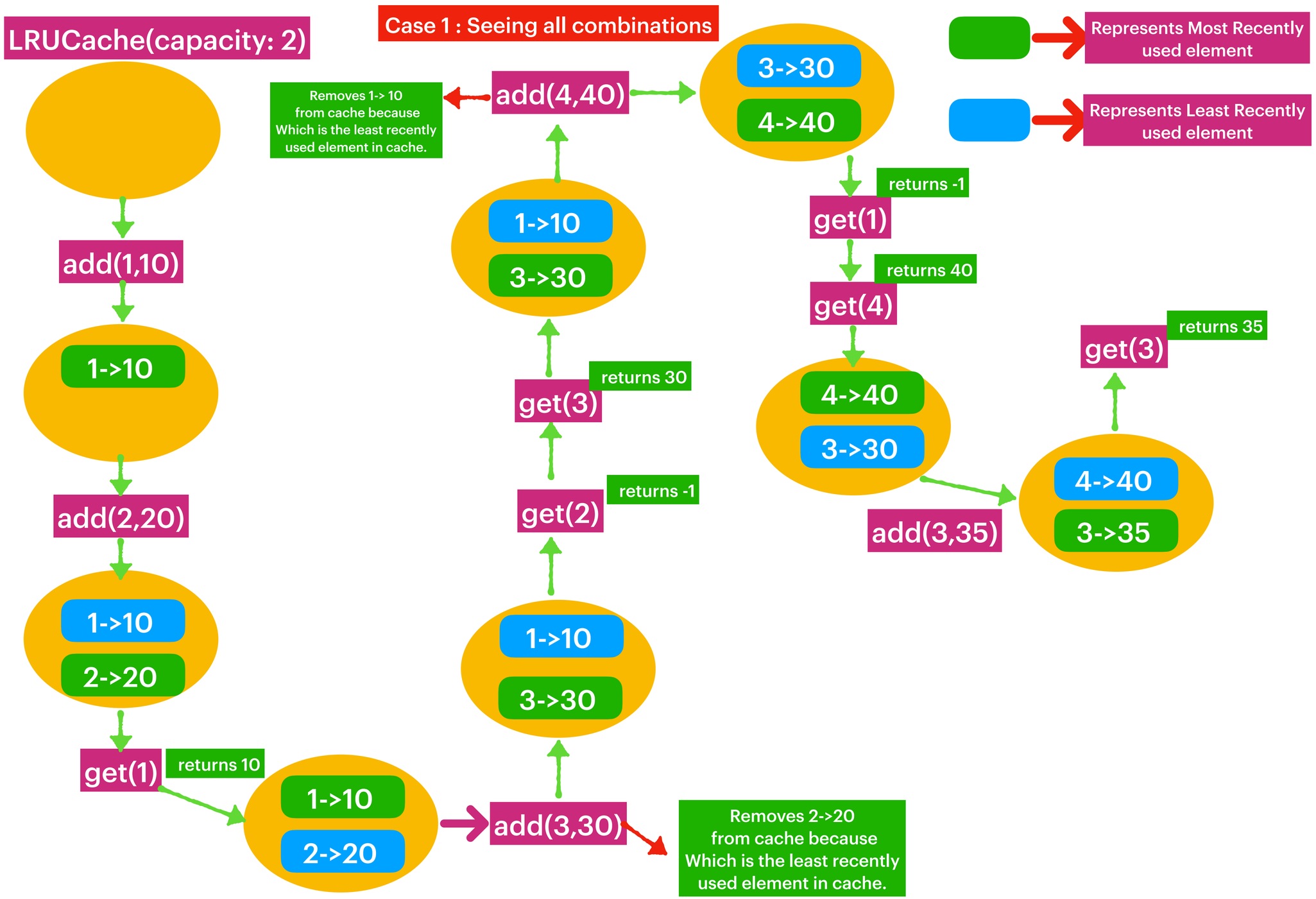
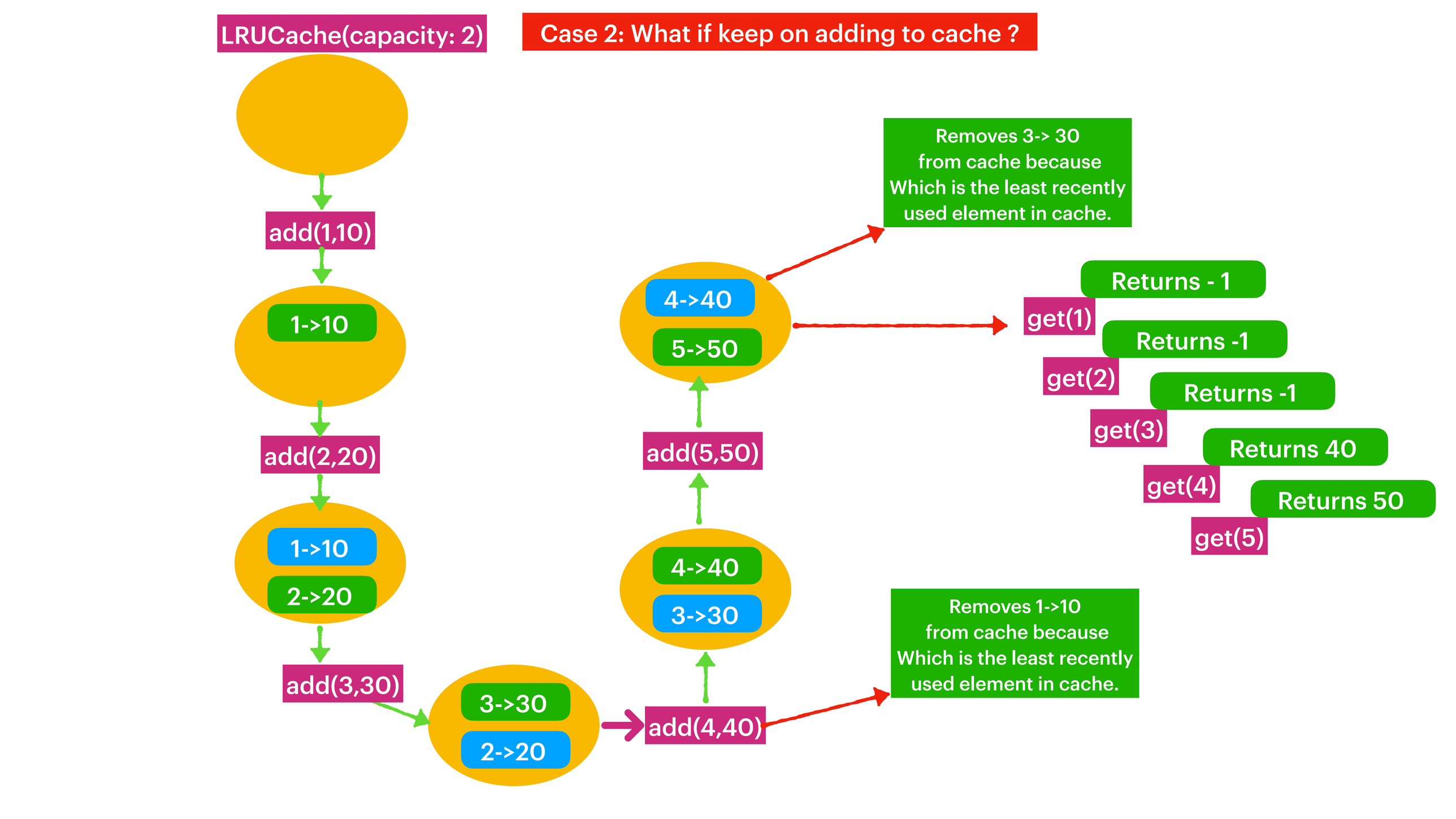
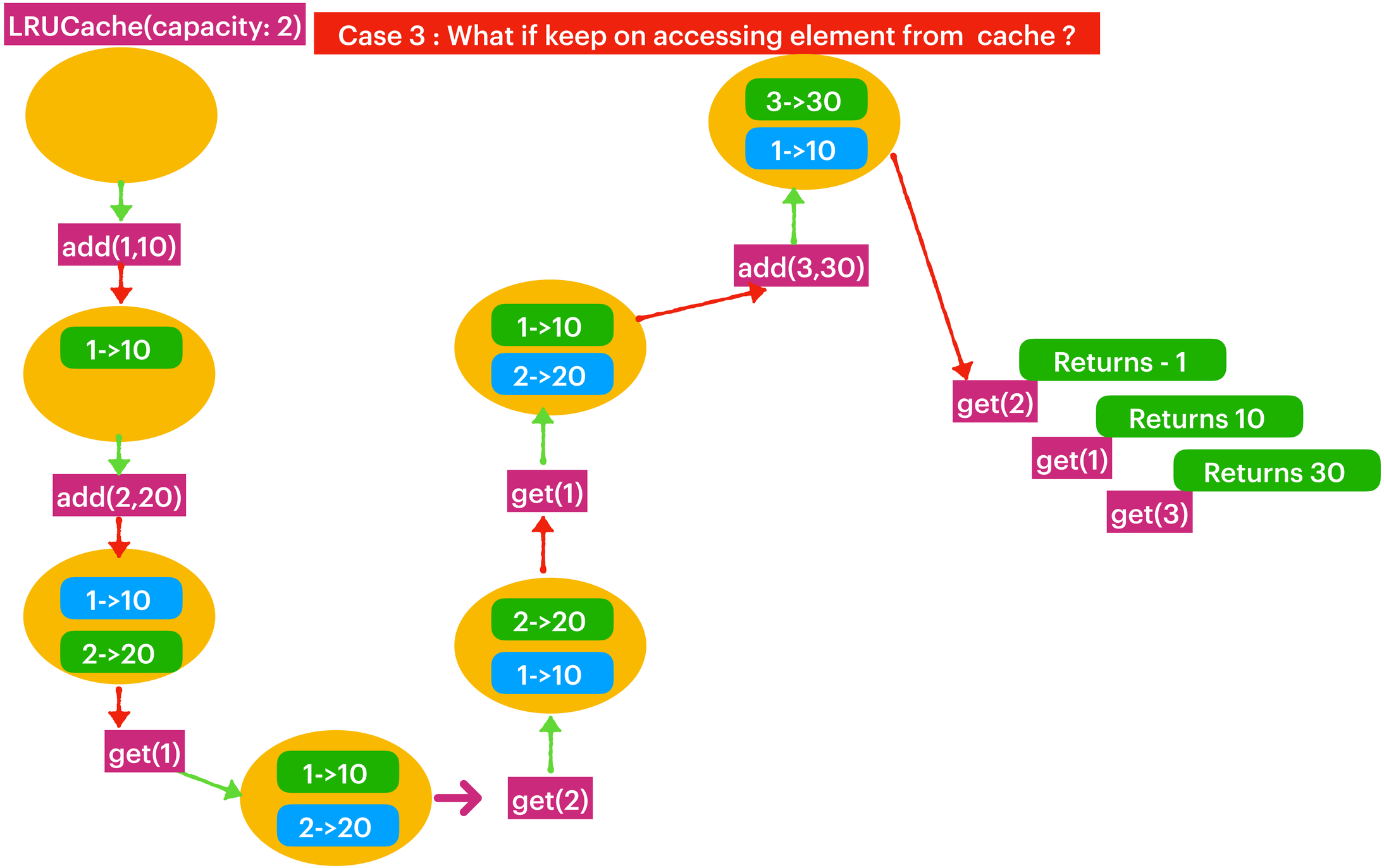

Design LRU (Least Recently Used) Cache :
public int get(key)
TimeComplexity : O(1)
public void add(key, value)
LRUCache size is fixed, if cache reaches the capacity, we would need to remove the “least recently used element”
while adding the new element to the Cache.
public LRUCache(int capacity) :
LRUCache 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)
add(1,10)
1->10
add(2,20)
1->10
2->20
get(1)
returns 10
1->10
2->20
Case 1 : Seeing all combinations
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
get(4)
returns 40
4->40
3->30
add(3,35)
4->40
3->35
get(3)
returns 35
Represents Most Recently
used element
Represents Least Recently
used element

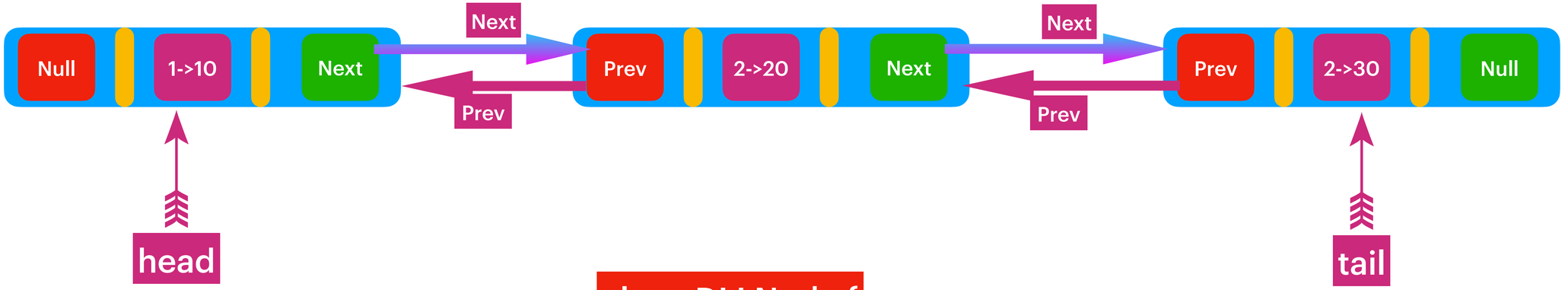

LRUCache(capacity: 2)
Case 2: What if keep on adding to cache ?
add(1,10)
1->10
add(2,20)
1->10
2->20
add(3,30)
3->30
2->20
add(4,40)
4->40
3->30
Removes 1->10
from cache because
Which is the least recently
used element in cache.
add(5,50)
4->40
5->50
Removes 3-> 30
from cache because
Which is the least recently
used element in cache.
get(1)
Returns - 1
get(2)
Returns -1
get(3)
Returns -1
get(4)
Returns 40
get(5)
Returns 50


LRUCache(capacity: 2)
Case 3 : What if keep on accessing element from cache ?
add(1,10)
1->10
add(2,20)
1->10
2->20
get(1)
1->10
2->20
get(2)
2->20
1->10
get(1)
1->10
2->20
add(3,30)
3->30
1->10
get(2)
Returns - 1
get(1)
Returns 10
get(3)
Returns 30

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.

Node

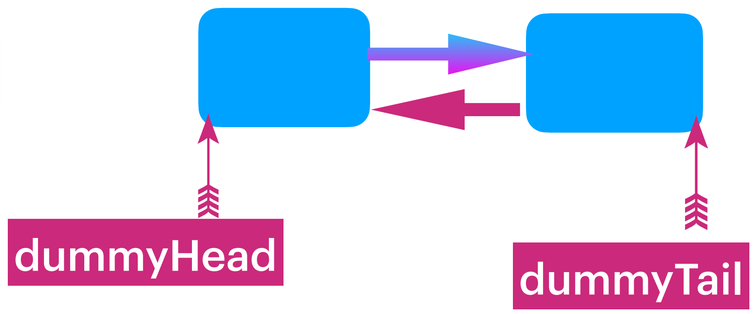



```
class  DLLNode {
int key,
int value,
Node next;
Node prev;
}
```

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

MRU : Most Recently Used Element

LRU : Least Recently Used Element

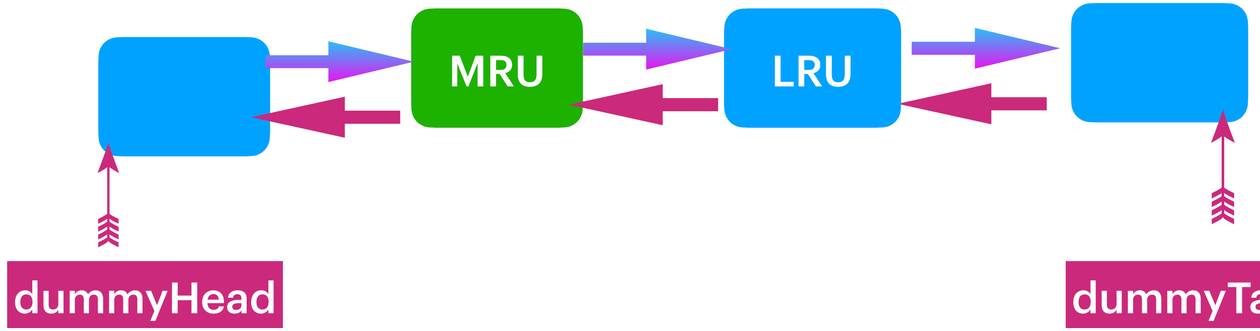


To maintain O(1) TimeComplexity
Use Map<key,DLLNode> with DoubleLinkedList
for add and get

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

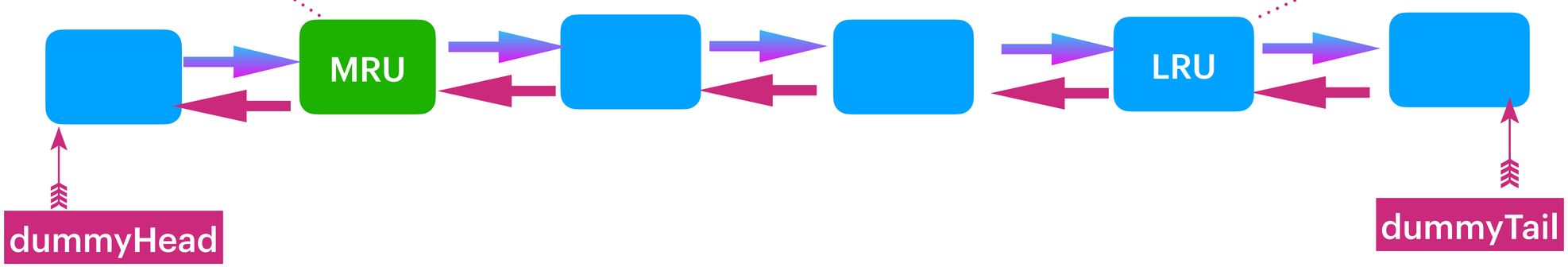
While
add(key, value)
(OR)
get(key)

Add the element right
after the dummyHead.

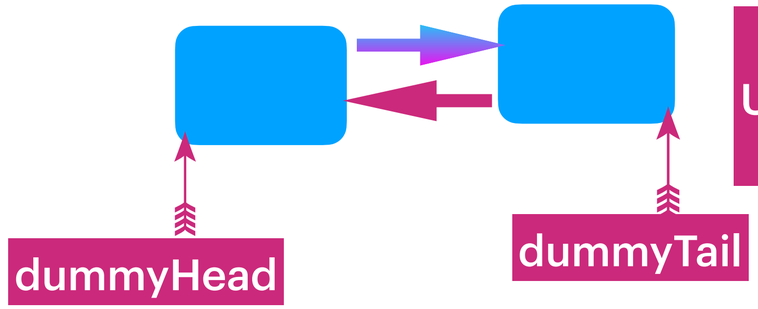


When the
size > capacity

Remove
LRU Element



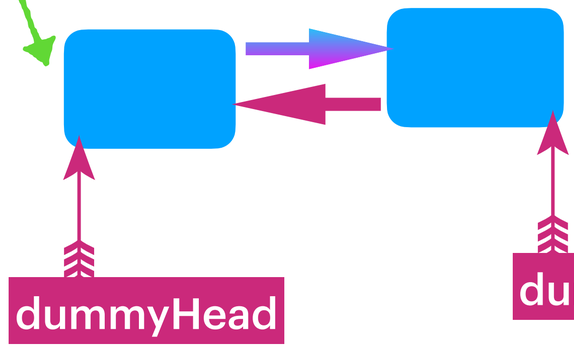
To avoid null checks maintain , take head & tail dummy nodes !!!



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

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

add(1,10)

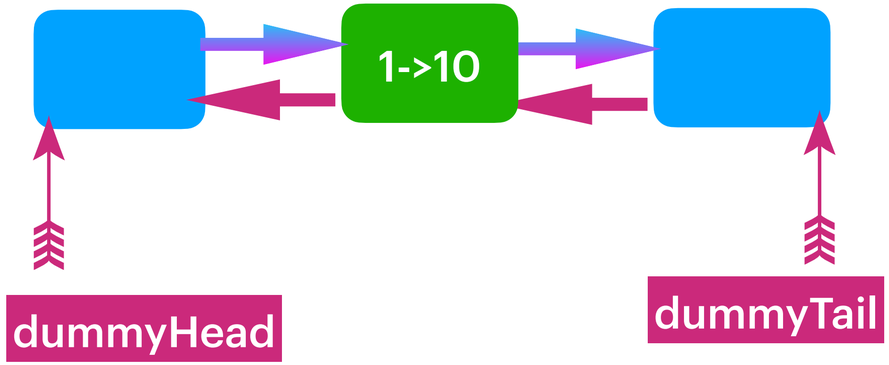


LRUCache : Capacity(2)

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.

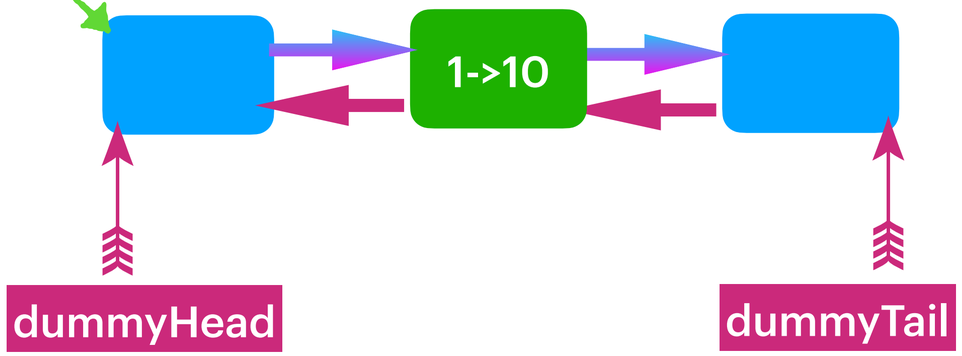
Node(1->10) does not exist so add to the dummyHead




```
public void addToHead(DLLNode currentNode)
{
    DLLNode headNext = dummyHead.next;
    dummyHead.next = currentNode;

    headNext.prev = currentNode;

    currentNode.next = headNext;
    currentNode.prev = dummyHead;

    map.put(currentNode.key, currentNode);
    size++;
}
```

LRUCache : Capacity(2)

add(2,20)

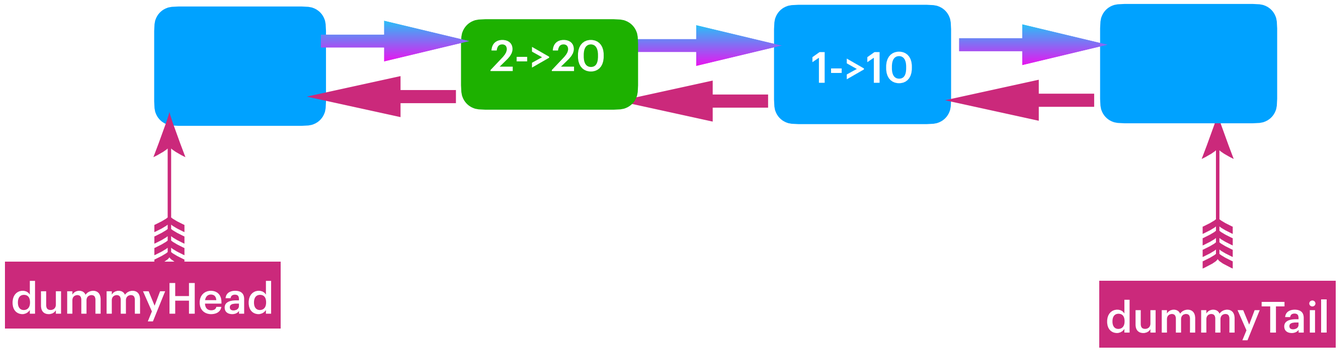


Case 1: If the node does exist addToHead & If the size > capacity, remove tail.prev.

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

```
public void addToHead(DLLNode currentNode)
{
DLLNode headNext = dummyHead.next;
dummyHead.next = currentNode;

headNext.prev = currentNode;

currentNode.next = headNext;
currentNode.prev = dummyHead;

map.put(currentNode.key, currentNode);
size++;
}
```

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

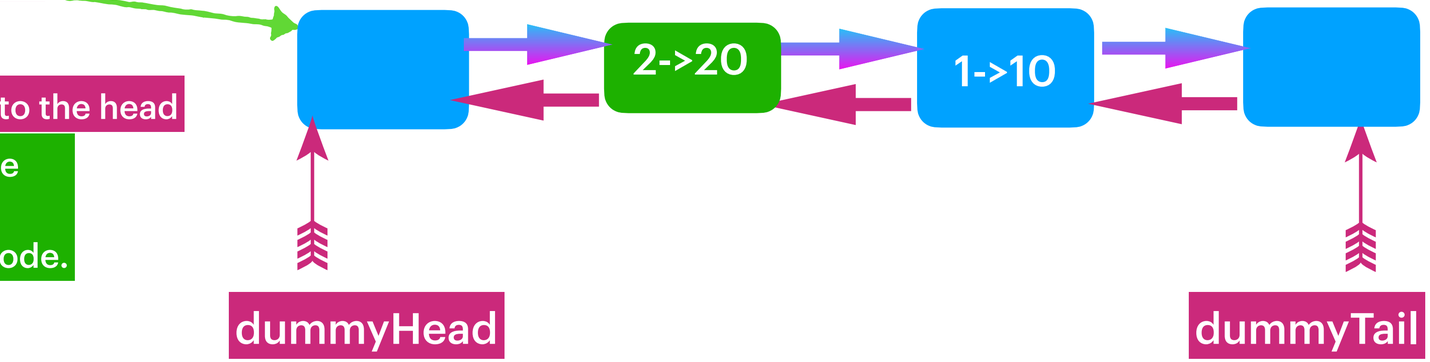


LRUCache : Capacity(2)

get(1)

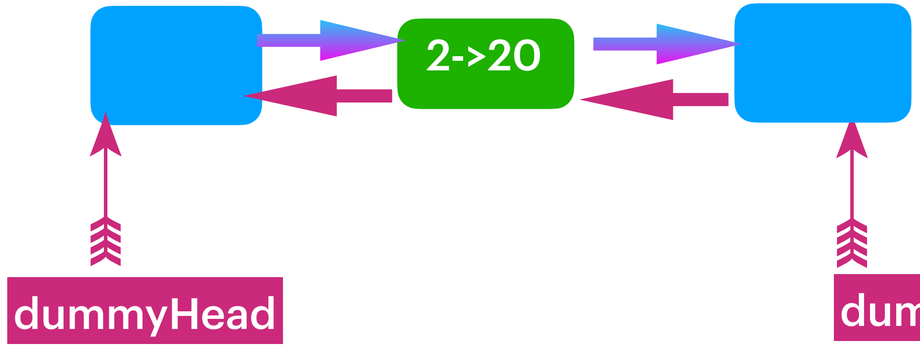
Move the accessed Node to the head

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

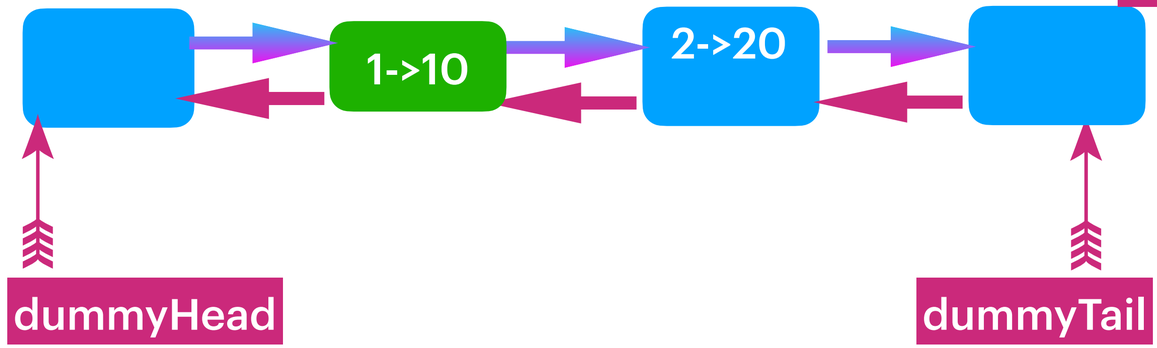



```
public void removeNode(DLLNode currentNode)
{
DLLNode prev = currentNode.prev;
DLLNode next = currentNode.next;

prev.next = next;
next.prev = prev;

map.remove(currentNode.key);
size--;

}
```

Remove currentNode(1)
From LinkedList.



Then add currentNode(1)
to the dummyHead.



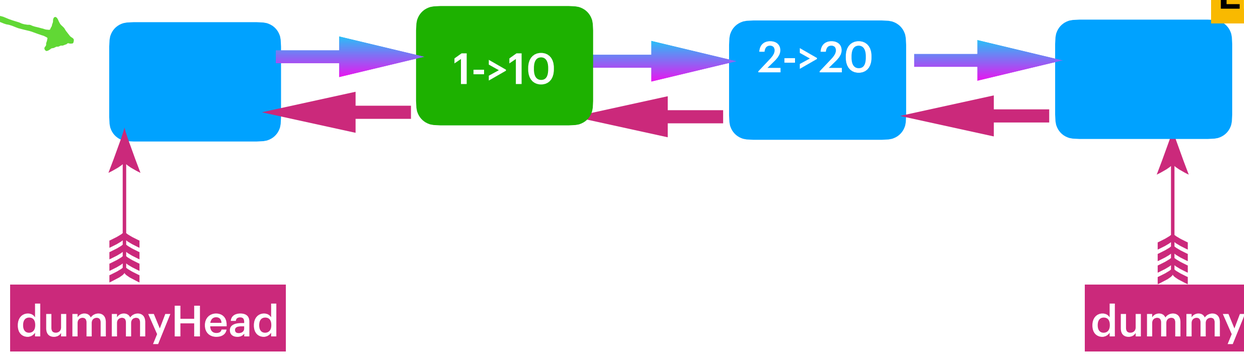

```
public void addToHead(DLLNode currentNode)
{
DLLNode headNext = dummyHead.next;
dummyHead.next = currentNode;

headNext.prev = currentNode;

currentNode.next = headNext;
currentNode.prev = dummyHead;

map.put(currentNode.key, currentNode);
size++;
}
```

LRUCache : Capacity(2)

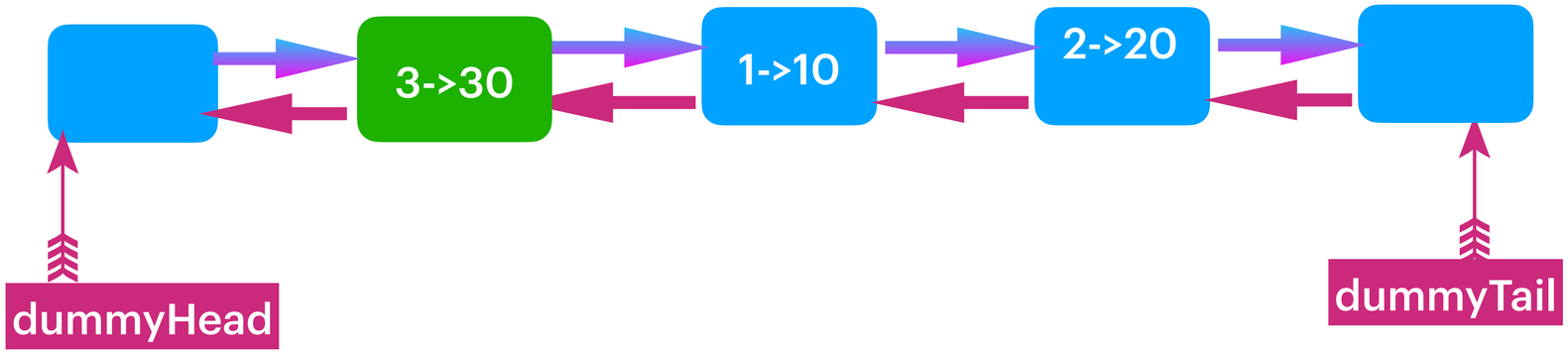
add(3,30)

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

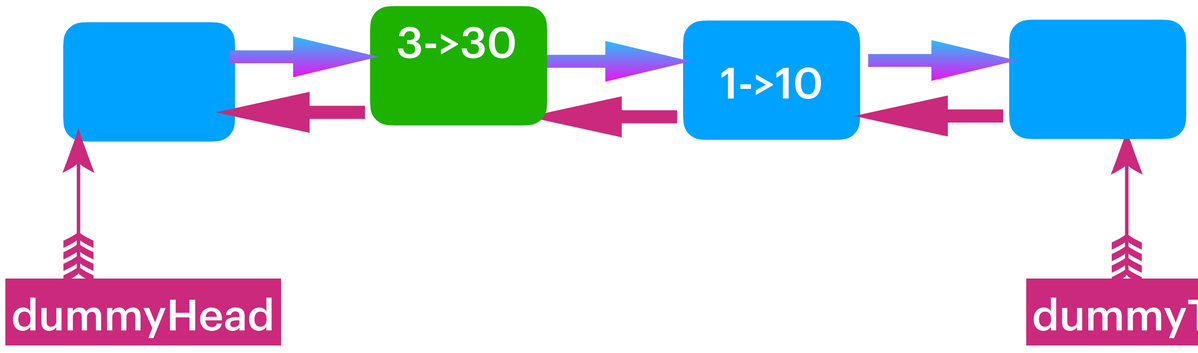
Case 2: If the node is present then Update the value moveToHead.



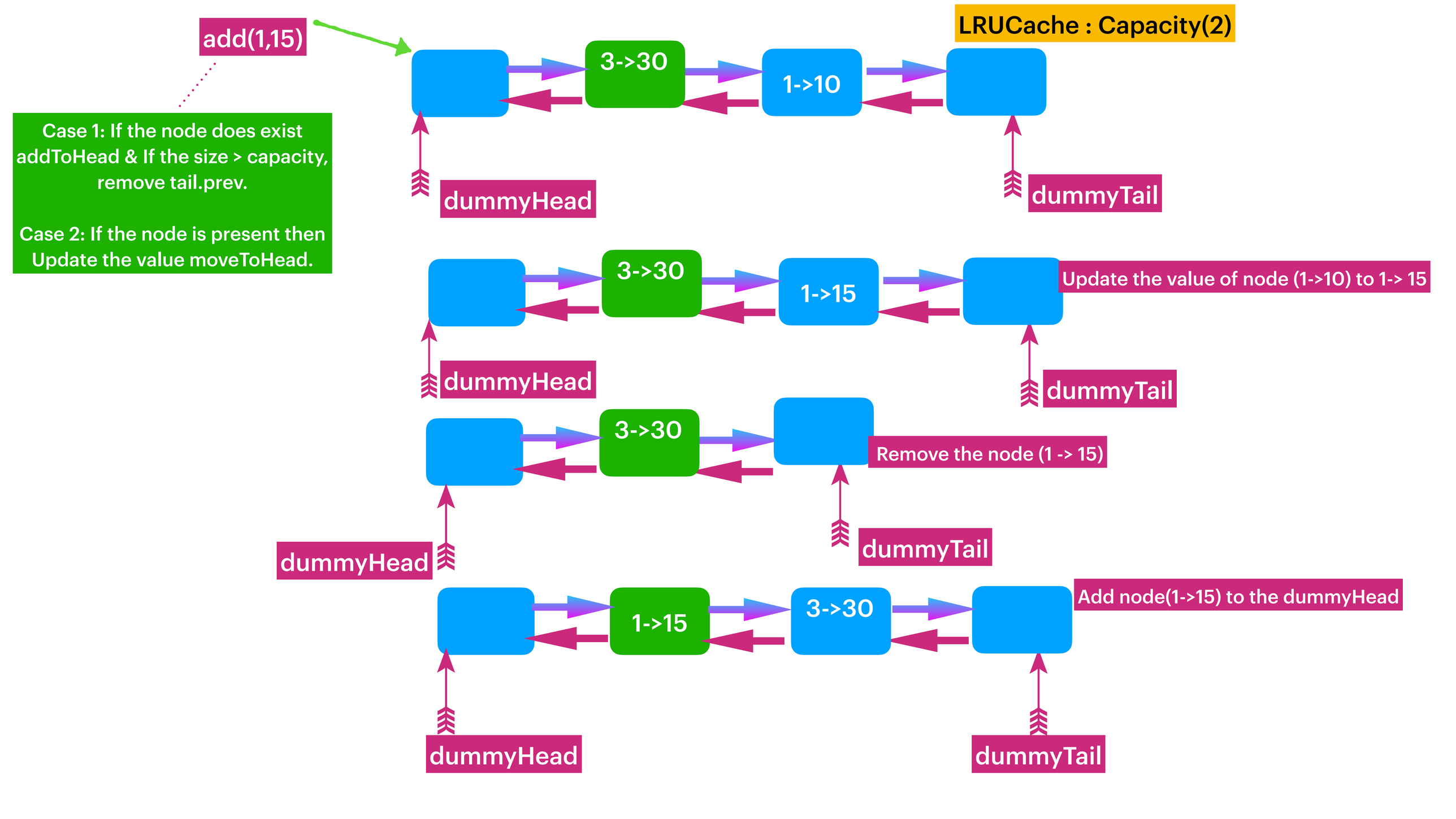
Node(3->30) does not exist so add to Head

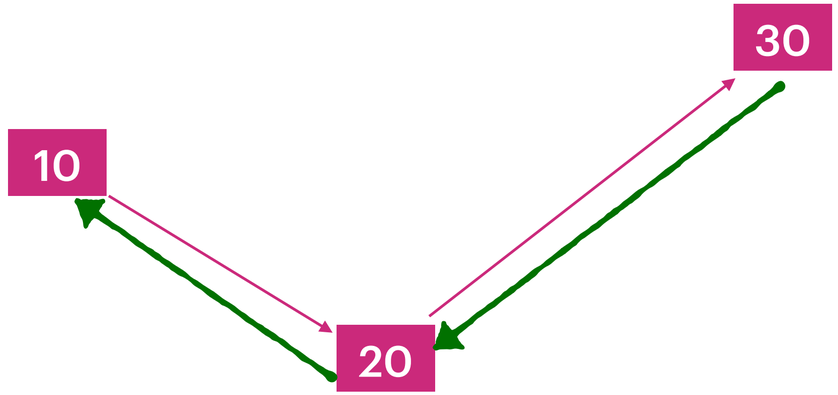


Size > capacity so remove dummyTail.prev

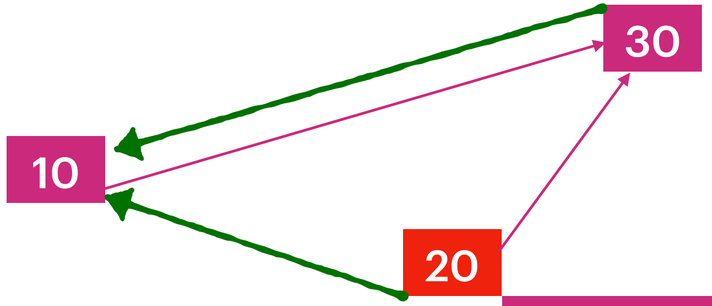



```
Remove dummyTail.prev
// dummyTail.prev always maps to LRU Element
public void removeTail()
{
DLLNode tailPrev = dummyTail.prev;
removeNode(tailPrev);
}
```


LRUCache : Capacity(2)
add(1,15)
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.
3->30
1->10
dummyHead
dummyTail
3->30
1->15
Update the value of node (1->10) to 1-> 15
dummyHead
dummyTail
3->30
Remove the node (1 -> 15)
dummyHead
dummyTail
1->15
3->30
Add node(1->15) to the dummyHead
dummyHead
dummyTail




```
Remove(20)::

DLLNode prev = currentNode.prev;
DLLNode next = currentNode.next;

prev.next = next;
next.prev = prev;
```



As node(20) is not referred by any reference eligible for Garbage Collection