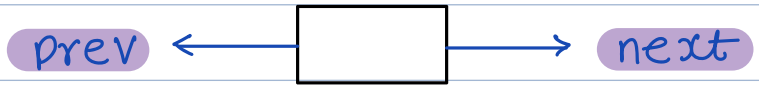
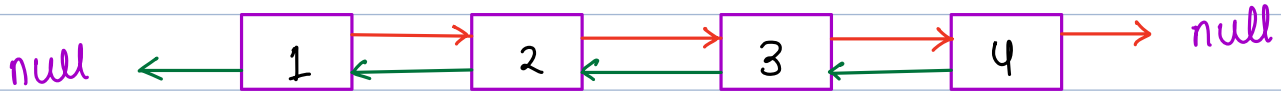
Linked List 3

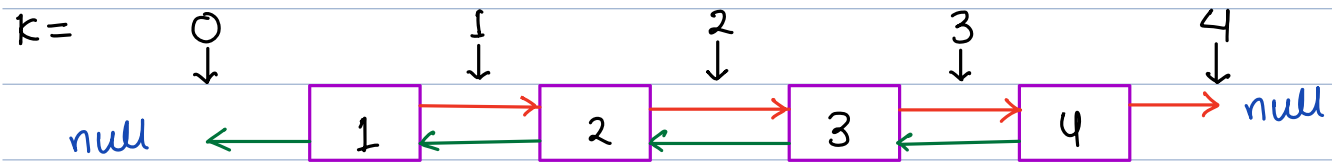
- Doubly Linked List
- Insert a node
- Delete first occurrence
- LRU cache *
- Shallow Copy vs Deep Copy
- Copy List.

Doubly Linked List

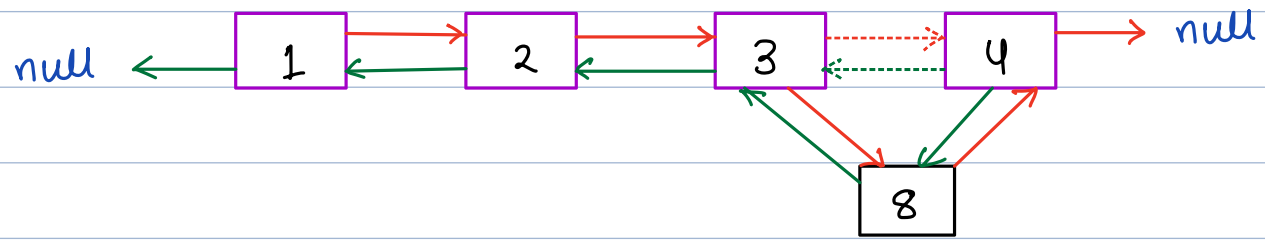




Given a doubly LL. Insert a node with data X at a position K [0<= K <= N]

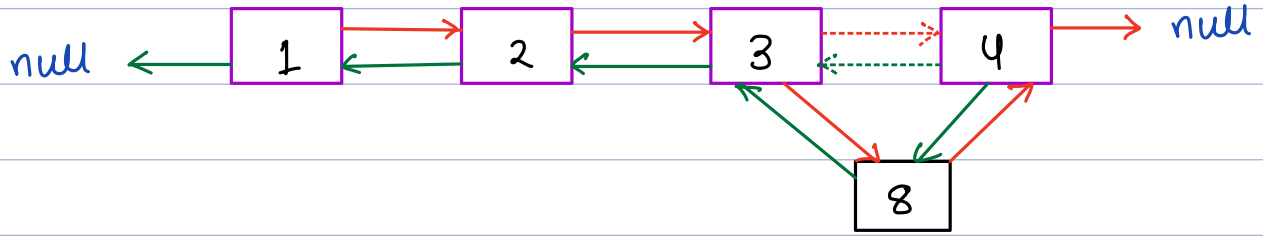


X = 8 , K = 3

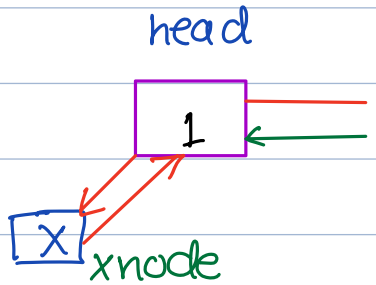


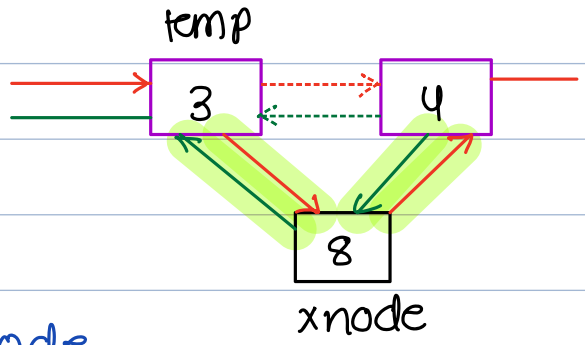
Corner Cases

1) head can be null
2) K == 0




```
Node    insertNode ( head , X , K ) {

    xnode = Node (X)
// Check for null
if (head == null ) {
    |     return xnode
    |
    }

if (K==0) {
    |     head.prev = xnode
    |     xnode.next = head
    |      return xnode
    |
    }

temp = head

for (i=0 ; i< K-1 ; i++ ) {
    |     temp = temp.next
    |
    }
```





① xnode.next = temp.next

if (temp.next != null) {
② temp.next.prev = xnode
} // ② can give NPE

③ xnode.prev = temp

④ temp.next = xnode

return head

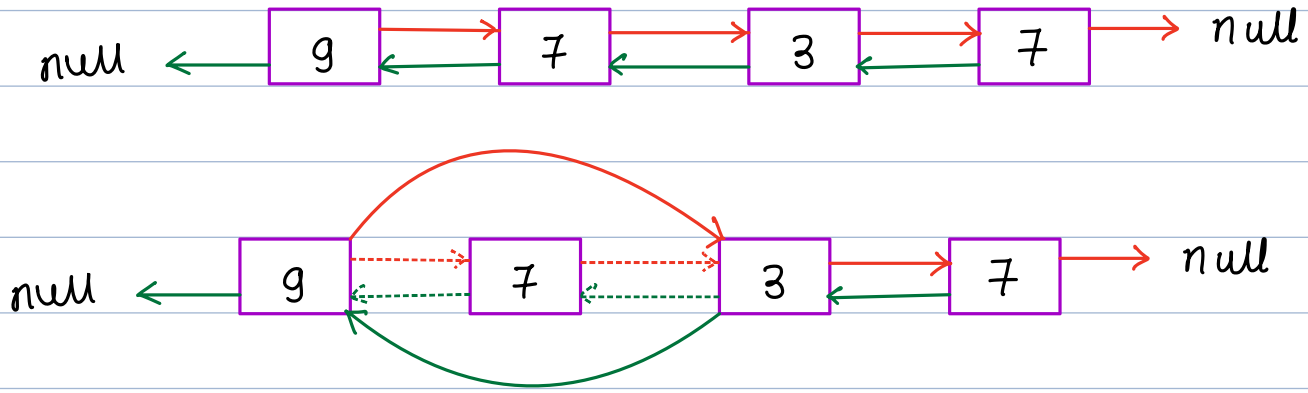
TC: O(k)

SC: O(1)

Q> Delete the first occurrence of data X in a given doubly linked list.

If not present then, no updates

X = 7



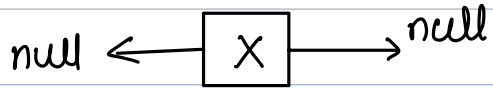

```
Node deleteNode ( head , X ) {

    if (head == null)   return head

    // Find the node to be deleted

    temp = head

    while ( temp != null) {
        if (temp.val == X ) {
            break
        }
        temp = temp.next
    }
```

TC: O(N)
SC: O(1)

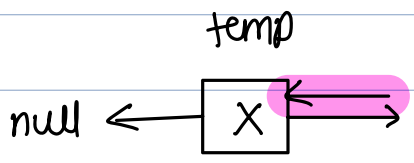
TC: O(N)

```
// check if temp is the node to be
   deleted
if (temp == null) {
    return head
}
```

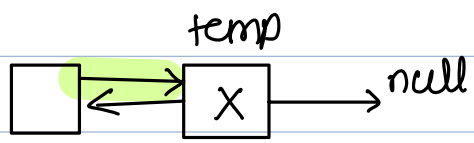
```
// Single node
if (temp.next == null &&
    temp.prev == null) {
    return null
}
```



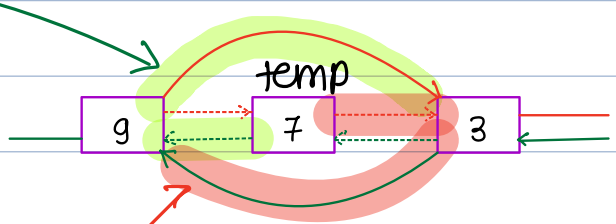

```
// Head to be deleted
if (temp.prev == null) {
    temp.next.prev = null
    return temp.next
}
```




```
// Temp is the last node
if (temp.next == null) {
    temp.prev.next = null
    return head
}
```




```
temp.prev.next = temp.next
temp.next.prev = temp.prev

return head
```



Q> Given a running stream of integer and a fixed memory of size M ie SC = O(M)

∀ intake, maintain most recent M elements

In simple words

∀ intake X ⟶ If X is not present

1> If memory is full then delete least recent item

2> Insert X as most recent

If X is present ⟶ Update X as most recent item

10 15 19 20 18 23 20 19 17 17 10

M = 5

~~10~~ ~~15~~ ~~19~~ ~~20~~

~~18~~ 23 20 19

~~17~~ 17 10

X

X is present in memory

① delete old X in memory

② insert X as most recent item

X is not present in memory

① check if our memory is full, delete the oldest item

② insert X as most recent item

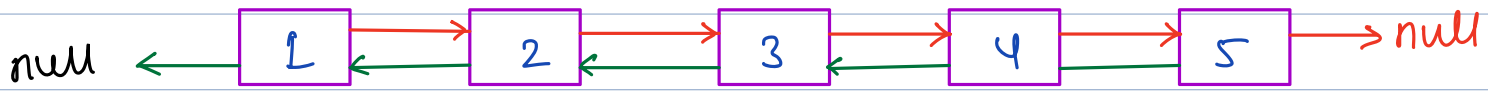
what is the best datastructure to do the checking part ie if X is present or not ?

Hashmap / ~~hashset~~. (X, Node)

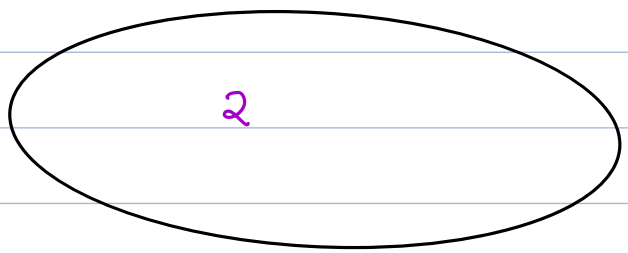
To maintain order → ~~Array~~, ~~Dynamic Array~~

~~tt~~, doubly LL

since delete take O(1) time if node is given



Pseudo code




```
if (x is present in map) {
    xnode = map.get(x)
    head = deleteNode(head, xnode)

    // insert at tail  assume we maintain head & tail
    tail.next = xnode
    xnode.prev = tail
    tail = xnode
}
```

(X, Node)

```
if ( x is not present in map ) {
    if (map.size() == M ) {   // memory full
        map.remove(head.val)

        head = deleteNode (head, head)
        xnode = new Node(x)
        map.put (x, xnode)

        tail.next = xnode
        xnode.prev = tail
        tail = xnode
    }
    else {
        xnode = new Node(x)
        map.put (x, xnode)

        if (map.size() == 1 ) {
            head = xnode
            tail = xnode
        }
        else {
            tail.next = xnode
            xnode.prev = tail
            tail = xnode
        }
    }
```

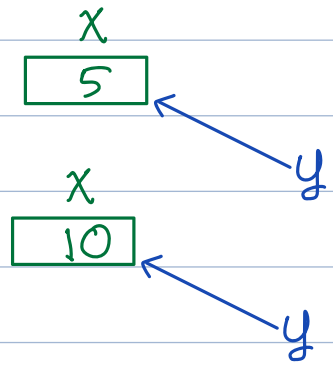
per x whats the TC : O(1)

22:29

Shallow copy

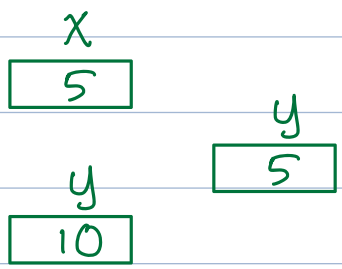
```
x = new Node(5)
y = x
y.val = 10
print(x.val)
```

10



Deep copy

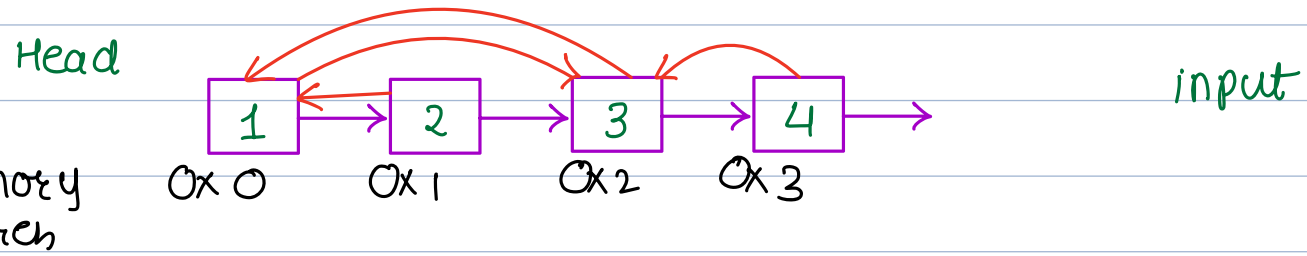
```
x = new Node(5)
y = new Node(x.val
y.val = 10
print(x.val)
```



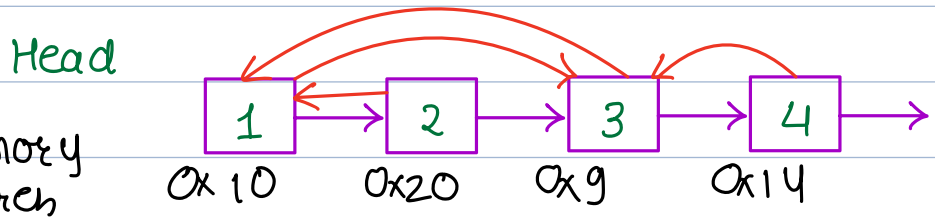
Q> Create a deep copy of given LL with random pointers

Random ⟶

Next ⟶

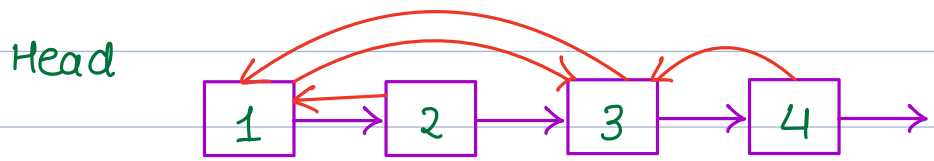


input

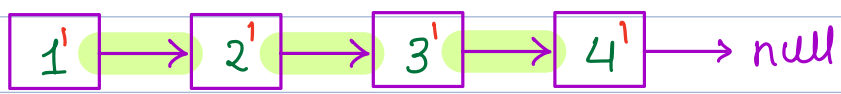


output.

Idea 1> Copy singly LL without random pointers



input



use Hashmap { key : old node
value : new node

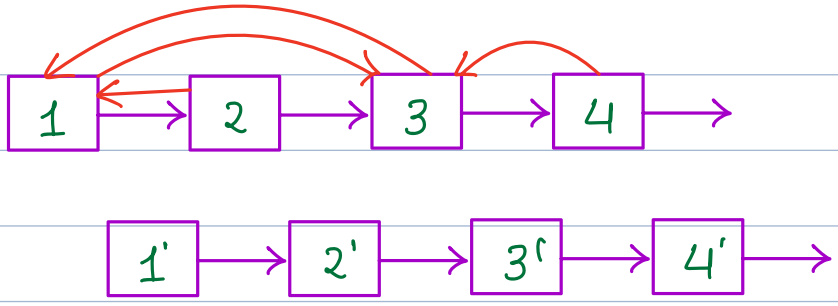
Pseudocode

```
Node    copyRandomList(head) {
        Map<Node, Node> map = new HM<>()
// map null to null
        // step 1 create old node to new node map
        temp = head
        while (temp != null) {
            map.put(temp, new Node(temp.val))
            temp = temp.next
        }

    temp = head
    while (temp != null) {
        newNode = map.get(temp)
        // next mapping
        newNode.next = map.get(temp.next)
        newNode.random = map.get(temp.random)
        temp = temp.next
    }

    return map.get(head)

}
```

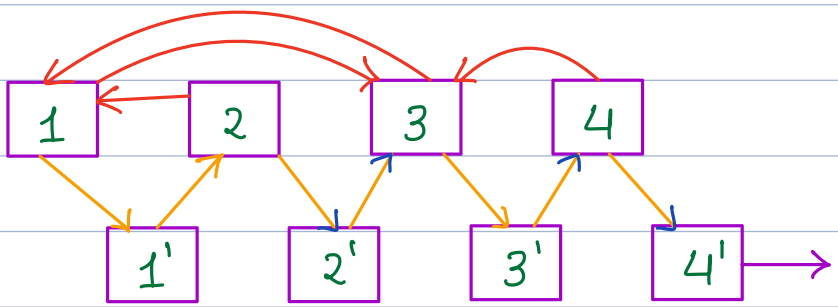
TC: O(N)

SC: O(N)

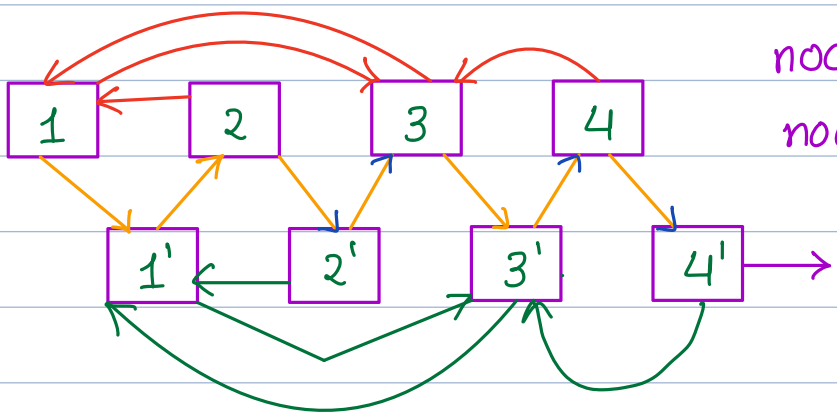
Extra



step 1 create a new linked without random pointers



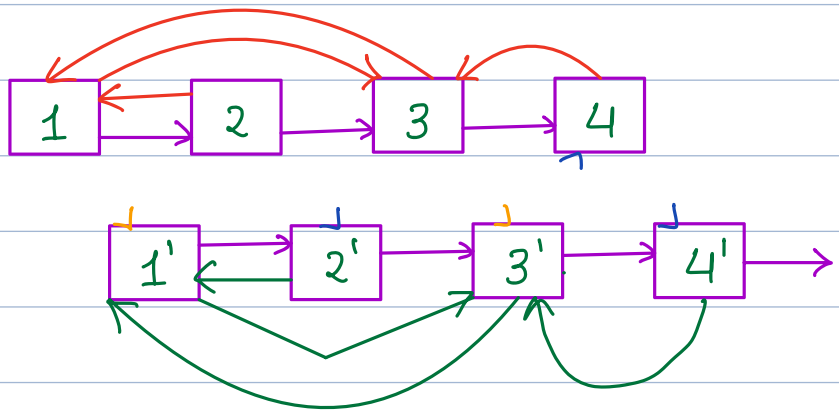
step2 Interweave old linked list and new



node.next.random =
node.random.next

step 3 create the random pointers for new LL

step 4 Break the interweaving done previously in step 2



Can you consider the space of new LL in SC