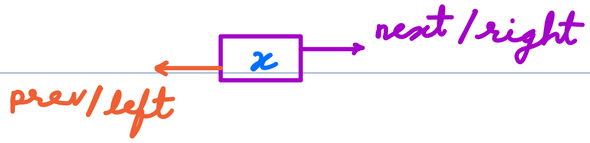
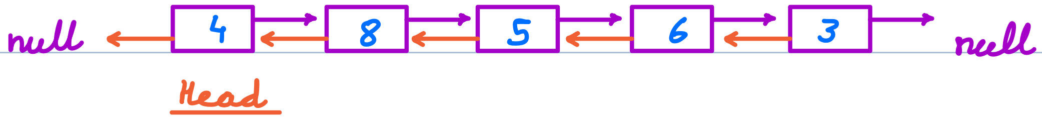
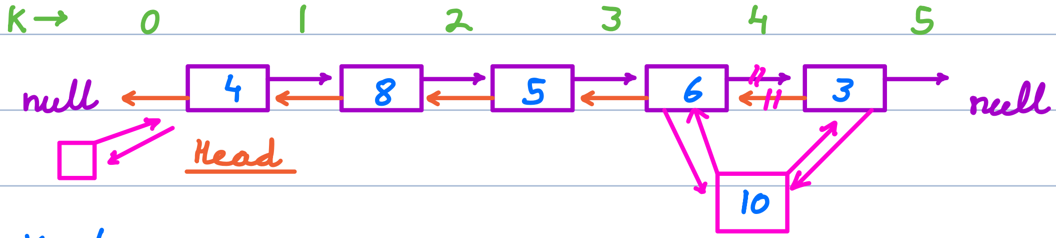
Doubly Linked List





Q → Insert a node with data X at position K [0 N] in doubly linked list.



K = 4

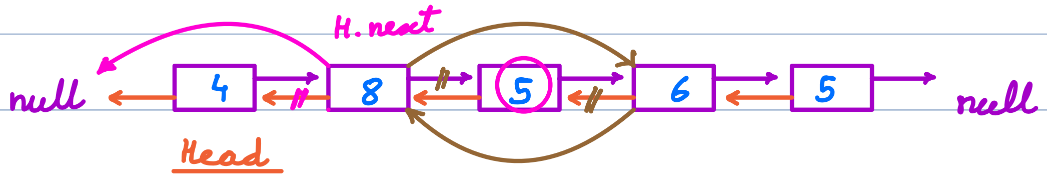
X = 10

```
nn = new Node(X)  // nn.next = nn.prev = null
if (Head == null)   return nn
if (K==0) {
    nn.next = Head
    Head.prev = nn
    return nn   // updated Head
}

temp = Head
for i → 1 to (K-1) {
    temp = temp.next
}

nn.prev = temp
nn.next = temp.next
if (temp.next != null)      temp.next.prev = nn
temp.next = nn
```

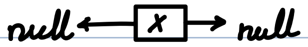
```
return Head
```

TC = O(K)   SC = O(1)

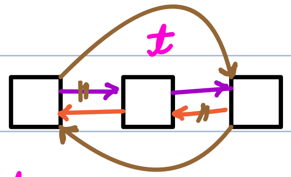
Q → Given a doubly linked list, delete the first occurrence of X. If not present, ignore.

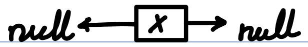


X = 5



```
if (Head == null) return Head
if (Head.data == X) {
    if (Head.next != null)
        Head.next.pre = null
    Head = Head.next
    return Head
}
Temp = Head
while (temp != null) {
    if (temp.data == X) break
    temp = temp.next
}
if (temp == null) return Head
// node to delete → temp
if (temp.prev != null) // non-Head
    temp.prev.next = temp.next
if (temp.next != null)
    temp.next.prev = temp.prev
return Head
```





$TC = O(N)$ $SC = O(1)$

L R U → least Recently Used

Q → Given a running stream of integers, & a fixed memory $\boxed{SC = O(M)}$. ∀ input maintain most recent M inputs in the memory. If memory is full remove least recent data.

Eg → $\boxed{M = 5}$ | i/p → 10 15 19 20 18 23 20 19 17 17 10

~~10~~ ~~15~~ 19
20 ~~18~~ 23
17 10

∀ intake x,

(once the memory is full)

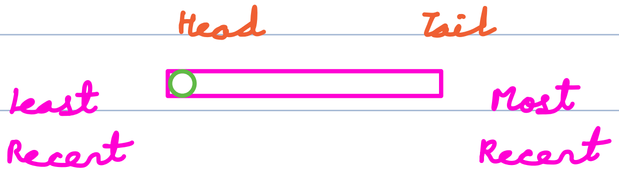
if x is not present

1) delete least recent item ✓
2) insert x as most recent item ✓

if x is already present

1) delete x from its position
2) insert x as most recent item ✓

Maintain elements in order of recency → ~~Array~~ / ~~Linked List~~ / ~~Dynamic Array~~ / DLL



if we directly reach the node to delete.

⇒ use HashMap

< x , node with data x >

```
∀ intake x,
if ( hm. containsKey (x)) {
        xn = hm.get (x)
        deleteNode (Head, xn) → TC = O(1)
        insertNode (Tail, xn) ↗
} else if ( hm. size () < M) {
        xn = new Node (x)
        insertNode (Tail, xn)
        mp. put (x, xn)
} else {  // Memory full
        mp. remove (Head.data)
        deleteHead (Head)
        xn = new Node (x)
        insertNode (Tail, xn)
        mp. put (x, xn)
}
```

TC per i/p = <u>O(1)</u>   SC = <u>O(M)</u>

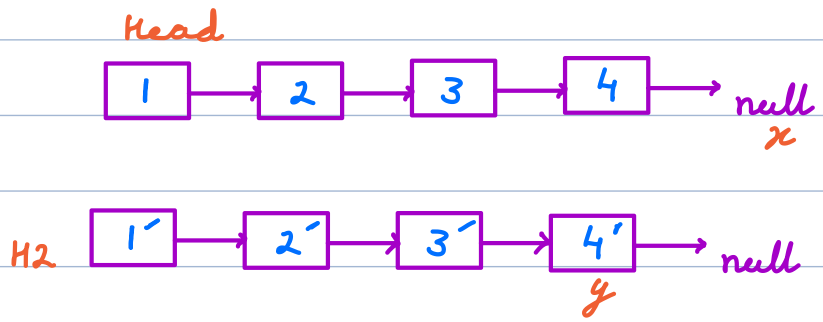
```
[~~5~~ 10] ← x = new Node (5)
           ↖ y = x
y . data = 10
print (x. data) // 10
```

<u>**Shallow Copy**</u>

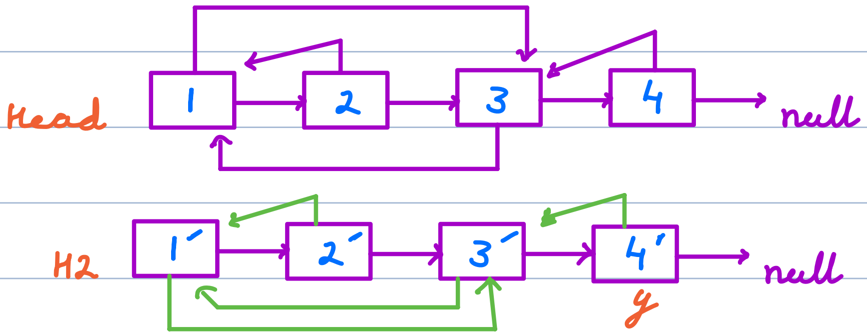
```
x = new Node (5) ⟶ [5]
y = new Node (x.data) → [~~5~~ 10]
y . data = 10
print (x. data) // 5
```

<u>**Deep Copy**</u>

Q → Create deep copy of doubly linked list with random pointers.



```
x = Head
H2 = new Node (Head.data)  // Head != null
y = H2          x = x.next
while (x != null) {
    y.next = new Node (x.data)
    y = y.next
    x = x.next
}
return H2
```



∀nodes → store cope reference   HashMap

<node, node>

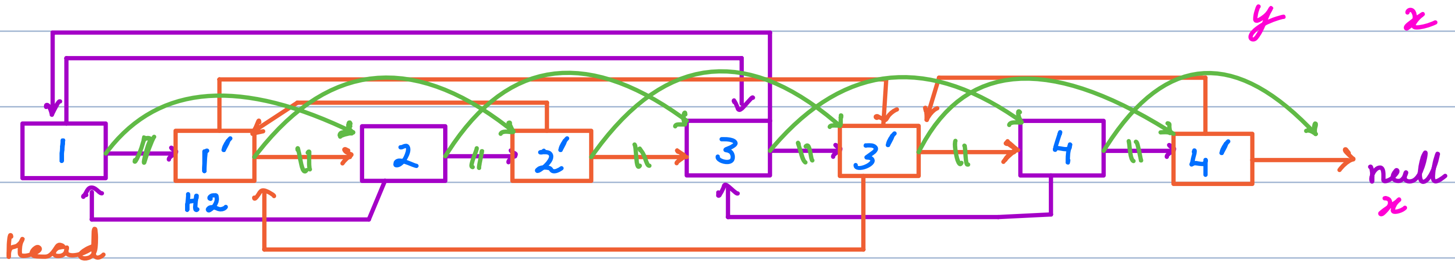
original → copy

∀nodes in copy list x,

x.random = mp.get(origral_x.random)

TC = $O(N)$ SC = $O(N)$

O(1)



```
① x = Head
  while (x != null) {
      y = new Node (x. data)
      y. next = x. next
      x. next = y
      x = x. next. next //y. next
  }
```

```
② x = Head
  while (x != null) {
     x. next. random
      = x. random. next
     x = x. next. next
  }
```

```
③   H2 = Head. next
    x = Head        y = H2
    while (x != null) {
        x. next = y. next
        x = x. next
        if (x != null) { y. next = x. next
                         y = x. next }
    }
    return H2
```

TC = O(N)   SC = O(1)