① Deep clone
② LRU cache

Doubly

SLL

```
class Node {
    int data
    Node next
    Node prev
    Node(int x) {
        data = x
        next = NULL
        prev = NULL
    }
}
```

LRU Cache

↓

Least recently used

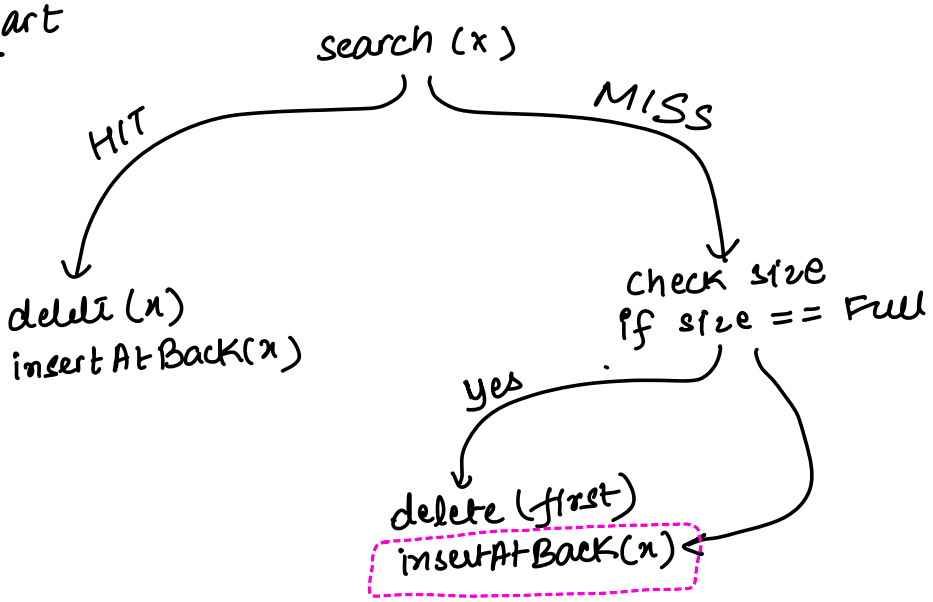
Data :

7	3	9	2	6	10	14	2	10	15	8	14
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑

Cache capacity → 5

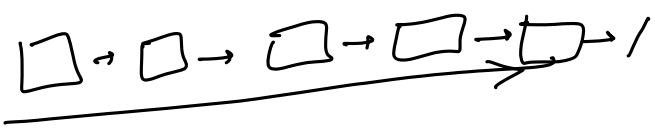
~~7~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ ~~10~~ ~~14~~ 2 10 15 8 14

Flow chart



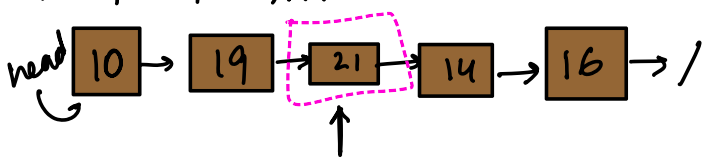
checksize ( )
: Maintain a variable to keep track of the size.

	Arrays	Linked List	LL + HashMap<int, Node>
search(x)	O(N)	O(N)	O(1)
delete(x)	O(N)	O(N)	O(1)
insertAtBack(x)	O(1)	O(1)	O(1)



Maintain order.

10, 19, 21, 14, 16, 21



Hashmap

<int, Node>

10 ,	ref(10)
19,	ref(19)
21,	ref(21)
14,	ref(14)
16,	ref(16)

n = 21

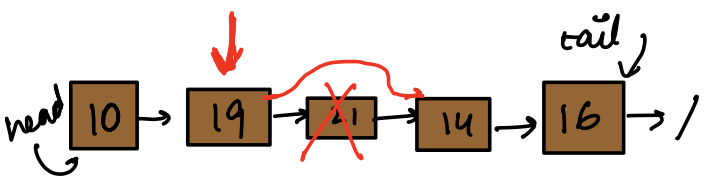
Node temp = hm[x]

temp = ref(21)

address of prev node

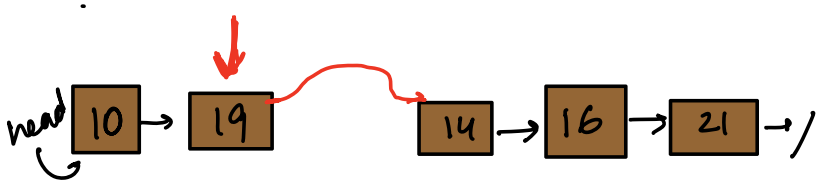
<int, Node>

10 ,	NULL
19,	ref(10)
21,	~~ref(19)~~
14,	~~ref(21)~~ ref(16)
16,	ref(14) ref(19)

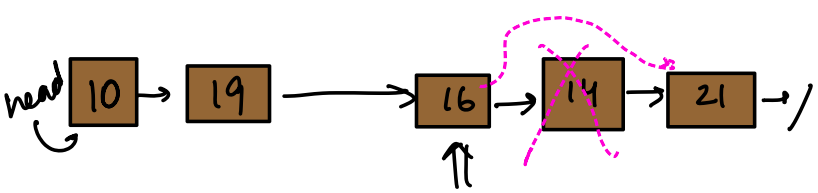


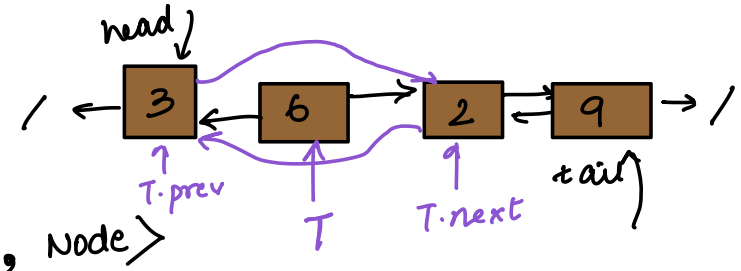
n=21

Node t = hm[n]



SC: O(N)





HashMap<int, Node>

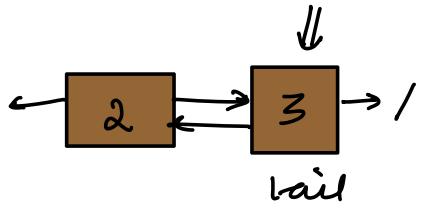
3, ref(3)
6, ref(6)
2, ref(2)
9, ref(9)

We don't have to change the reference upon removing the node.

```
void insertAtBack (Node newNode) {
    tail.next = newNode
    newNode.prev = tail
    tail = tail.next
}   tail.next = NULL
```

If data is already present in the list

```
void remove (Node T) {
    Node p = T.prev
    Node n = T.next
    n.prev = p
    if (p != NULL) {
        p.next = n
    }
}
insertAtBack(T)
```



* If T.next = NULL then don't delete it.

Node is not already present

What if the node we are trying to insert is not present.

If size != FULL

⇓

insertAtBack(ref x)

hm.insert({x, ref(x)})

if size is full

⇓

delete (head)

// remove entry of head node.

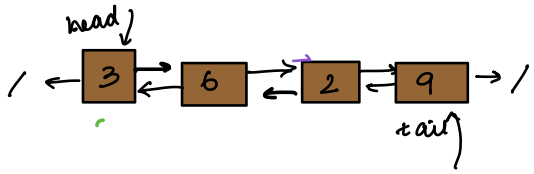
insertAtBack (ref x)

HashMap<int, Node> hm

head = NULL
tail = NULL

3, ref(3)
6, ref(6)
2, ref(2)
9, ref(9)

Data x is coming:



x = 6

if (x is in hm)

HIT

Node t = hm[x]
if (t.next == NULL)

True

{DO nothing}
Return

False

Not the last node

remove(T)
insertAtBack(T)

Miss

Node t = new Node(x)
if (hm.size() == capacity)

YES

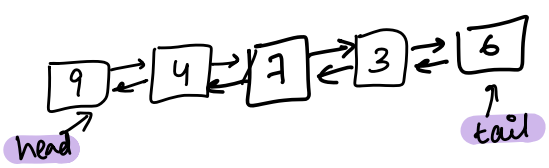
hm.delete(head.data)
delete(head)
insertAtBack(t)
hm.insert({x, t})

NO.

3, 4, 7, 9, 4, 7, 7, 3, 6

Capacity
5

DDL



TC: O(1) → for each operation
SC: O(capacity)

map

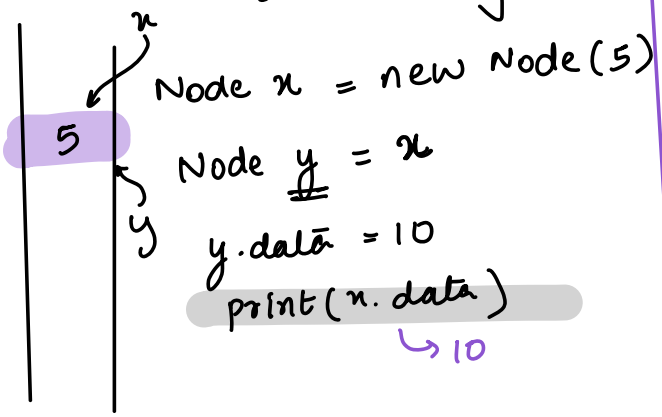
<3, ref(3)>
<4, ref(4)>
<7, ref(7)>
<9, ref(9)>
<6, ref(6)>

Break.

10:36 [complete break]

Doubts → [10:36]

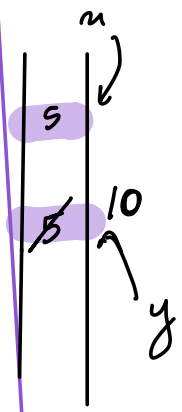
Shallow copy




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

↳ 10

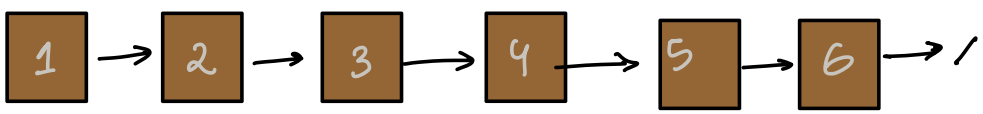
Deep copy




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

Que. Given a LL, create a deep copy of given LL.

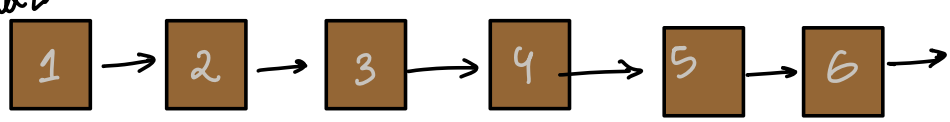
head ↓ x



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

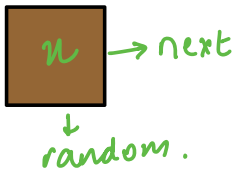
```
Node newNode = new Node (x.data)
head2 = newNode
```

head2

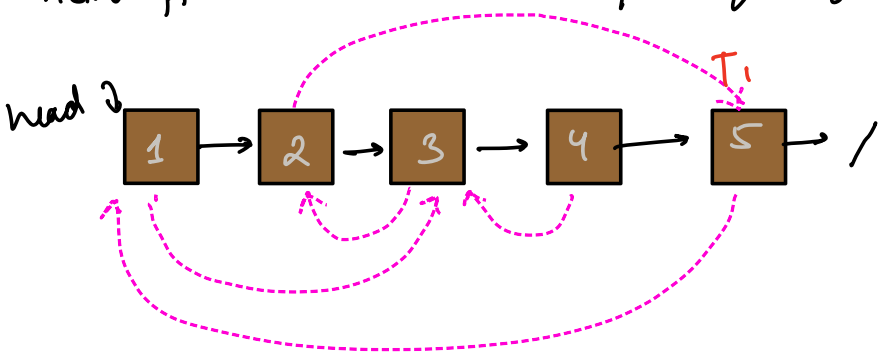


return head 2.

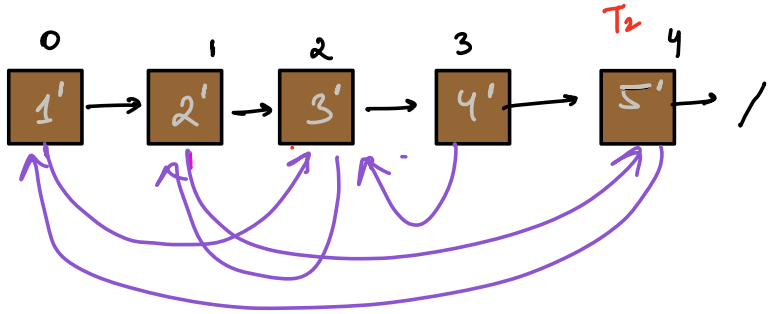
Follow Up



Given a node with a random pointer along with next ptr. Return the deep copy of the LL.



2 pointers
↓
next
random



TC: $O(N^2)$
SC: $O(1)$

Node r = T1.random

Node n = hm[3]

T2.random = n

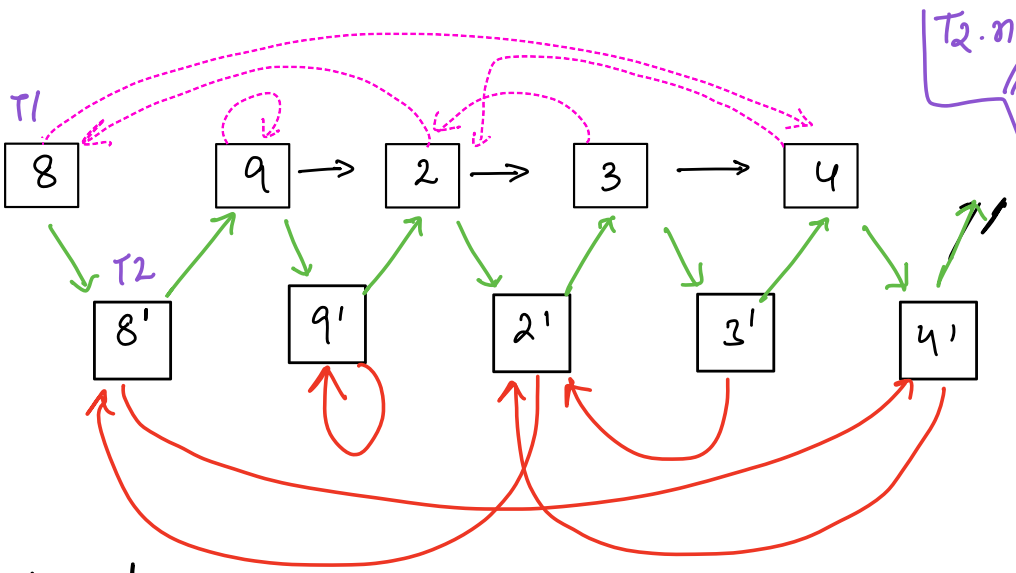
1 → 1'
2 → 2'
3 → 3'
4 → 4'
5 → 5

Steps

1. Create Deep copy using only next ptrs.
2. Create a hashmap mapping
3. Join random pointers.

TC: $O(N)$
SC: $O(N)$

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



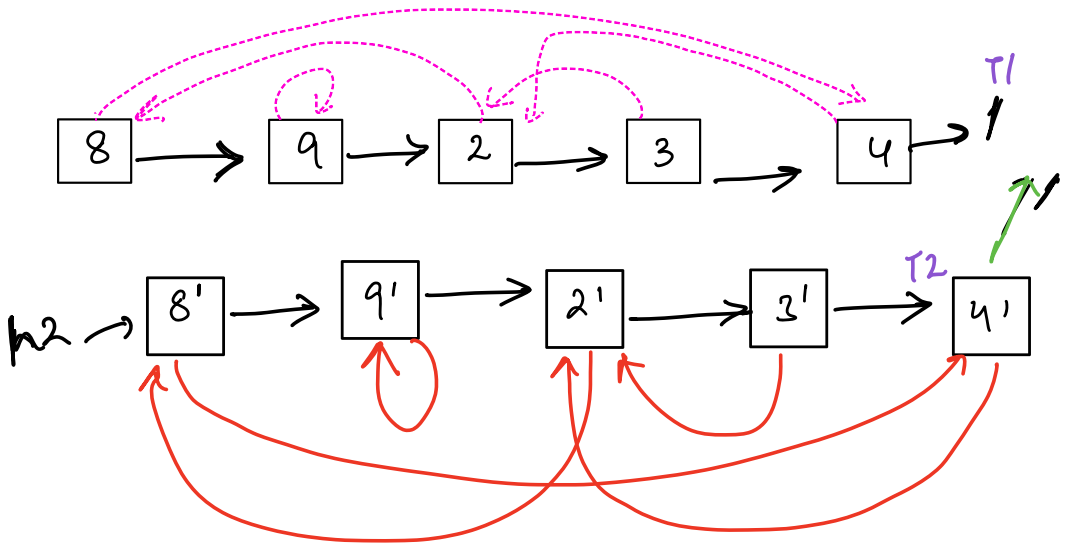
T2.next == NULL
// DON'T
↑ T2

Pseudocode

1) Insert each copy of a node in b/w [TO DO]
O(N)

2)
```
while (T1 != NULL && ){
    if (T1.random != NULL)
    {  T2.random = T1.random.next
    }
    else {T2.random = NULL }
    T1 = T1.next.next
    if(T2.next != NULL ){
    {    T2 = T2.next.next
    }
}
```

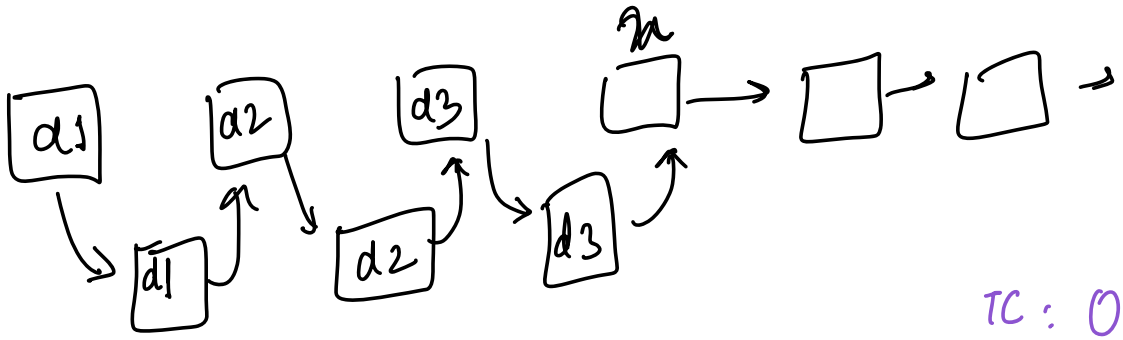
O(N)



3)

```
while ( T1 != NULL ) {
    T1.next = T2.next
    T1 = T1.next
    if (T1 != NULL) {
        T2.next = T1.next
    }
    T2 = T2.next
}
return h2
```

O(N)

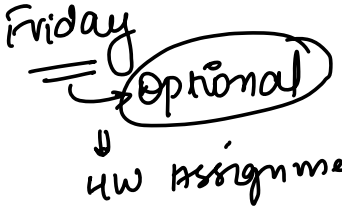


TC : O(N)

SC : O(1)

Applications

1. Cache - DLL
2. Used to implement Stacks & Queues.
3. Used to store / represent Graphs.



Monday → off.