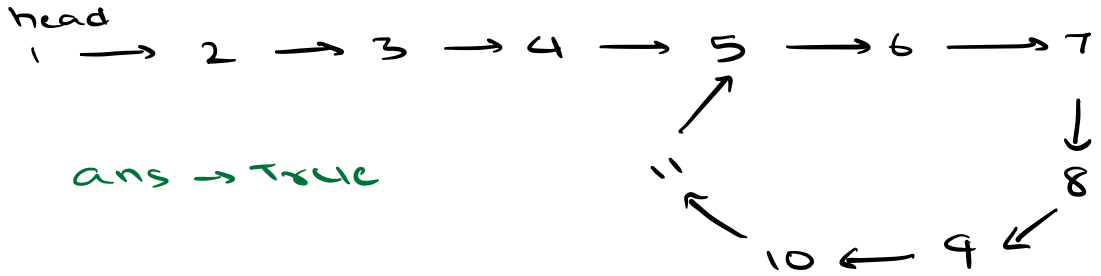
Agenda

1. Check for cycle in LL
2. Find starting point of loop
3. Introduction to Doubly LL
4. Insertion / Deletion in Doubly LL
5. LRU Cache

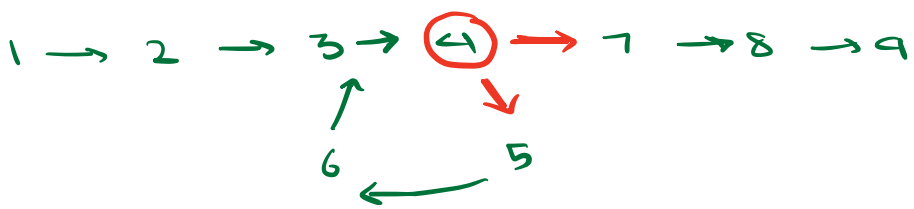
Clone a LL → separate recording will be added

1. Detect cycle in LL (nodes with duplicate data allowed)



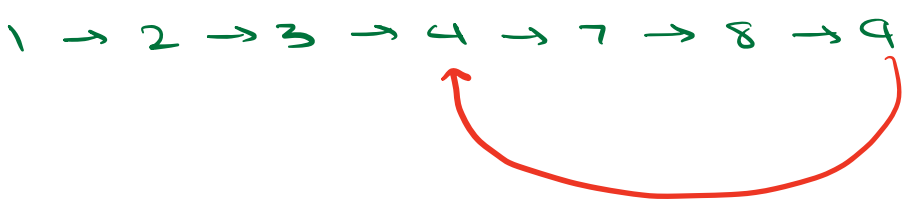
ans → True

ans → False



NOTE: Loop cannot be in middle as one node will have 2 neigbours which is not possible

Conclusion : Loop (if present) at end of LL



Idea 1: Start a **temp** at the head and **traverse LL** using temp

if temp **reaches NULL** ✗

- Yes → No cycle
- No → cycle (TLE)

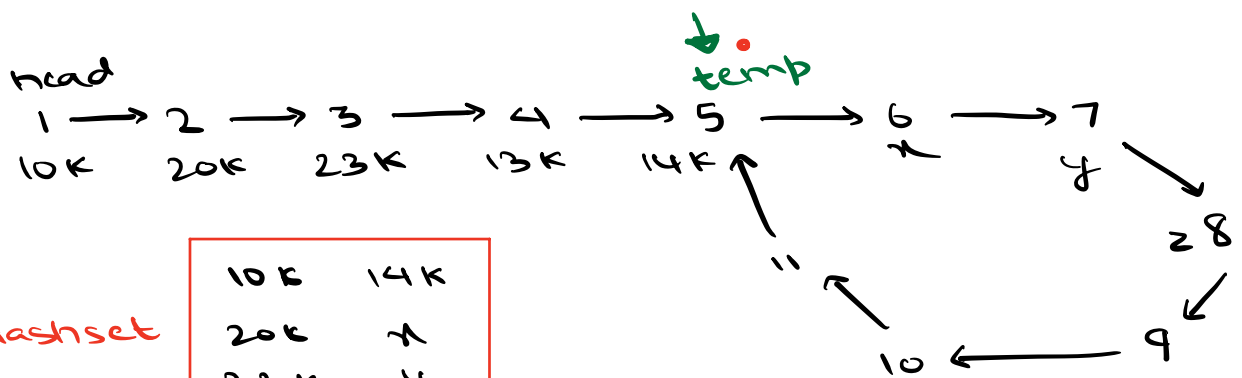
Hashset <Node>

Idea 2: Iterate LL, whatever node we visit → put them in HS

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

If we find any node already in HS → cycle in LL

End of LL → No cycle in LL



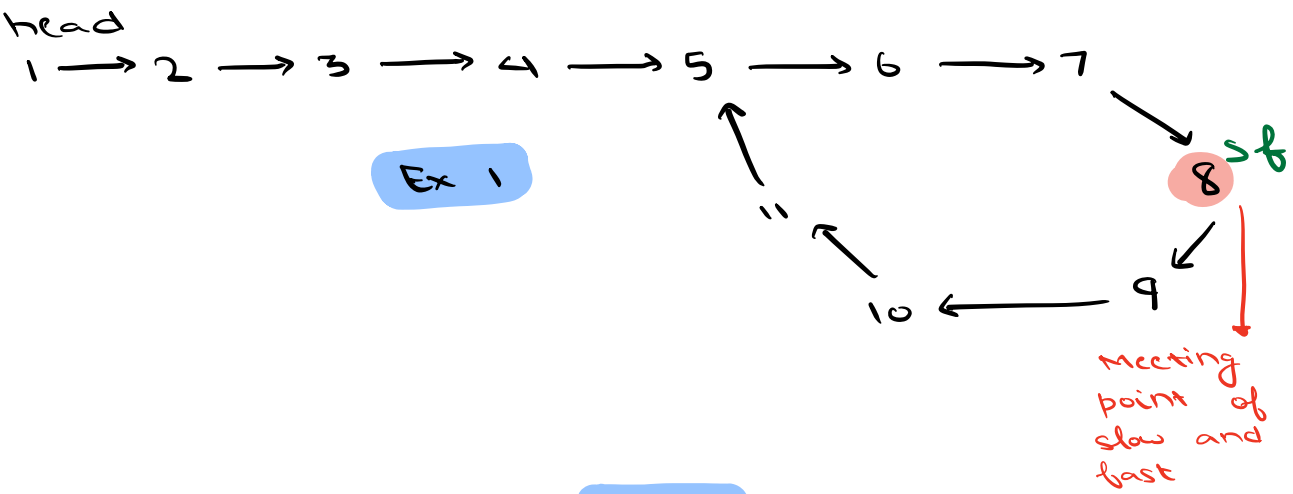
Hashset

10K	14K
20K	x
23K	y
13K	z

- When we reach node 5 the 2nd time, its address is already in HS ; suggesting a loop in LL.

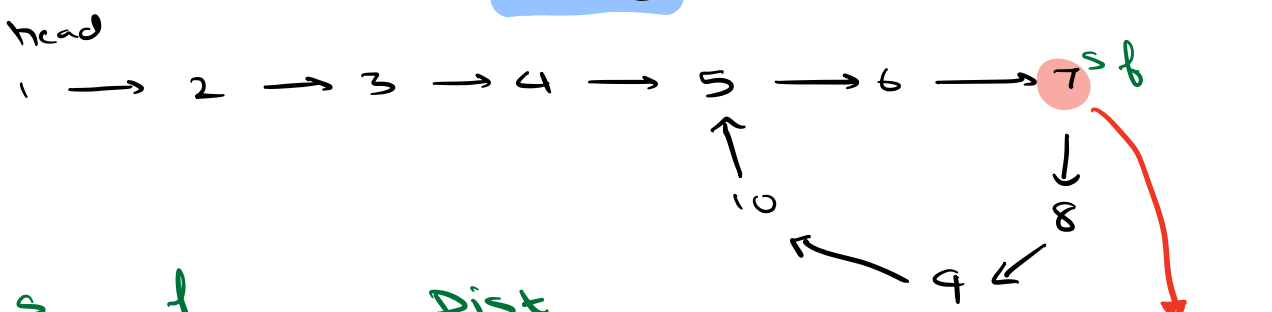
Idea 3: Using slow and fast ref variable

Keep s and f at head. At a time move s by 1 step (s=s.next) and f by 2 steps (f= f.next.next). If they meet → cycle



Ex 2

Ex 3



s	f	Dist
5	9	2 → reduced by one
6	5	1 → reduced by one
7	7	0

When slow ptr enters cycle, lets say dist b/w them is x. Now we observe that after every step, dist. reduces by 1, hence dist. eventually becomes 0.

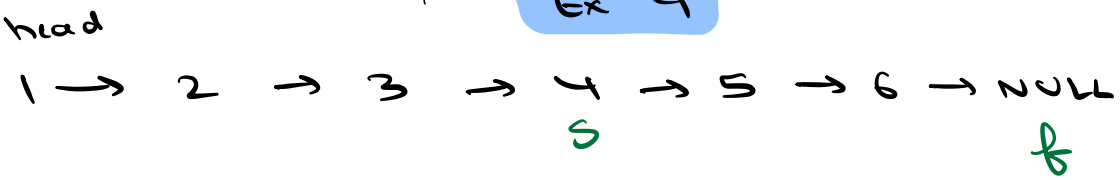
When to **break the loop**?

Ex 2

(a) In this case, when **f reaches last node**, now it cannot move by 2 steps, so stop traversal

---

Ex 4



(b) In this case, when **f reaches NULL**, now it cannot move by 2 steps, so stop traversal

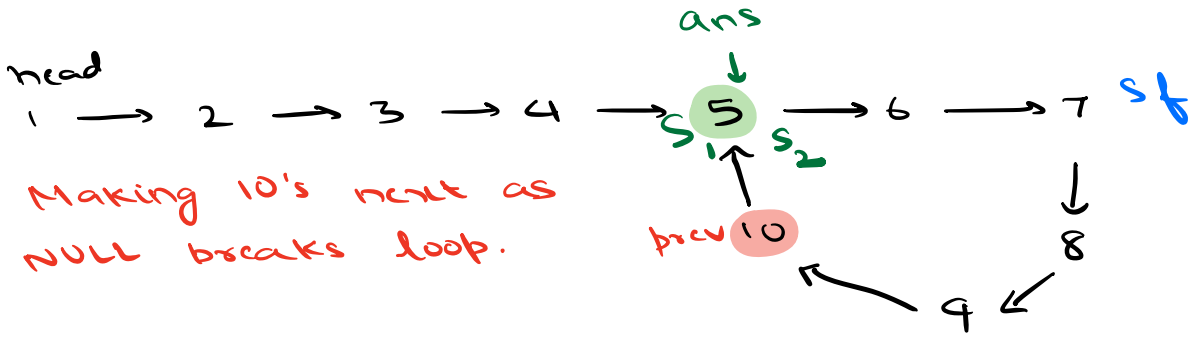
---

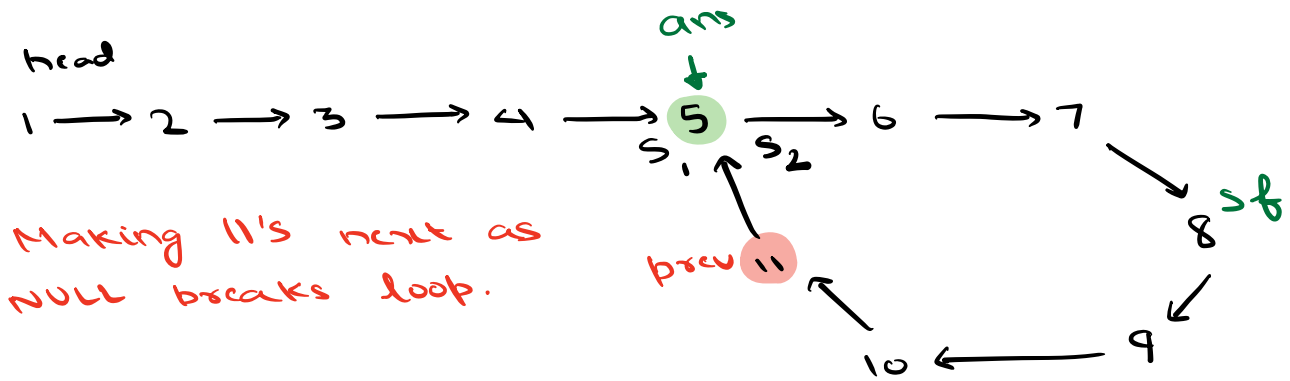
```
while (f != NULL && f.next != NULL){
    s = s.next
    f = f.next.next
    if (s == f){
        return true
    }
}
```

return false

---

2. If there is a loop in LL, find starting point of the loop.





Take 2 ref variables s1 and s2

Node s1 = head

Node s2 = Intersection of s and f

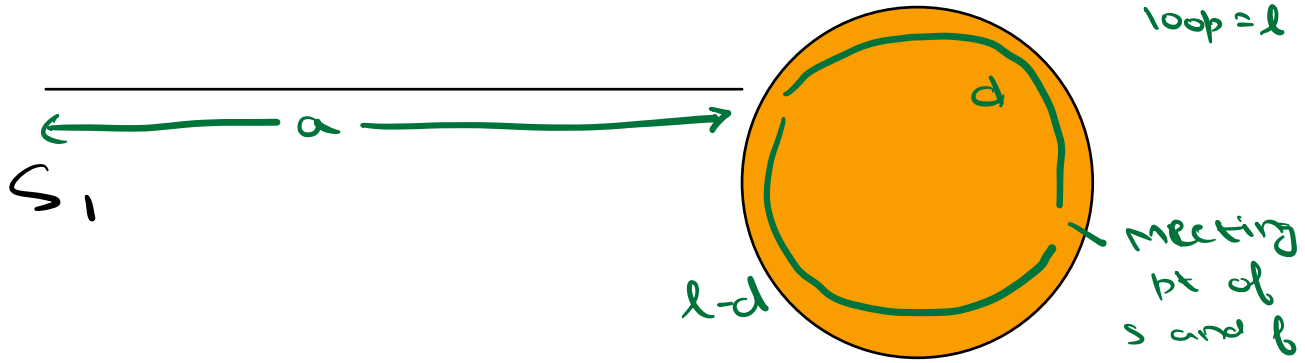
Move both s1 and s2 1 step at a time.

Their meeting point → start of loop

```
bool detectcycle(Node head){
    Node s = h
    Node f = h
    bool isCycle = false

    while(f != NULL && f.next!=NULL){
        s = s.next
        f = f.next.next
        if(s==f){
            isCycle = true
            break
        }
    }

    is(isCycle == false)
        return false

    Node s1 = head, s2 = s/f
    Node prev = NULL → Maintain prev of s2 as when s2 reaches starting pt of loop, prev node in loop's next → NULL
    while(s1 != s2){
        s1 = s1.next
        prev = s2
        s2 = s2.next
    }

    Node loopstart = s1
    prev.next = NULL
}
```

↓

break loop



$d_s = a + d$

$d_f = a + c \times l + d$

$$d_f = 2 \times d_s$$

$$a + cl + d = 2a + 2d$$

$$cl = a + d$$

$$a = cl - d$$

$$\Rightarrow a = cl - d \underline{-l + l}$$

$$a = cl - l + l - d$$

$$a = l(c-1) + l - d$$

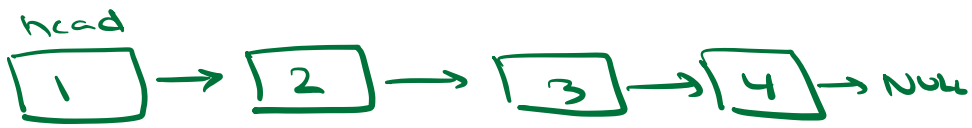
$$a = \underline{l n + l - d}$$

a ↓ : Dist travelled by S1

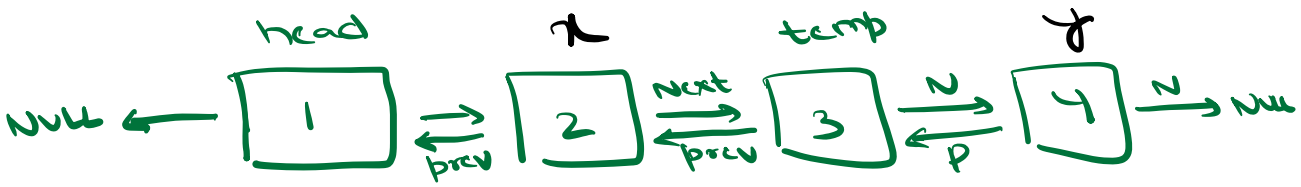
$l n + l - d$ ↓ : Dist travelled by S2

When S1 covers 'a' dist, it reaches loop start. When S2 covers the loop some no. of times, it reaches original pos. On covering '$l-d$', it reaches loop start.

Singly LL



Doubly LL

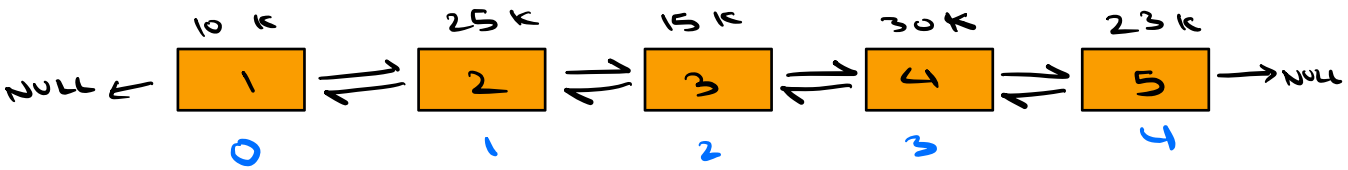



```
Class Node {
    int data
    Node next
    Node prev
}
```

→ right
→ left

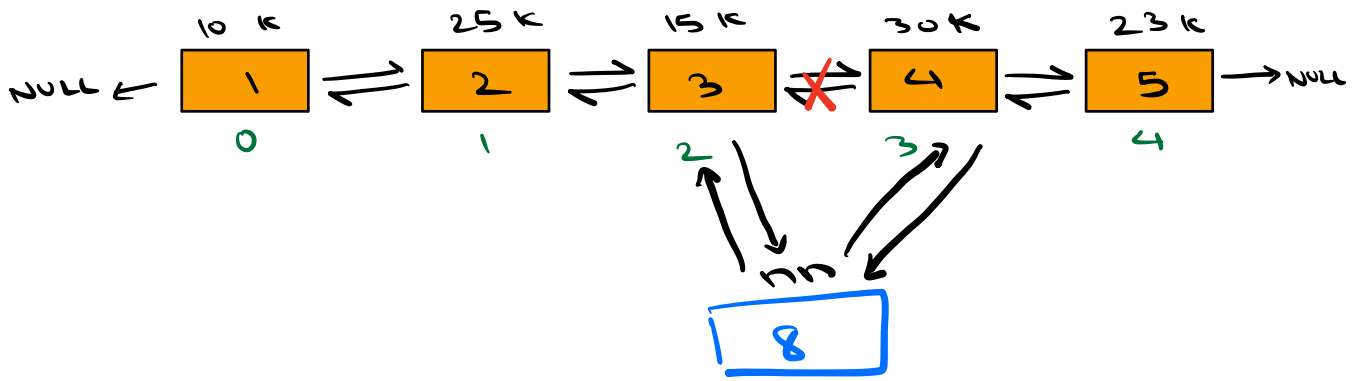
```
Node x = temp.prev
Node y = temp.next
```

3. A doubly linked list is given. A node is to be inserted with data X at position K.
The range of K is between 0 and N
where N = length of doubly LL.

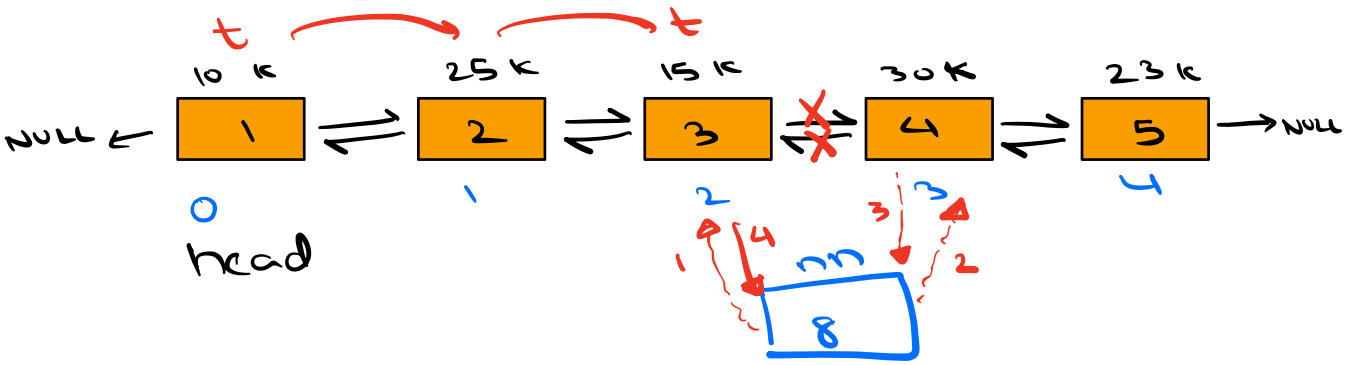


K → 0 to N
K → 0 to 5

X = 8, K = 3



Node nn = new Node(x)



k = 3

Stop at k-1 i.e. 2

↓

temp initially at head

jump 2 times

Update nn links {

nn.prev = temp ①

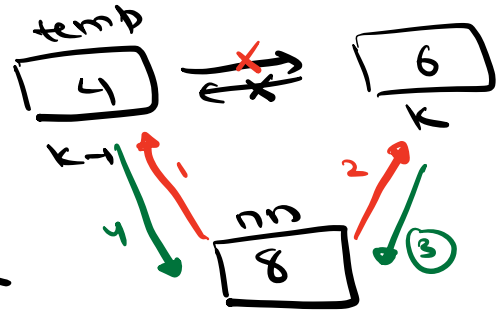
nn.next = temp.next ②

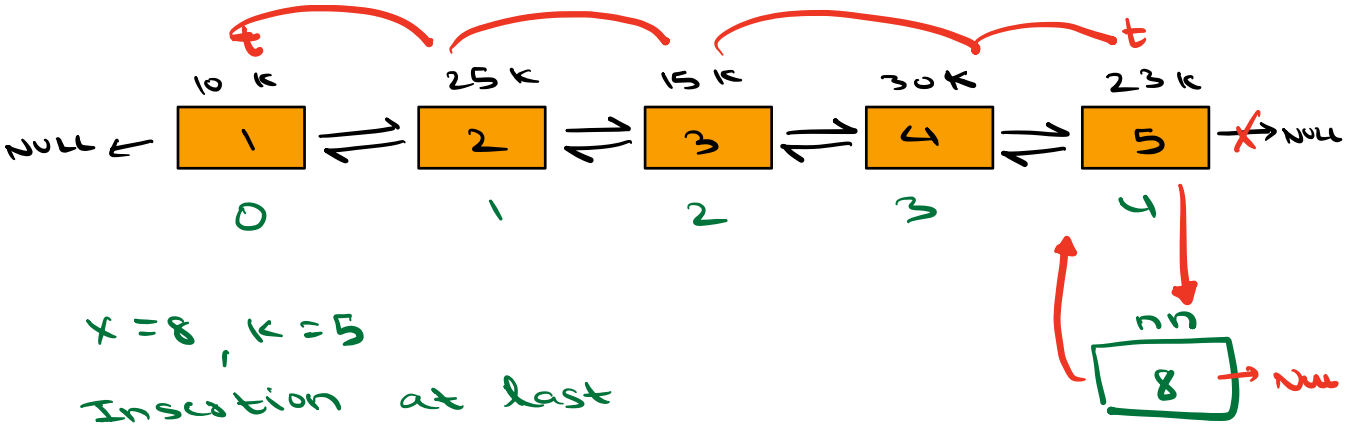
temp.next.prev = nn ③

temp.next = nn ④

```
Node nn = new Node(x)

//Reach k-1 th Node
Node temp = head

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

```
1  nn.prev = temp
2  nn.next = temp.next
3  if (temp.next != NULL)
       temp.next.prev = nn
4  temp.next = nn
```





x = 8, k = 5

Insertion at last works fine

```
Node nn = new Node(x)
if (head == NULL)   return nn

if (k == 0) {
    nn.next = head
    head.prev = nn
    head = nn
    return head
}

// Reach k-1th node
Node temp = head

for (i = 0 ; i < k-1 ; i++) {
    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(N)

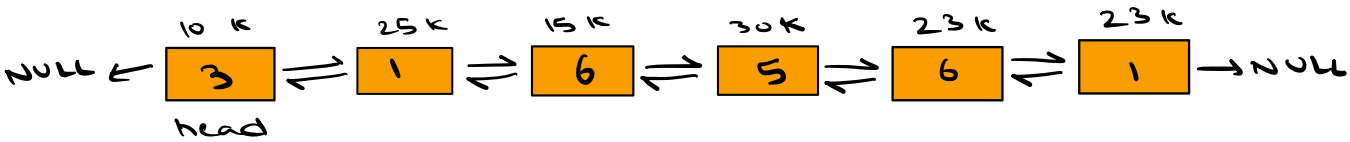
SC : O(1)

For insertion, edge cases :

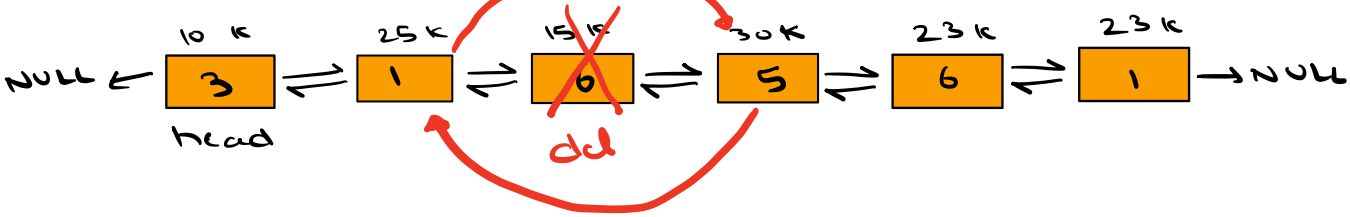
a) LL is empty → head is NULL

b) Insertion at head → k=0

4. Given a doubly linked list of length N, we have to delete first occurrence of data x from given list. If x is absent, don't do anything.



x = 6




```
Node temp = head

while (temp != NULL){
    if (temp.data == x){
        break
    }
    temp = temp.next
}

if (temp == NULL)
    return head

// else delete
```

Find node to be deleted

Reached end node but could not find x, hence no deletion

```
Node tp = temp.prev
Node tn = temp.next
```

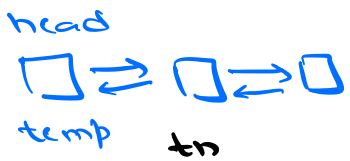
① If LL has 1 node and that is deleted

```
if (tp == NULL && tn == NULL) {
    head = NULL
    return NULL
}
```

C++ → free(temp)

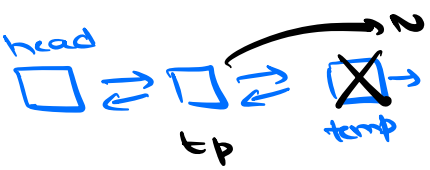
② Deleting head of LL

```
else if (tp == NULL) {
    tn.prev = NULL
    head = tn
    return head
}
```



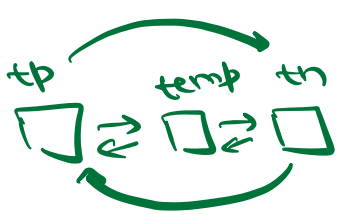
③ Deleting last node of LL

```
else if (tn == NULL) {
    tp.next = NULL
    return head
}
```



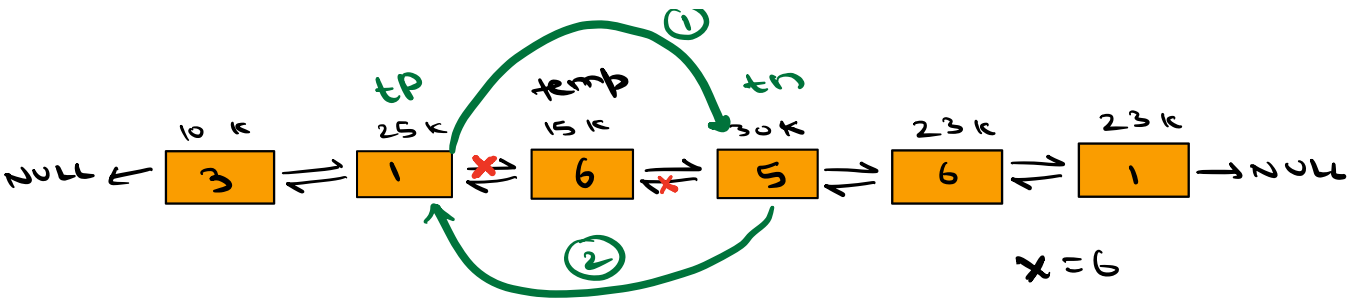
④ Delete intermediate node

```
else {
    tp.next = tn
    tn.prev = tp
    free(temp)
    return head
}
```

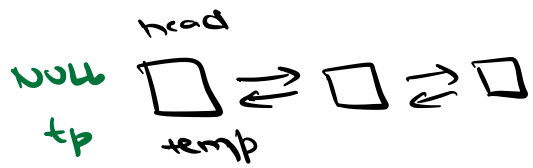


TC : O(N)

SC : O(1)




```
Node tp = temp.prev
Node tn = temp.next
tp.next = tn
tn.prev = tp
free (temp)
```



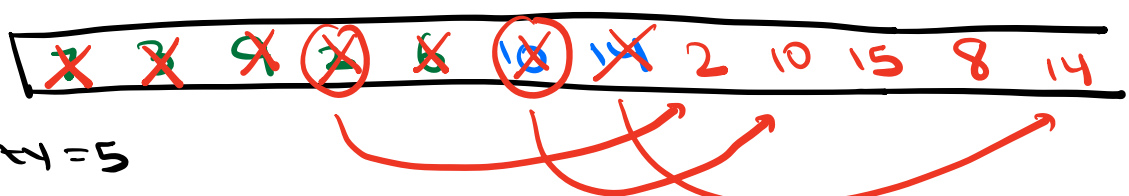
MOST RECENT INTEGER

LRU Cache → temporary storage → small

↓

Delete Least Recently Used

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



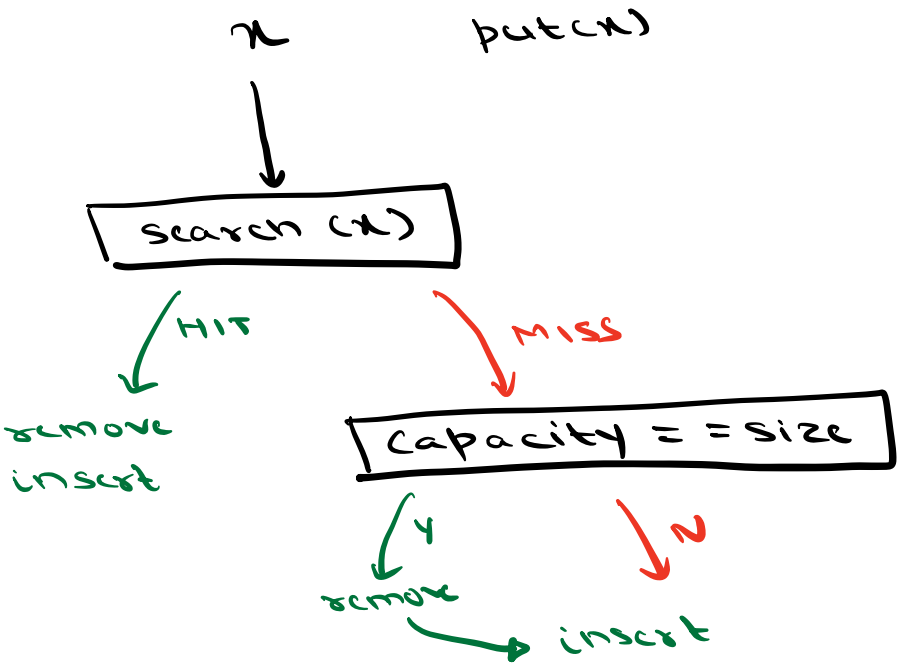
Capacity = 5

size = 0 1 2 3 4 5

present in cache → HIT

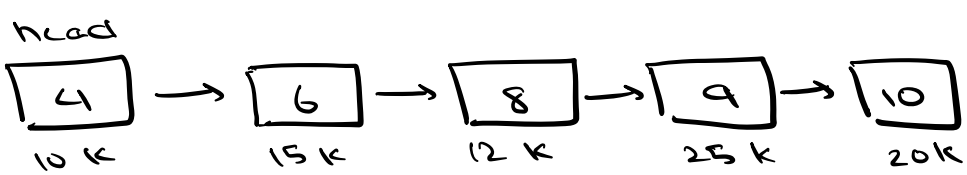
not present in cache → miss

search
insert
delete



	array	LL	HM	LL+HM
search	O(N)	O(N)	O(1)	O(1)
insert	O(1)	O(1)	O(1)	O(1)
delete	O(N)	O(1)	order of insertion not maintained	O(N)

Del in LL → O(N)



capacity = 5

Cache

4, 10K
6, 15K
8, 12K
9, 25K
10, 20K

When data 8 comes → HIT → del 8 at 12K
but 6.next has to be updated.

Iterating to reach prev of 8 takes O(N)
Hence deletion in LL is O(N).

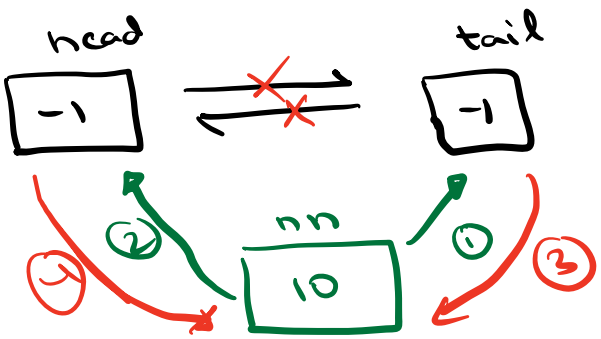
Reaching prev from node 8 is O(1) if we take DLL

	DLL + HM
search	O(1)
insert	O(1)
delete	O(1)

10 15 19 20 15 18 23 20 19 17 17

capacity = 5

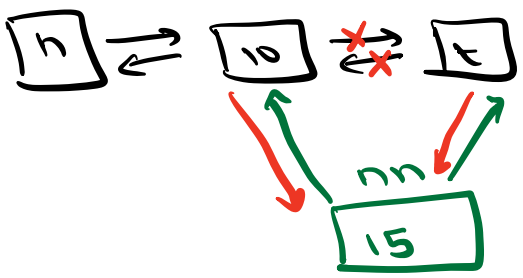
DLL



HM
<int, Node>
10, addr of 10

```
add To Tail (Node nn) {
    nn.next = tail                  (1)
    nn.prev = tail.prev             (2)
    tail.prev = nn                  (3)
    nn.prev.next = nn               (4)
}
```

10 15 19 20 15 18 23 20 19



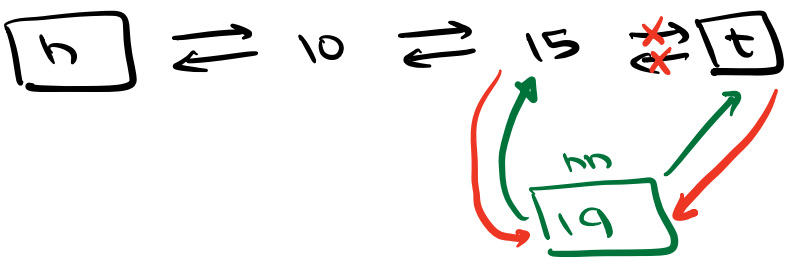
Add 15

Map

10, addr of 10

15, addr of 15

Add 19



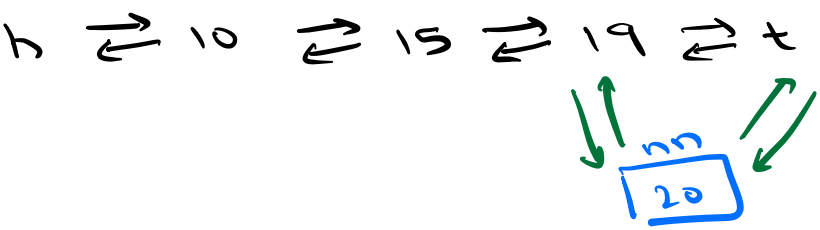
Map

10, addr

15, addr

19, addr

Add 20



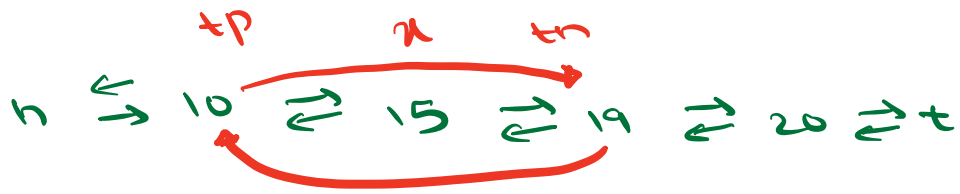
10, addr

15, addr

19, addr

20, addr

Add 15

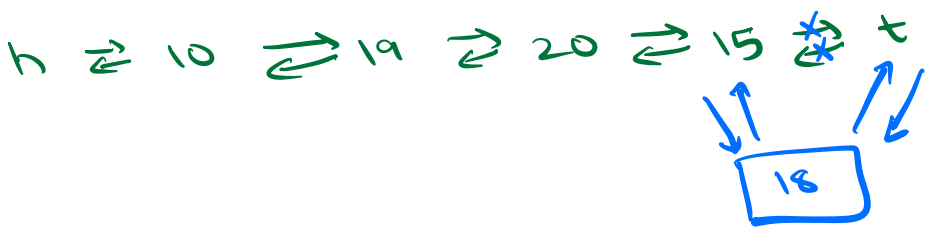


→ remove (Node x)<

```
tp = x.prev
tn = x.next
tp.next = tn
tn.prev = tp
```

→ add to Tail (x)

Add 18



HM

10, addr

15, addr

19, addr

20, addr

18, addr

Add 23

① remove (h.next)
// del 10 node

HM

~~10, addr~~

15, addr
19, addr
20, addr
18, addr

② hm.delete(10)
③ add to tail (23)

h ⇄ 19 ⇄ 20 ⇄ 15 ⇄ 18 ⇄ 23 ⇄ t

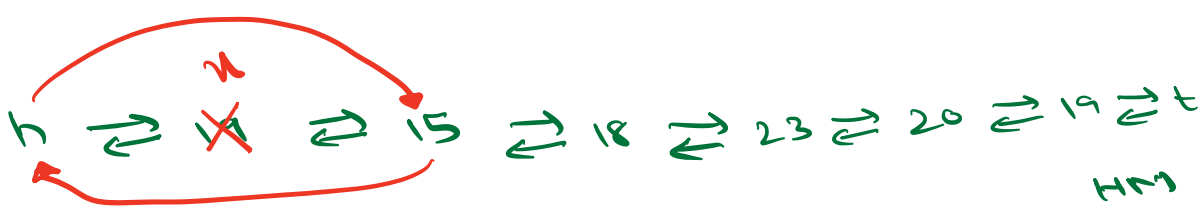
Add 20

Node x = hm.get(20)
remove(x)
insert Totail(x)

HM
15, —
19, —
20, —
18, —
23, —

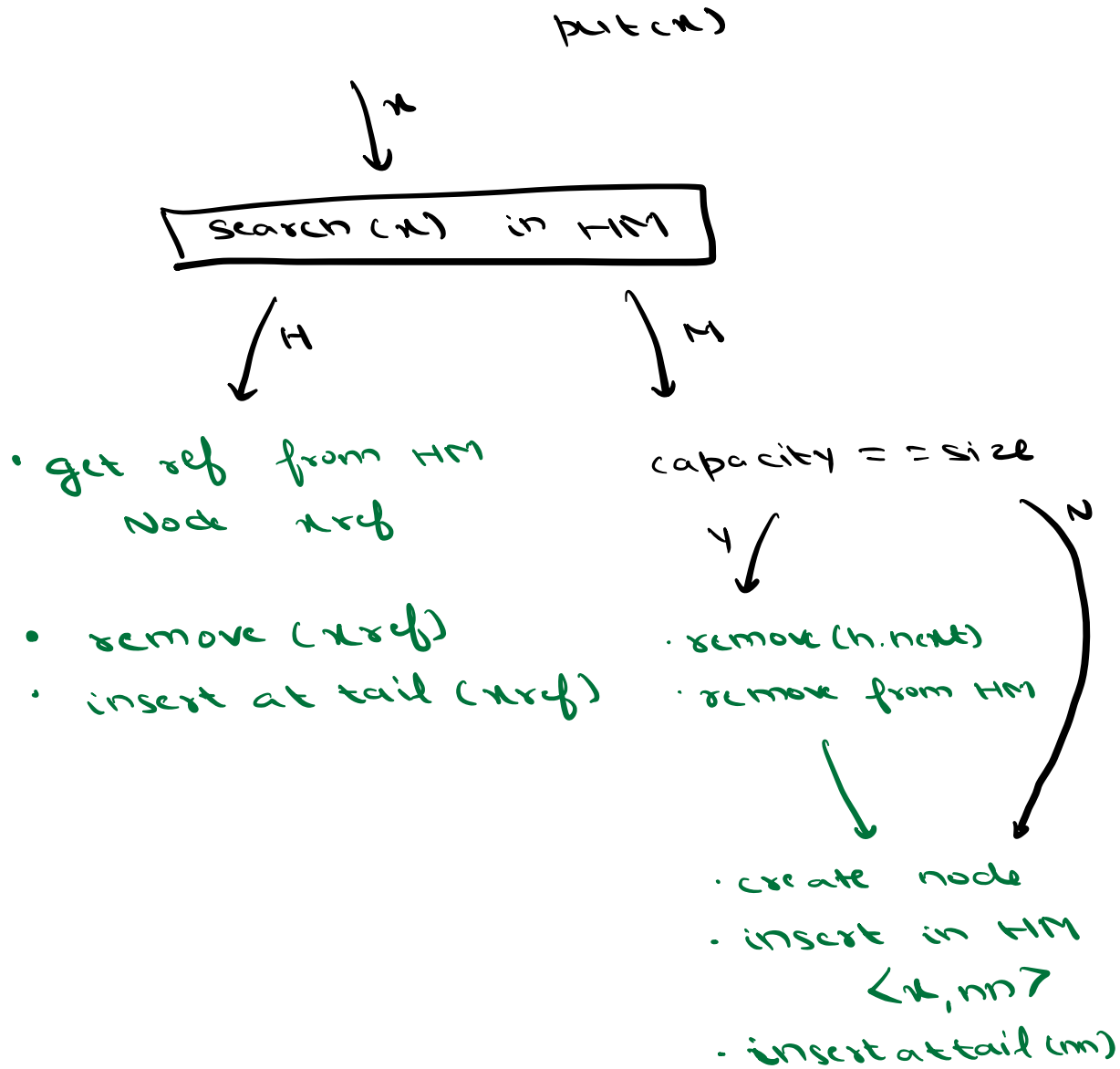
h ⇄ 19 ⇄ 15 ⇄ 18 ⇄ 23 ⇄ 20 ⇄ t

Add 19



Node x = hm.get(19)
remove(x)
insert Totail(x)

HM
15, —
19, —
20, —
18, —
23, —


put(x)
x
search(x) in HM
H
M
• get ref from HM Node xref
• remove (xref)
• insert at tail (xref)
capacity == size
Y
N
. remove (h.next)
. remove from HM
. create node
. insert in HM <x, nn>
. insert at tail (nn)

```
Node h = new Node(-1)
Node t = new Node(-1)
h.next = t
t.prev = h
```


```
Hashmap<int, Node> hm

LRU(int d, int cap) {
    if (hm.search(d)) {
        Node t = hm[d]
        remove(t)
        add to tail(t)
    }
    else {
        if (hm.size() == cap) {
            Node del = h.next
            hm.delete(del.data)
            remove(del)
        }
        Node nn = new Node(d)
        add to tail(nn)
        hm.insert(d, nn)
    }
}
```

```
remove (Node x) {
    tp = x.prev
    tn = x.next
    tp.next = tn
    tn.prev = tp
}
```

```
addToTail (Node nn) {
    nn.next = tail
    nn.prev = tail.prev
    tail.prev = nn
    nn.prev.next = nn
}
```
