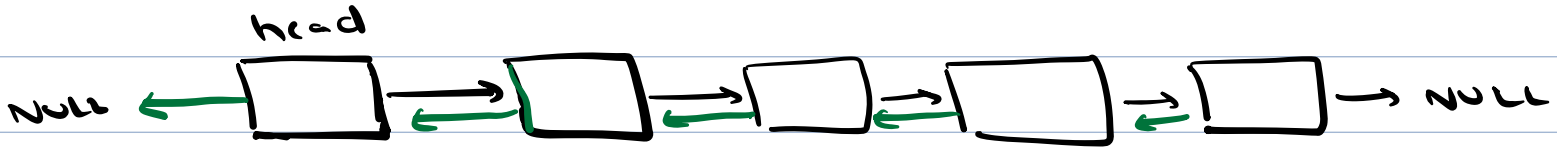
Agenda

1. What is doubly LL
2. How is doubly LL different from singly LL
3. Problems
   - Insert a node
   - Delete a node
   - Cache
   - Clone LL

Linked list is a data structure to store and organize a collection of elements (in nodes).

Doubly Linked List

A doubly linked list is similar to a singly linked list but with an additional feature each node contains references to both next & previous nodes in the list.




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

prev = NULL

① head.prev = NULL

② next of last node = NULL

Correlation with Singly Linked List

The main difference lies in the bidirectional traversal capacity of doubly linked list (which comes at a cost of increased memory usage)

* prev → left ; next → right

Implement playlist feature using doubly LL

↓

① Add song

② Skip to prev and next song

<< (II) >>

③ current song

Add(1, "Love story") ✓

Add(2, "Imagine") ✓

Play Next ✓

Current Imagine

Add(3, "California") ✓

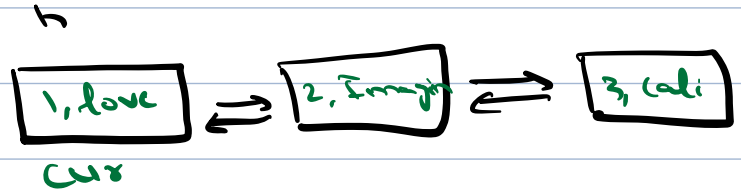
Play Next ✓

Current ✓ Cali

Play Previous ✓

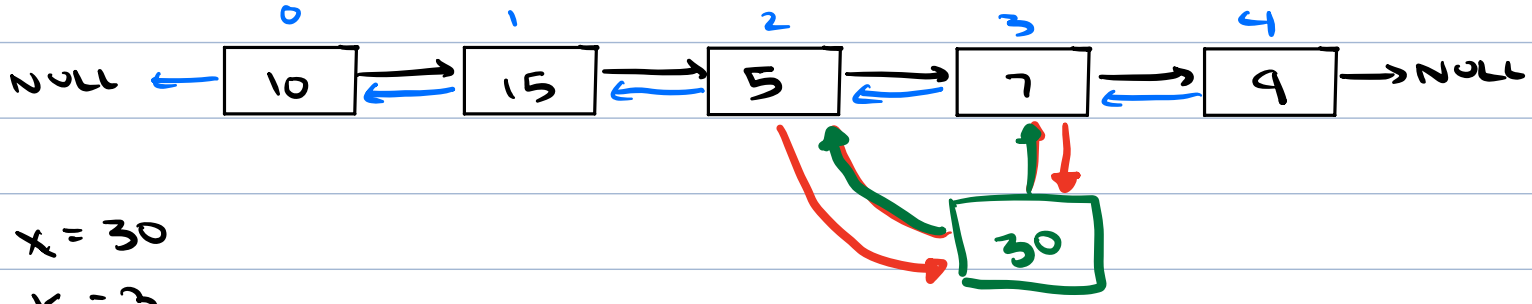
Play Previous ✓

Current Love



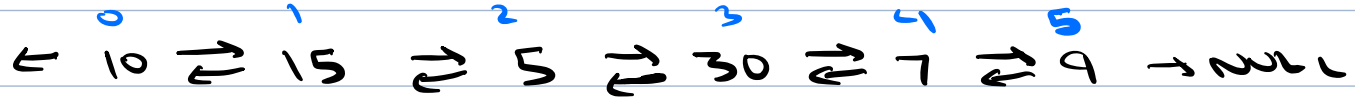
1. Insert node in a doubly linked list

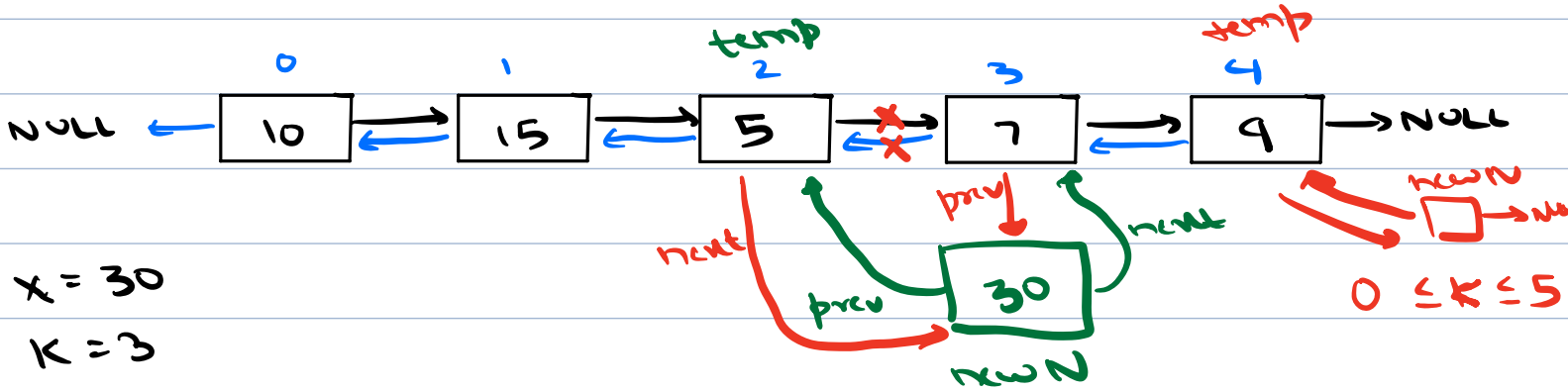
A node is to be inserted with data x at position k. Range of k is between 0 and N where N is length of doubly LL.



x = 30

k = 3





x = 30

k = 3

① Create newNode with data x

② Traverse till k-1 idx → temp

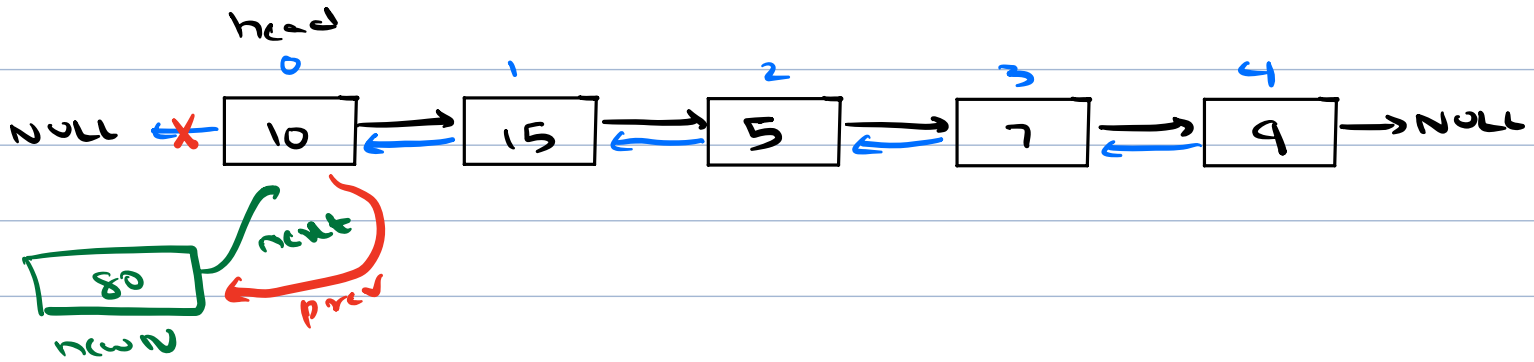
③ newN.prev = temp

newN.next = temp.next

④ temp.next = newN

if (newN.next != NULL)

newN.next.prev = newN



x = 80

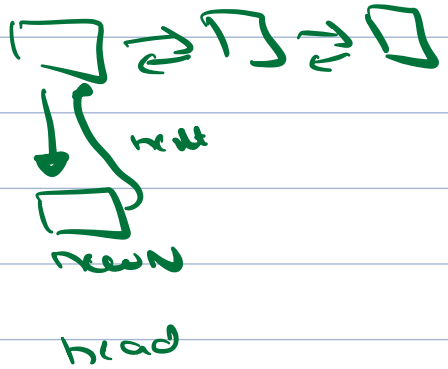
p = 0

① newN. next = head

② head. prev = newN

③ head = newN

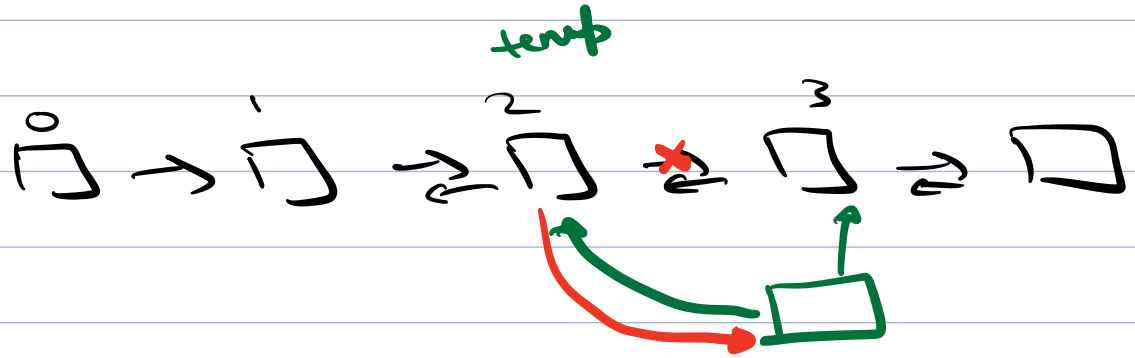
```
Node insert (Node head, int x, int k) {

    Node newN = new Node (x)
    if (head == NULL) {
            return newN
    }

    if (k == 0) {
        newN. next = head
        head. prev = newN
        head = newN
        return head
    }
    else {
        Node temp = head          → (k-1) idx
        for (j=1 ; j <= k-1 ; j++)
                temp = temp. next
```



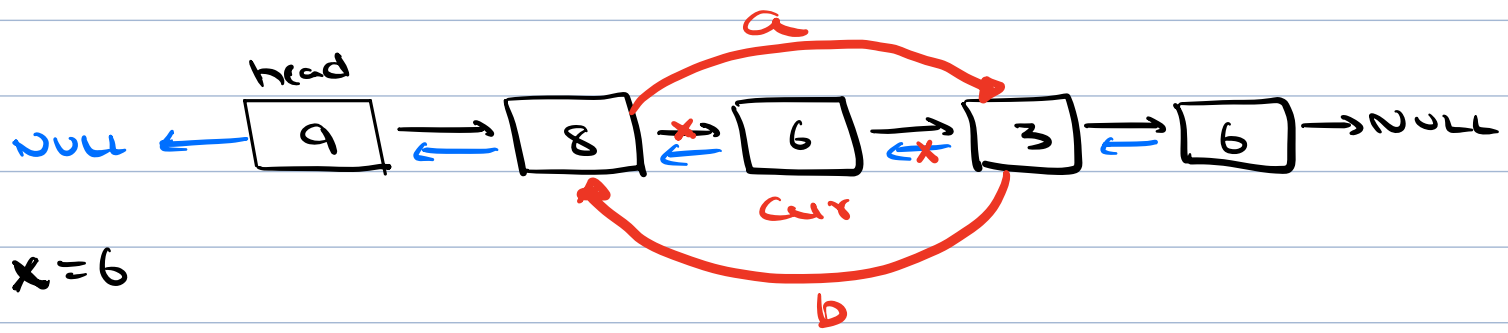
```
newN.prev = temp
newN.next = temp.next
temp.next = newN
if (newN.next != NULL)
newN.next.prev = newN
return head
}
```

TC : O(K) → O(N) (worst case)

SC : O(1)

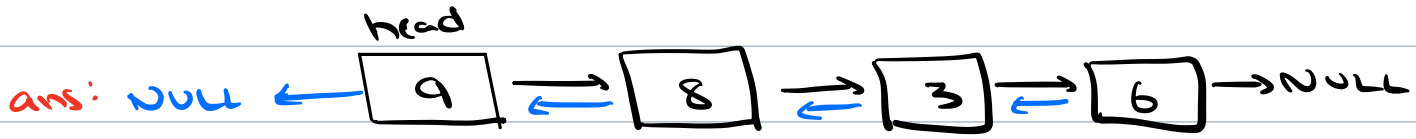


2. Given a doubly LL of length N, delete 1st occurrence of data x from given LL. If x is absent, don't do anything.



x = 6

ans:



① If x is not present

No change → return head

② If x is present at middle

Node cur (del node)

Node prevNode = cur.prev

Node nextNode = cur.next

ⓐ prevNode.next = nextNode

ⓑ nextNode.prev = prevNode

free (cur)

③ If x is at head

head = head.next

head.prev = NULL

④ If x is at tail

Node prevNode = cur.prev

prevNode.next = NULL

⑤ If x is the only node present in LL

return NULL

```
Node delete(Node head, int x) {

    // search for node with data x
    Node cur = head
    while (cur != NULL) {
        if (cur.data == x)
            break
        cur = cur.next
    }
    if (cur == NULL)        // x is not found
        return head         or empty LL
```

```
// single node
if ( cur.prev == = NULL  &&  cur.next = = NULL) {
        return  NULL
}
else if ( cur.prev = = NULL ) { // head
        head = head.next
        head.prev = NULL
        return head
}
else if ( cur.next = = NULL) { // last
        cur.prev.next = NULL
        return head
}
else {
    Node prevNode = cur.prev
    Node nextNode = cur.next
    prevNode.next = nextNode
    nextNode.prev = prevNode
    return head
}
}
```

TC: O(N)

SC: O(1)

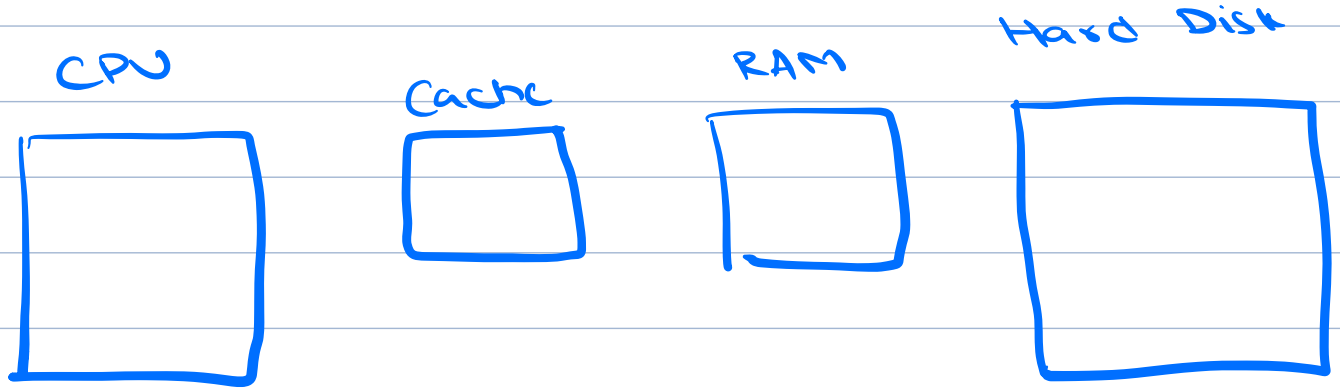
10:22

3. Given a running stream of integers and fixed memory size of M, we've to maintain most recent M elements. In case current memory is full, delete the least recent element and insert current data into the memory (as the most recent item).

LRU Cache

Eg 10 15 19 20 18 23 20 19 17 17 10

M = 5



Memory → size M

Data

HIT

MISS

① Search

② Order in data

③ Delete

↓ Start ↓ Middle

④ Insert

↓ Last

① Remove de from its pos

② Add to last

size == capacity

Y ↓ N ↓

① Remove starting de

↓ Least recent de

② Add to last

Remove starting de

Add to last

DLL + Hashmap

↓ Insert, Delete, Order

↓ Search

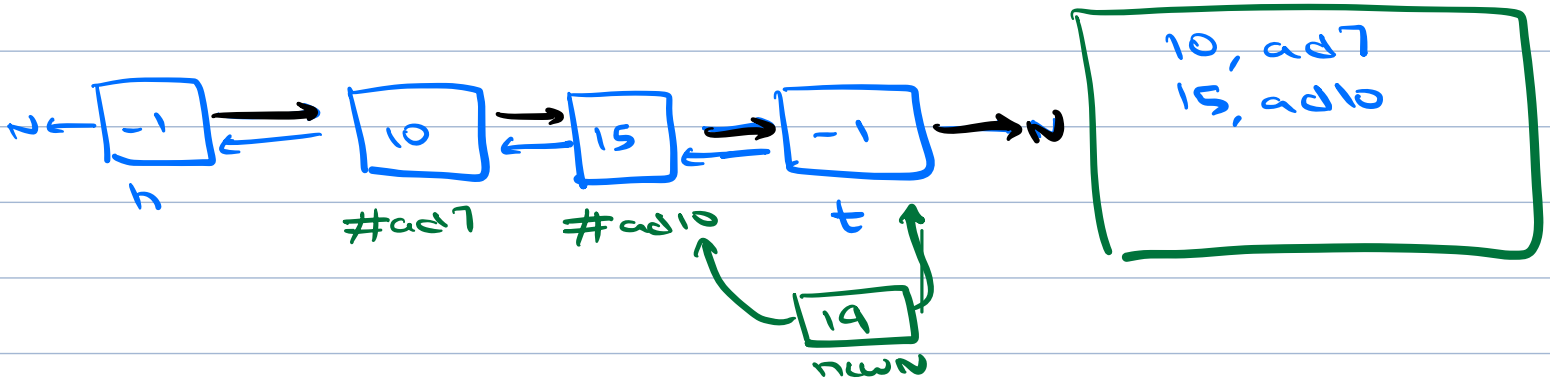
HM <data, Node>

Eg 10 15 19 20 18 23 20 19 17 17 10

M = 5

Doubly LL

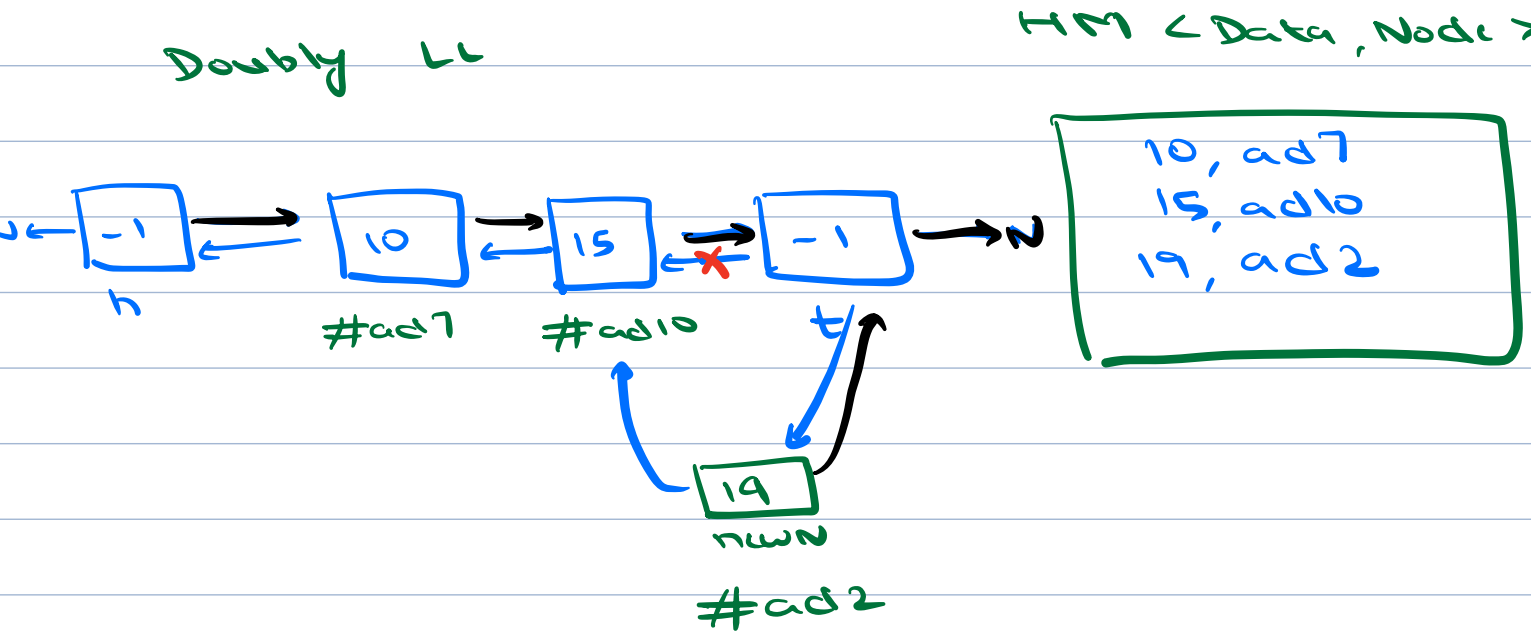
HM <Data, Node>



```
void insertAtTail(Node newN) {
    newN.next = tail
    newN.prev = tail.prev
    tail.prev = newN
    newN.prev.next = newN
}
```

Doubly LL

HM <Data, Node>



Eg 10 15 19 20 18 23 20 19 17 17 10

M = 5

h ⇄ 10 ⇄ 15 ⇄ 19 ⇄ 20 ⇄ 18 ⇄ 23 ⇄ t

If size == cap

delete(h.next)

HM <Data, Node>

~~10, ad7~~
15, ad10
19, ad2
20, ad4
18, ad20
23, ad30

```
void delete (Node cur) {
    cur.prev.next = cur.next
    cur.next.prev = cur.prev
}
```

Eg 10 15 19 20 18 23 20 19 17 17 10

M = 5

h ⇄ 15 ⇄ 19 ⇄ 18 ⇄ 23 ⇄ 20 ⇄ t

```
Node cur = HM[data]
delete (cur)
insertAtTail (cur)
```

HM <Data, Node>

~~10, ad7~~
15, ad10
19, ad2
20, ad4
18, ad20
23, ad30

Eg 10 15 19 20 18 23 20 19 17 17 10

M = 5

Node cur = HM[data]

delete (cur)

insertAtTail (cur)

HM <Data, Node>

~~10, ad7~~

15, ad10

19, ad2

20, ad4

18, ad20

23, ad30

---

hashmap <int, Node> hm

Node head = new Node (-1)

Node tail = new Node (-1)

head.next = tail

tail.prev = head

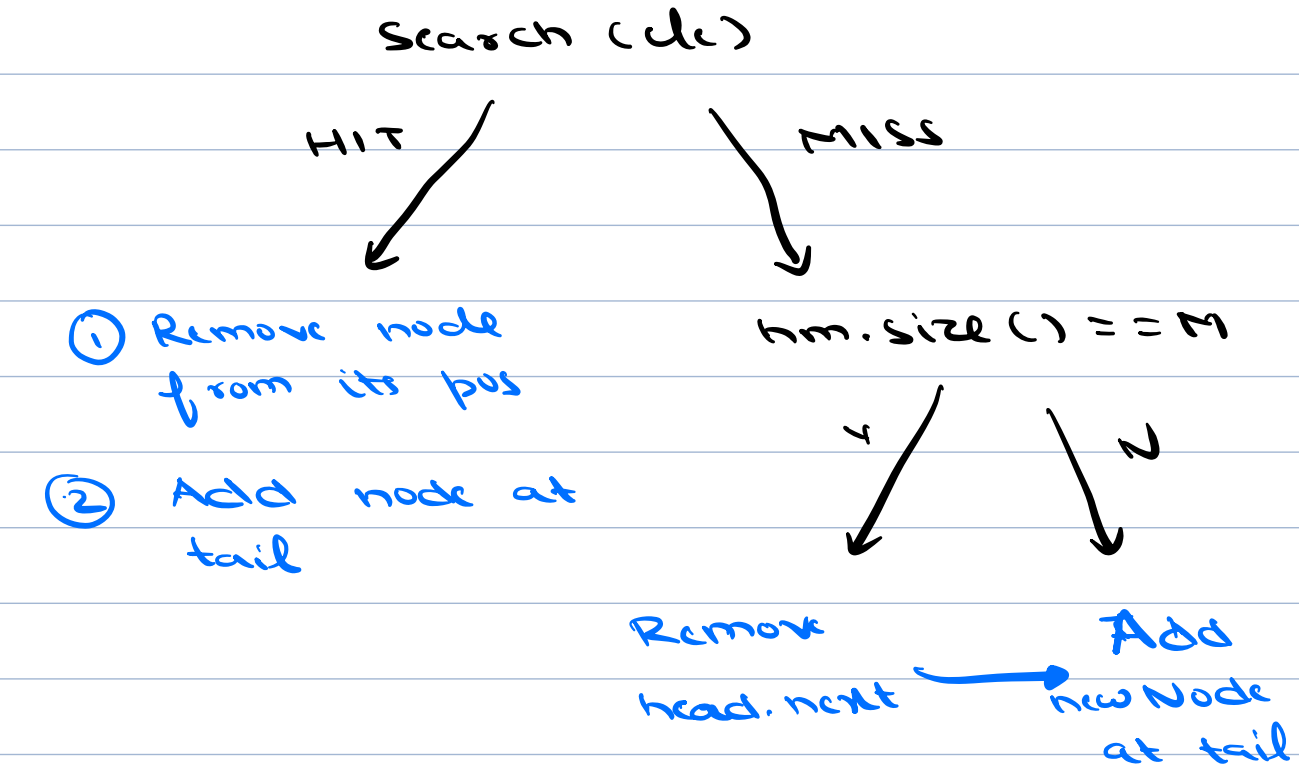
h → 1 → 2 → 3 → t

```
for (i=0 ; i < x ; i++) {
    int dc = x[i]
    if (hm.contains(dc) == true) {
        Node cur = hm[dc]
        delete (cur)
        insertAtTail (cur)
```

```
        }
    else {
        if (hm.size() == M) {
            delete (head.next)
            hm.delete (head.next.data)
        }
        Node nn = new Node (dc)
        insertAtTail (nn)
        hm.insert (dc, nn)
    }
}
```

```
void delete (Node cur) {
    cur.prev.next = cur.next
    cur.next.prev = cur.prev
}
```

```
void insertAtTail (Node newN) {
    newN.next = tail
    newN.prev = tail.prev
    tail.prev = newN
    newN.prev.next = newN
}
```



Cache size → M, No. of dc → N

TC to insert / query : dc → O(1)
↓
O(N)

SC : O(M)
↓
LL + Hashmap

Contest → 2 Pointers, LL, Stacks, Queues