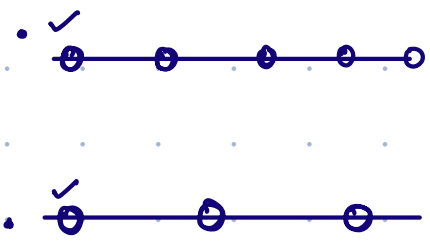
SCALER

LinkedList - 3

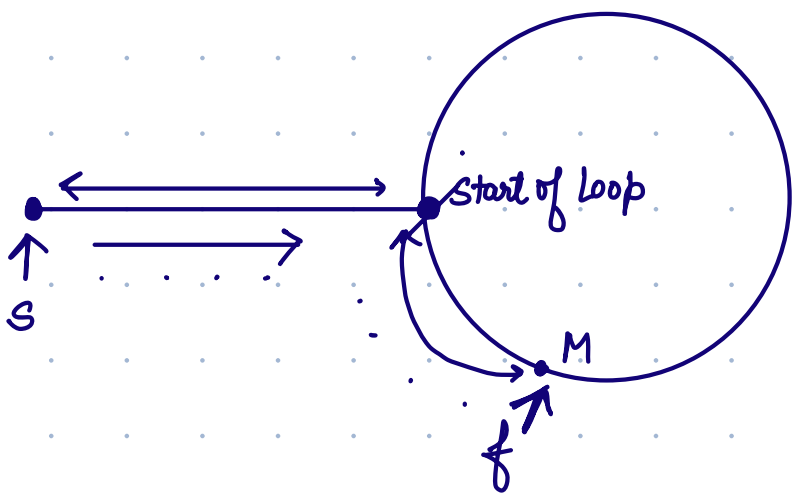
TABLE OF CONTENTS

Q.1. Merge 2 sorted lists



$= O(N+M)$

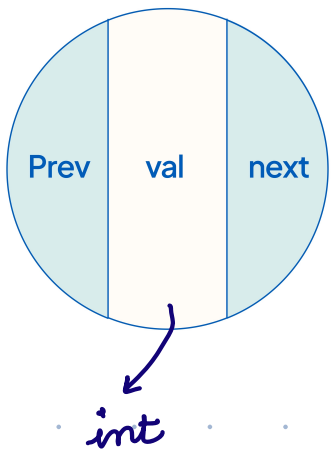
Q2



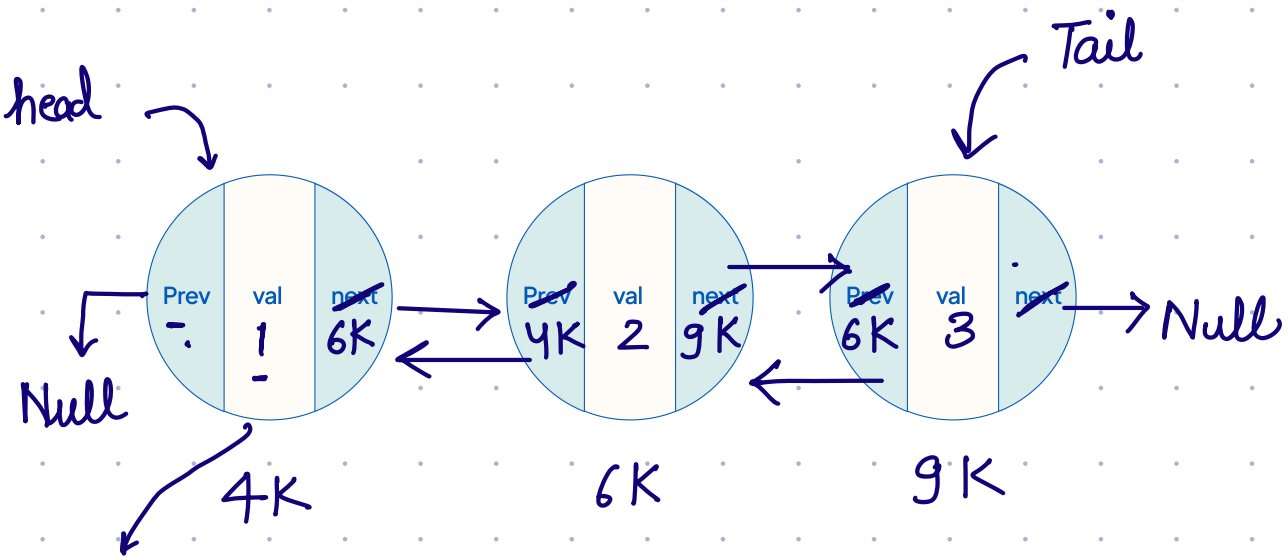
Floyd's Cycle detection technique

Doubly LinkedList

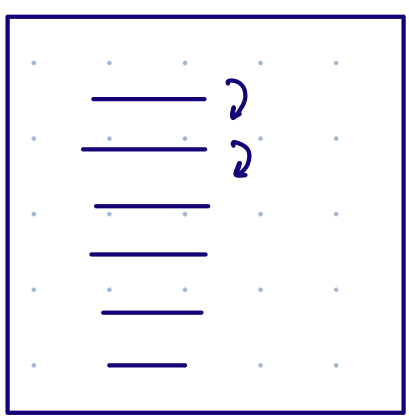
```
class Node {
    int val;
    Node next;
    Node prev;
    public Node (int v) {
        this.val = v;
        this.next = null;
        this.prev = null;
    }
}
```



1 2 3



val → Song ID



Shuffle

Can you make sure non-repetition of songs

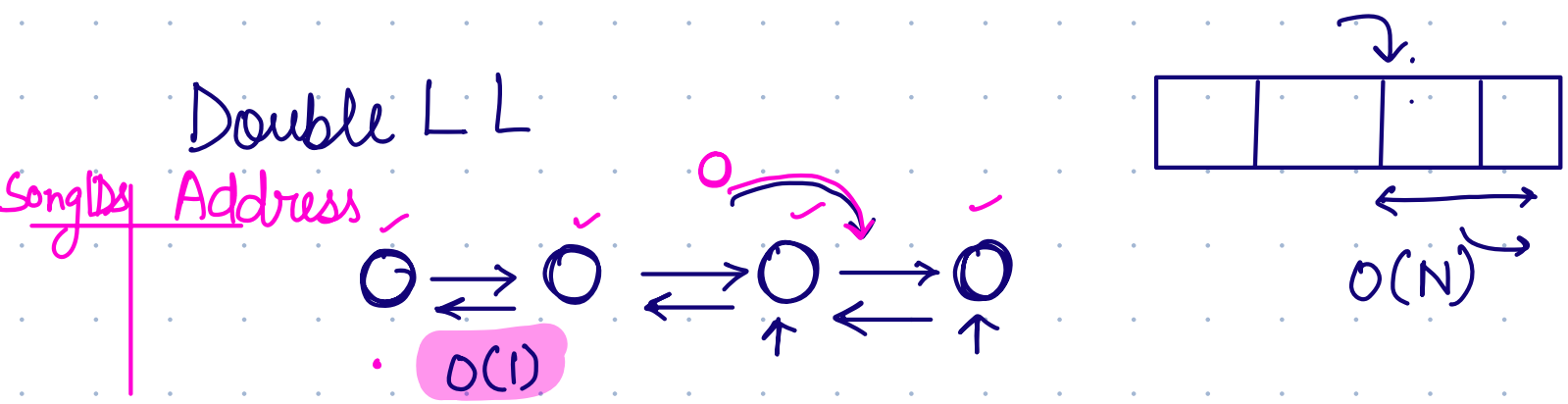
Scenario

Spotify wants to enhance its user experience by allowing users to navigate through their music playlist seamlessly using "next" and "previous" song functionalities.

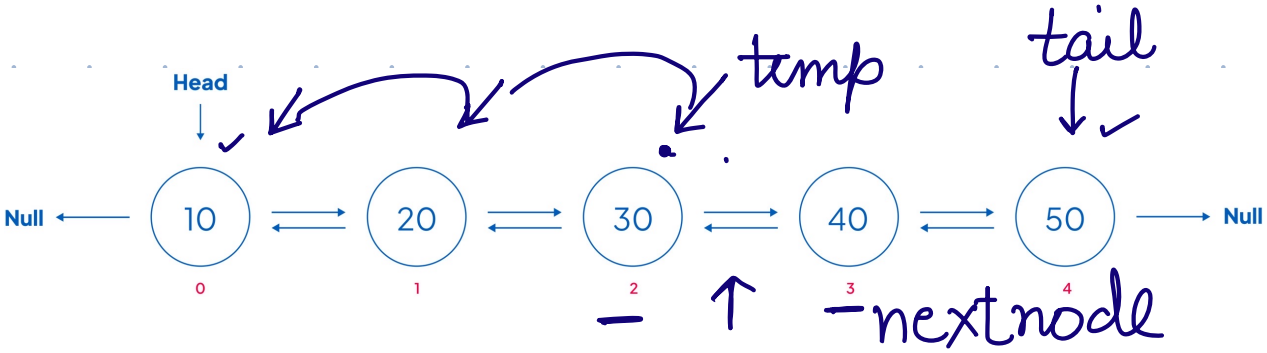
Problem

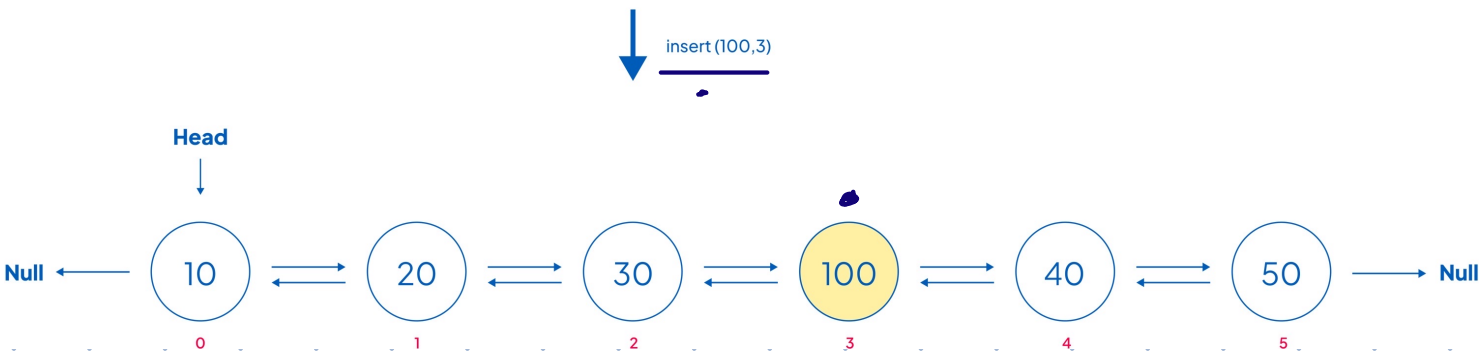
You are tasked to implement this feature using a doubly linked list where each node represents a song in the playlist. The system should support the following operations:

- **Add Song :** Insert a new song into the playlist. If the playlist is currently empty, this song becomes the "Current song".
- **Play Next Song :** Move to the next song in the playlist and display its details.
- **Play Previous Song :** Move to the previous song in the playlist and display its details.
- **Current Song :** Display the details of the current song being played.

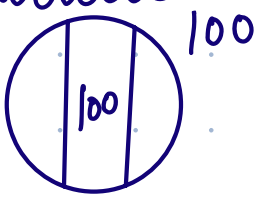


Insertion in Doubly LinkedList





Adding in middle 3



newnode

2 jumps

i i-1

O(N)

temp.next = newnode

newnode.prev = temp

[temp ✓

nextnode = temp.next]

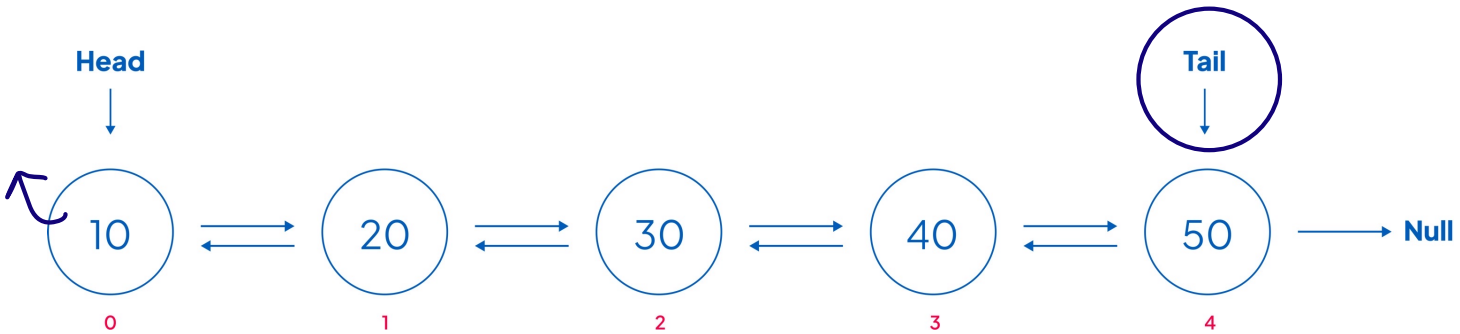
newnode.next = nextnode

nextnode.prev = newnode

4 ↗

Edge - Case

- **At index 0**

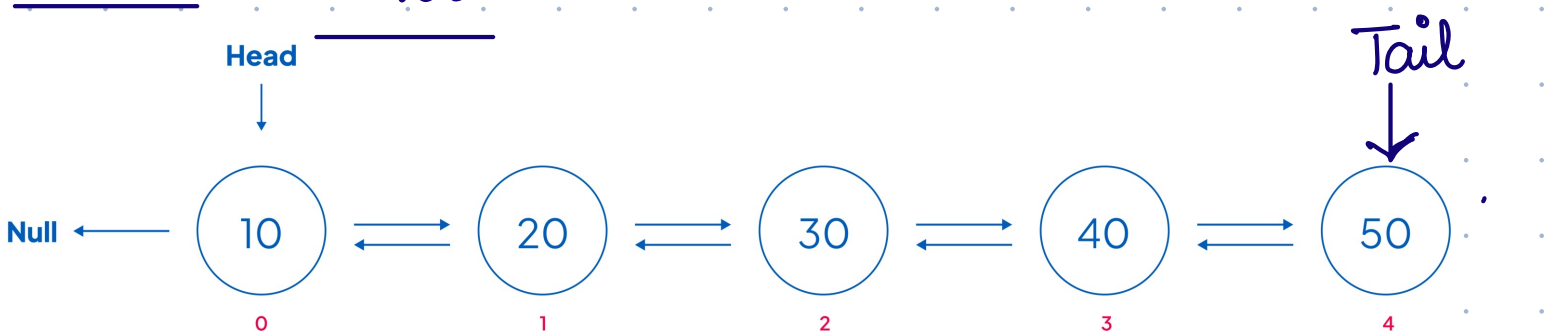


TC: O(1)

```
newnode
newnode.next = head
newnode.prev = null
head.prev    = newnode
head         = newnode
```

i == 0

- **At index 5** - end



len = 5

```
newnode
newnode.prev = tail
newnode.next = null
```

- tail.next = newnode
- tail = newnode

TC : O(1)

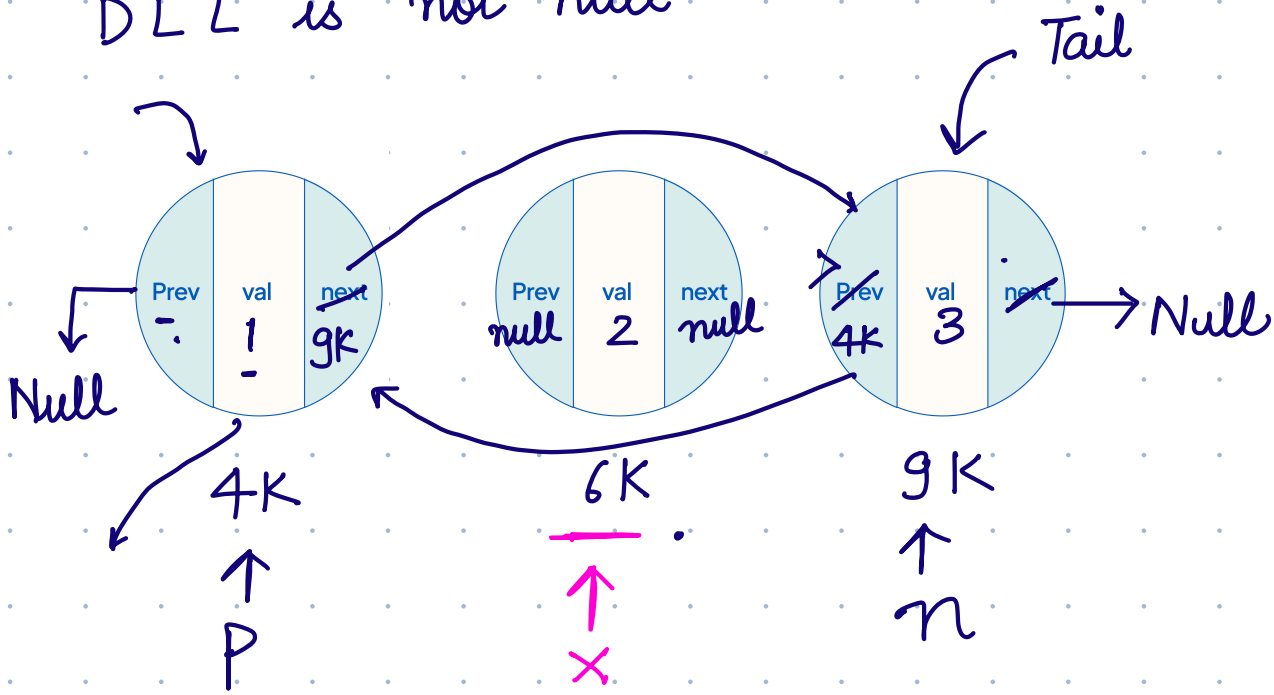
adding at head

adding in middle

adding at last

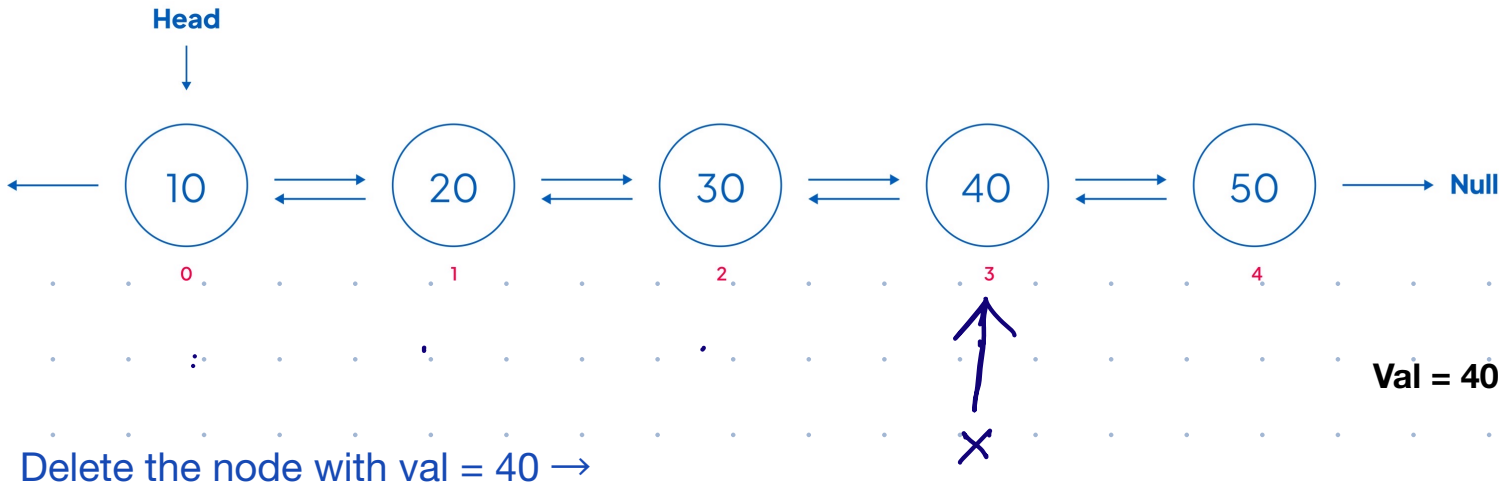
Delete a node in DLL

1) Node reference is given
2) Given node will not be head/tail
3) DLL is not null




```
function remove ( Node x) {
    p = x.prev
    n = x.next
    p.next = n
    n.prev = p
    x.next = null
    x.prev = null
    free (x)
```

Delete a node from D.L.L



Delete the node with val = 40 →

↳ val = uniquely identify

↳ node = unique val

Deleting in middle

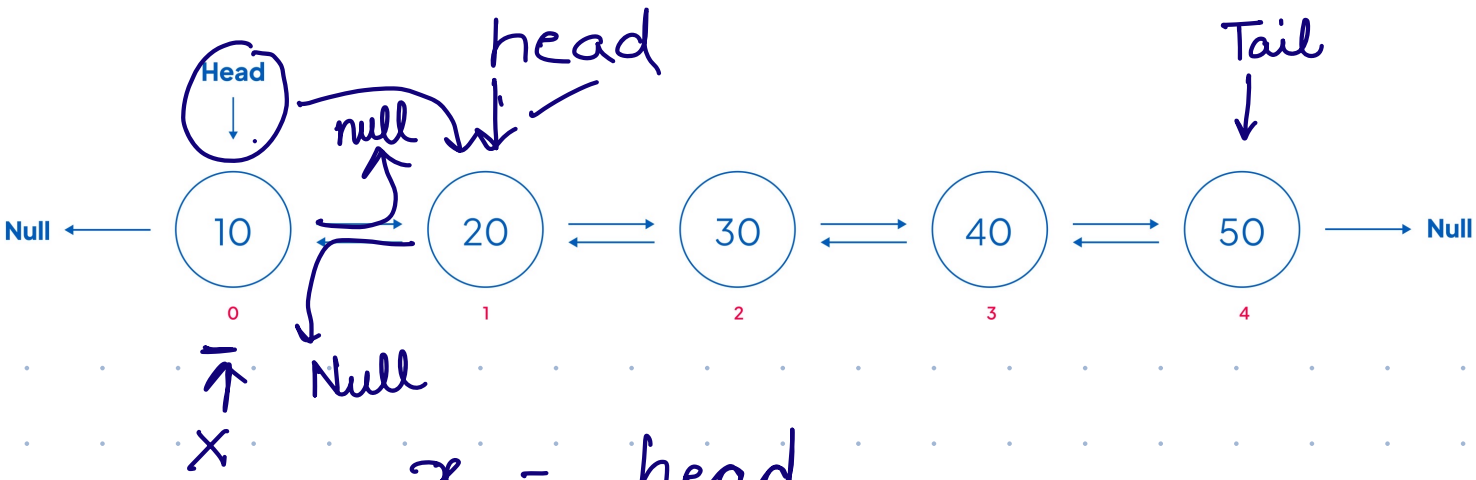
Traverse and find the address of node that you want to delete = Node X

→

```
function remove ( Node x) {
    p = x.prev
    n = x.next
    p.next = n
    n.prev = p
    x.next = null
    x.prev = null
    free (x)
```

Edge - Case

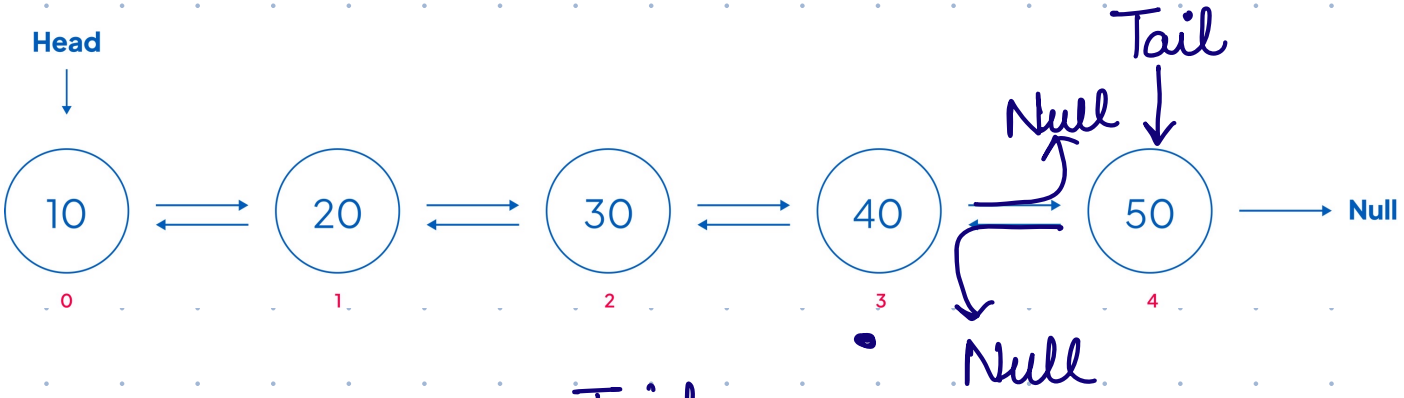
- **idx → 0** **val → 10**




```
x = head
head = head.next
head.prev = null
x.next = null
free(x)
```

- last - node

val → 50



$x = Tail$

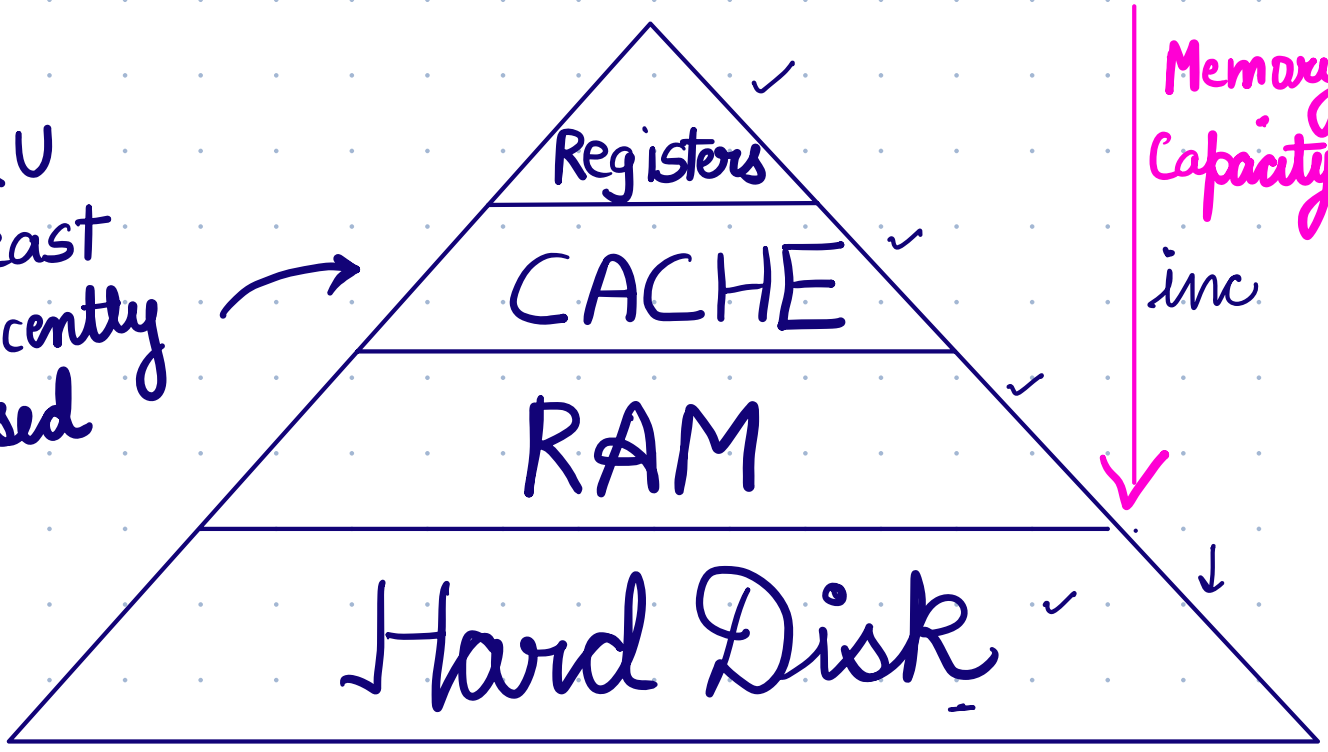
$Tail = Tail.prev$

$Tail.next = null$

$x.prev = null$

$free(x)$

Memory Hierarchy



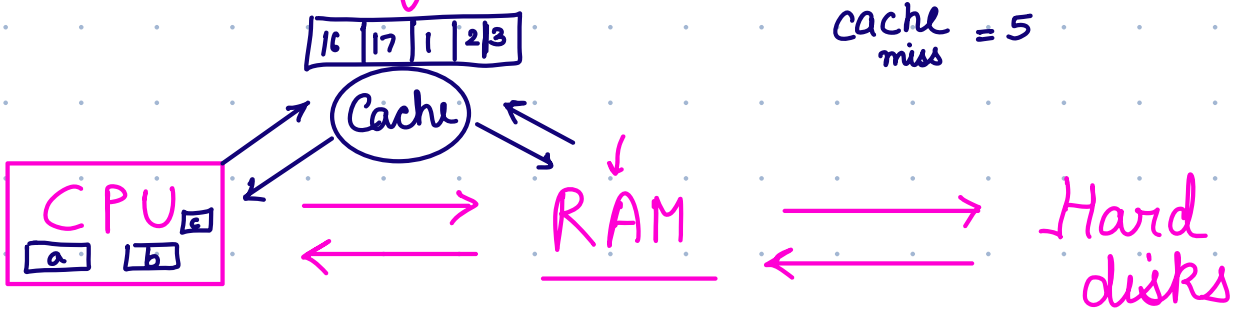
LRU
Least
Recently
used

Memory Capacity inc

Speed of memory increases

LRU cache — most recently used is more likely to be accessed again.

Least Recently used



cache miss = 5

Cache HIT

Cache MISS (it will get data from RAM)

Read write from RAM

(as less as Cache MISSES as possible)

L.R.U Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.

set(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

LRU

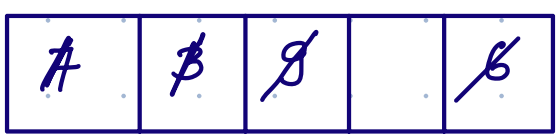
The ~~LU~~Cache will be initialised with an integer corresponding to its capacity. Capacity indicates the maximum number of unique keys it can hold at a time.

Definition of "least recently used" : An access to an item is defined as a get or a set operation of the item. "Least recently used" item is the one with the oldest access time.

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

Capacity = 5

Size = 3



2 10 15 8 14.

Searching

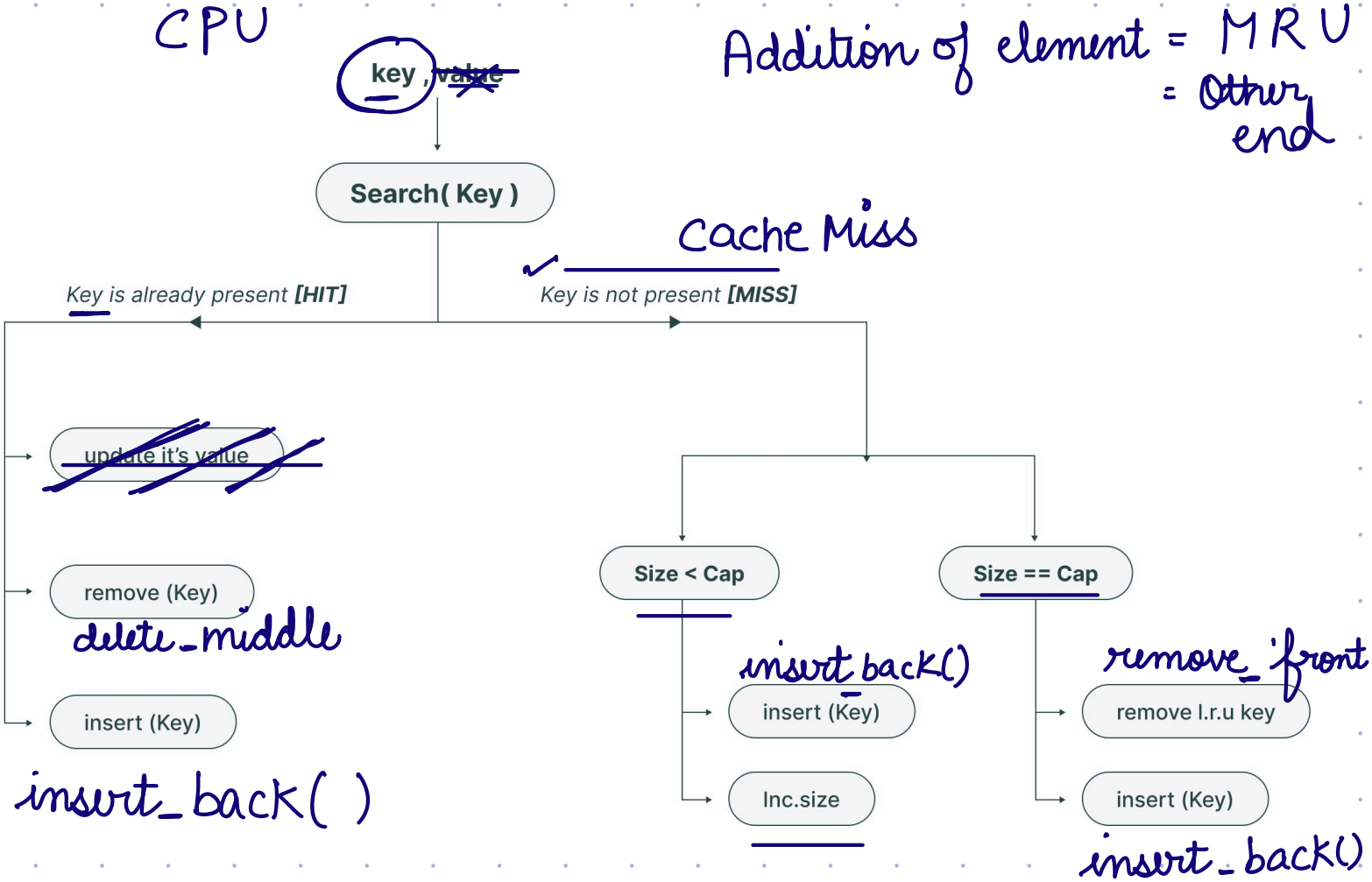
↳ HashMap

Size = ~~0 1 2 3 4~~ 5

LRU = 7 3 9 6

Removing an element = LRU
= One End

Addition of element = MRU
= Other end



insert

search → 0(1)

remove

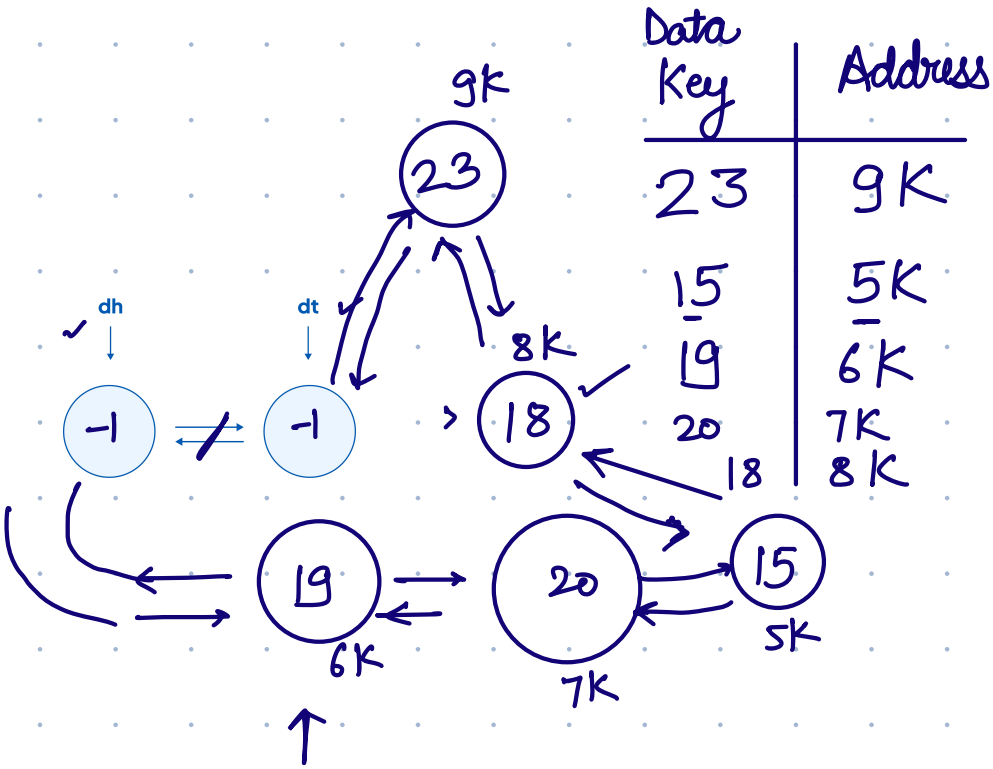
*dry -run

[10 15 19 20 15 18 23 20]

Capacity = 5

size = ~~0 1 2 3 4~~ 5

DLL



---

data (n)

search (n)

present in Cache

delete (n)

insert_back (n)

size ++

miss in cache

size == cache?

Yes

delete_front()

insert_back (n)

No

insert_back (n)

size ++

Code:

```
class Node {
    int data;
    Node prev;
    Node next;

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

```
Node head = new Node(-1)

Node tail = new Node(-1)
head.next = tail
tail.prev = head

cap = 5
```

```
HashMap<Integer, Node> hm
```

```
function LRU (int X) {
    if (hm.containsKey(X)==true) {
        Node t = hm.get(X)
        DeleteNode(t)
        Node nn = new Node(X)
        insert_back(nn, tail)

        hm.put(X, nn)
    } else {
        if (hm.size() == cap) {
            LRU [ Node t = head.next
                  hm.delete(t.data);
                  DeleteNode(t);
                  Node nn = new Node(X)
                  insert_back(nn, tail)

        }
        hm.put(X, nn)
    }
    else {
        Node nn = new Node(X)
```

insert_back (nn, tail)

hm. put (x, nn)

}

}

}

Delete Node (Node t)

insert_back (nn , tail)

Check Palindrome

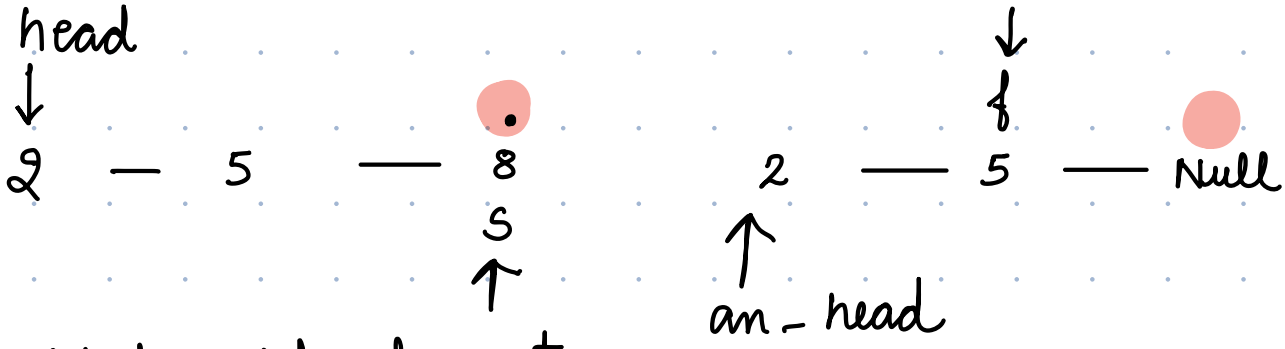
2 → 5 → 8 → 7 → 3

No

2 → 5 → 8 → 5 → 2

Yes!

- 1 → Null



O(N) 1) find mid element

2)

O(1) head / an_head

O(N) 3) Reverse (an_head)

O(N) 4) Compare head an_head one by one.

TC: O(N)