SCALER

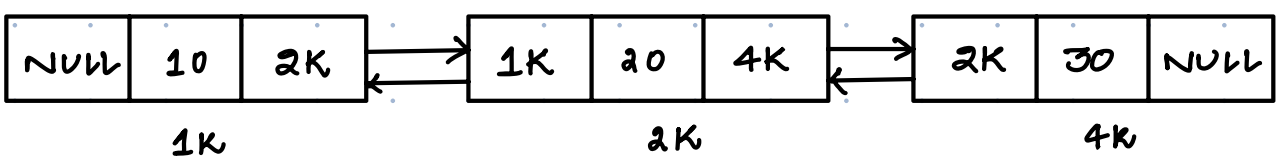
LinkedList - 3

Agenda

Insert a Node in DLL
Delete the first occurance of an element in DLL
LRU Cache
Cycle Detection in LinkedList
Start of the Cycle

Doubly LinkedList


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



QUIZ prev Pointer of Head of Doubly Linked List points to:

NULL

Scenario

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

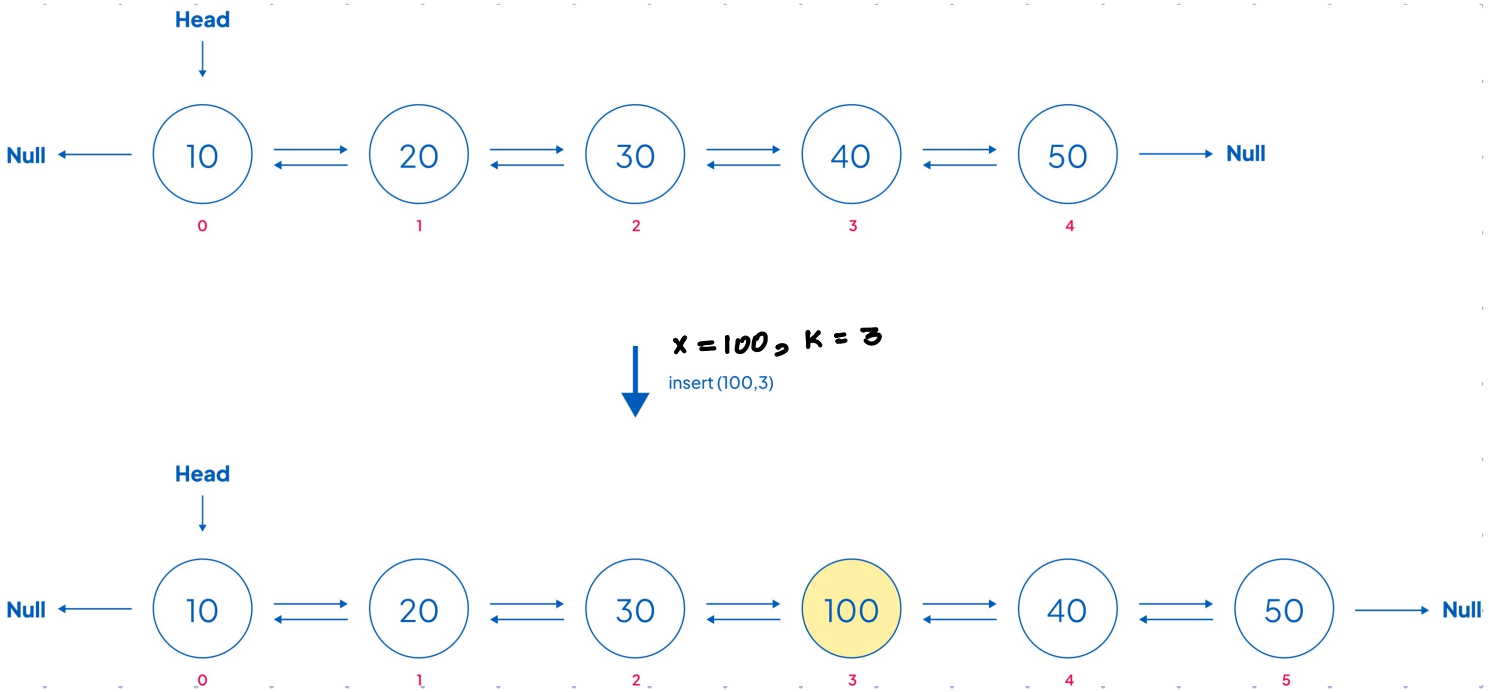
Problem

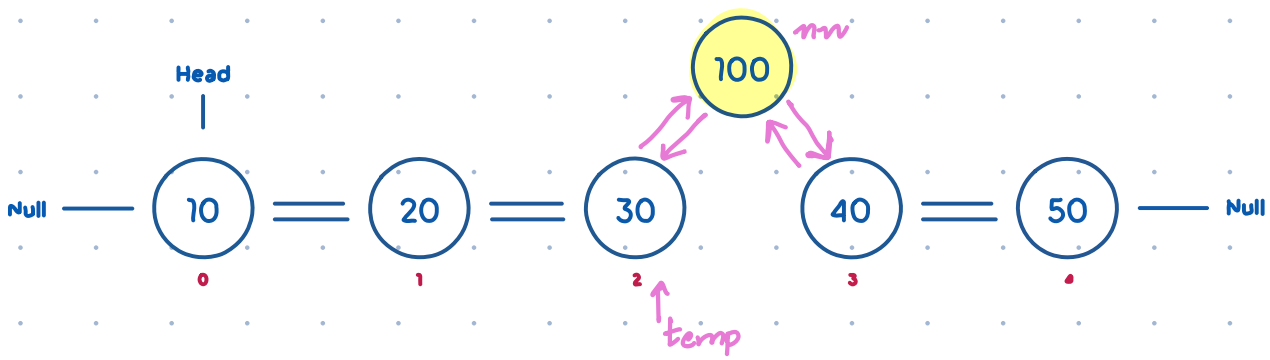
You are tasked to implement this feature using a doubly linked list where each node represents a song in the playlist. The system should support the following operations:

- **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

Insert a node with data “x” at index “k”.





- Create New Node
- Iterate till $(K-1)^{th}$ index
- Form new connection

```
nn.next = temp.next
temp.next.prev = nn
```

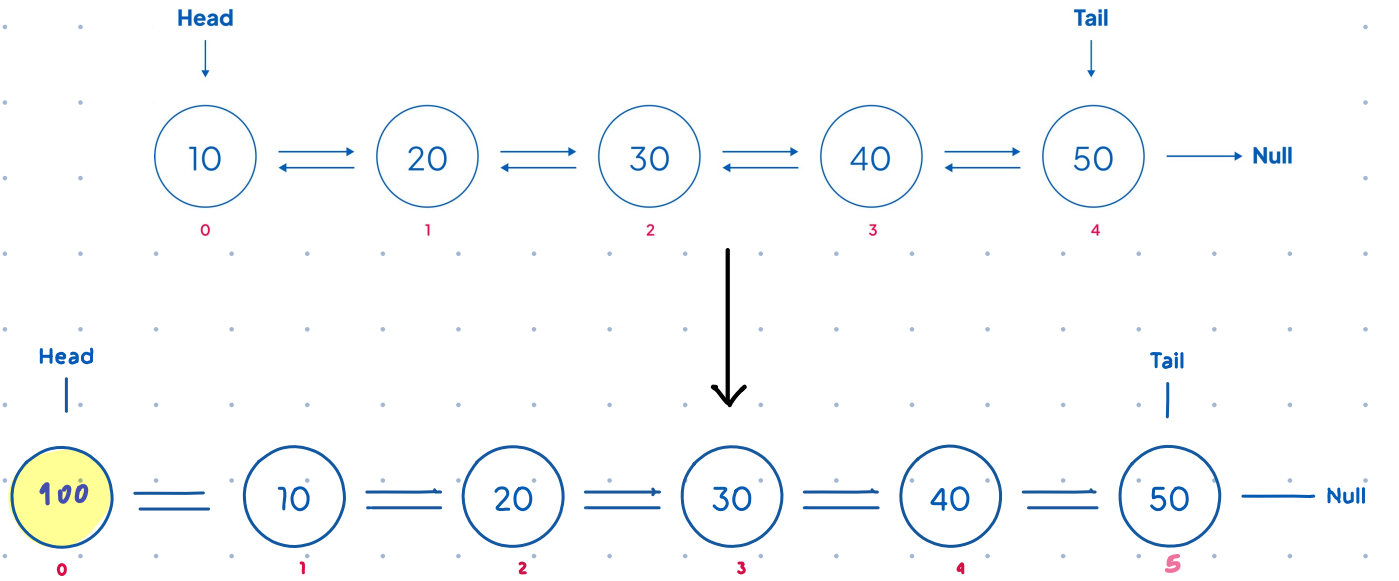
(temp.next = 40)

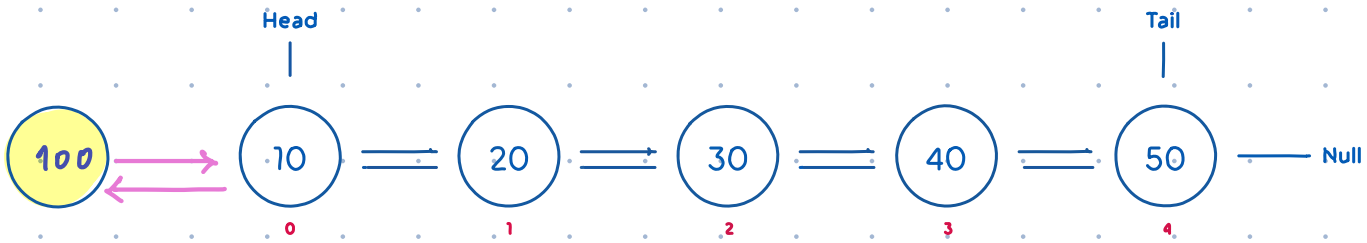
- Update prev connection

```
temp.next = nn;
nn.prev = temp;
```

Edge - Case

Case 1 : K = 0 , X = 100 (Start of LL)



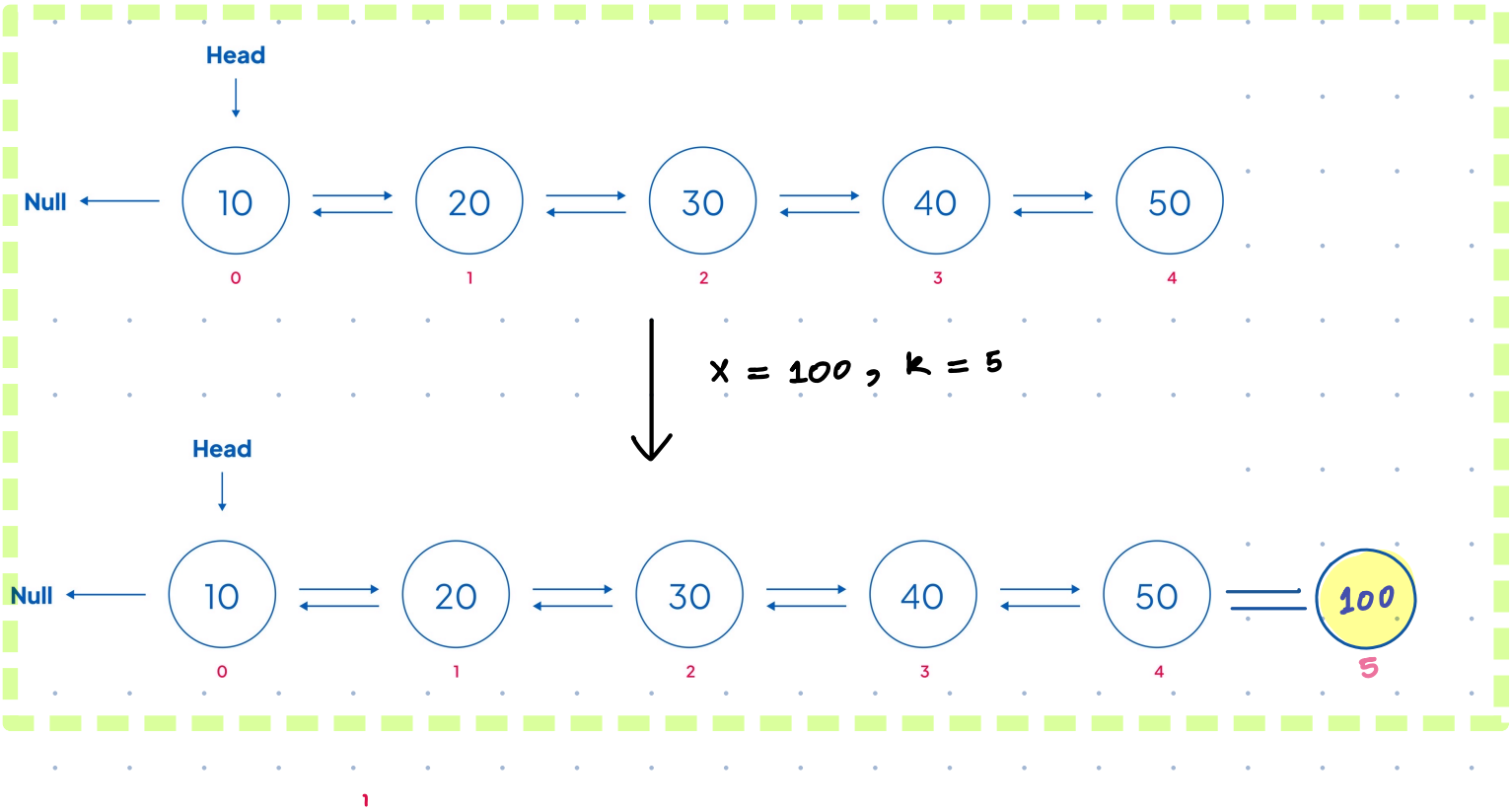


- Create new node
- nn.next = head

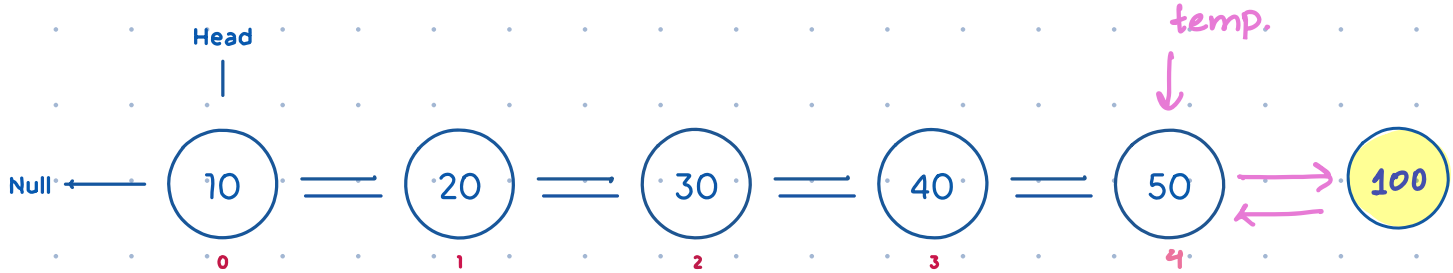
  head.prev = nn
- Update head

  head = nn

Case 2 : K = 5 , x = 100 (end of LL)



1



- Create New Node
- Iterate till last node / $(K-1)^{th}$ node
- nn.prev = temp

  temp.next = nn

< / > Code

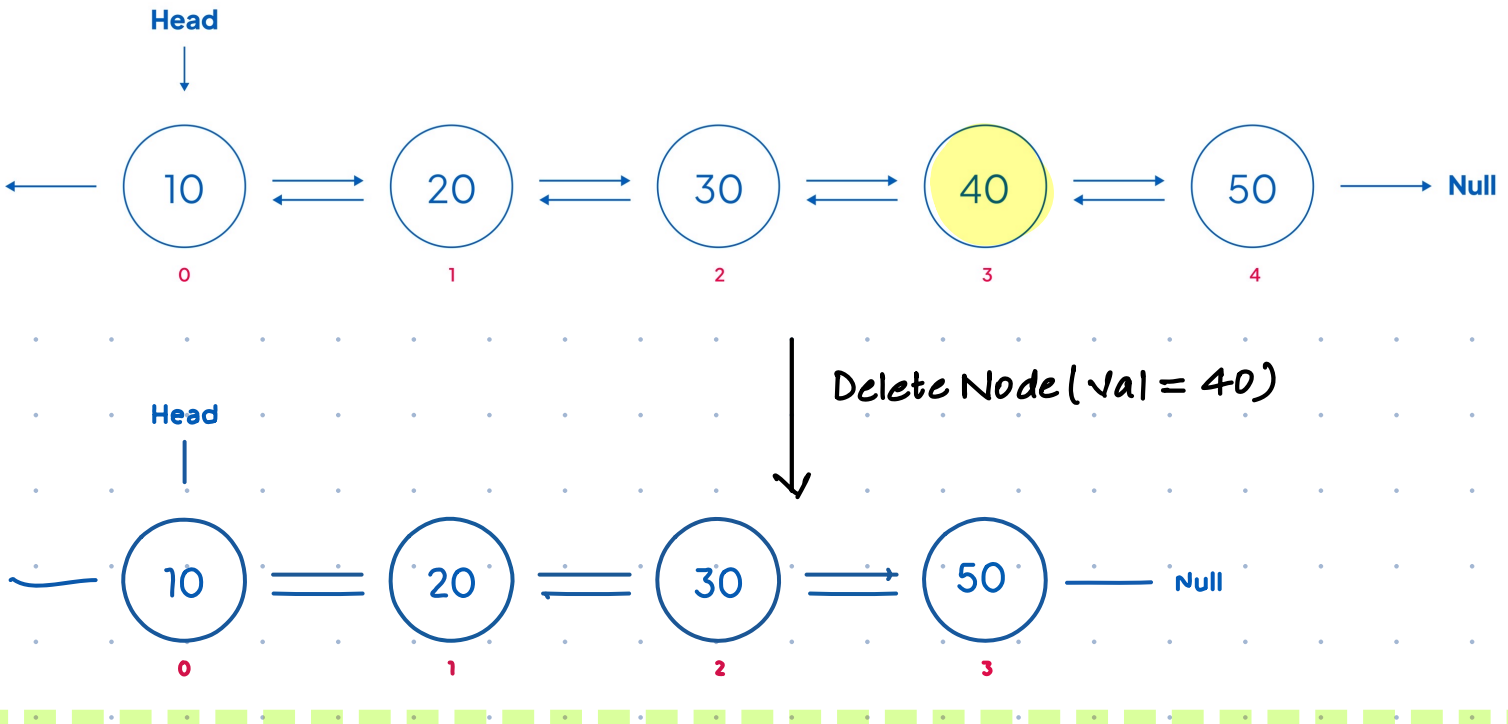
```
Node InsertInDLL (Node head, int K, int x) {

}
```

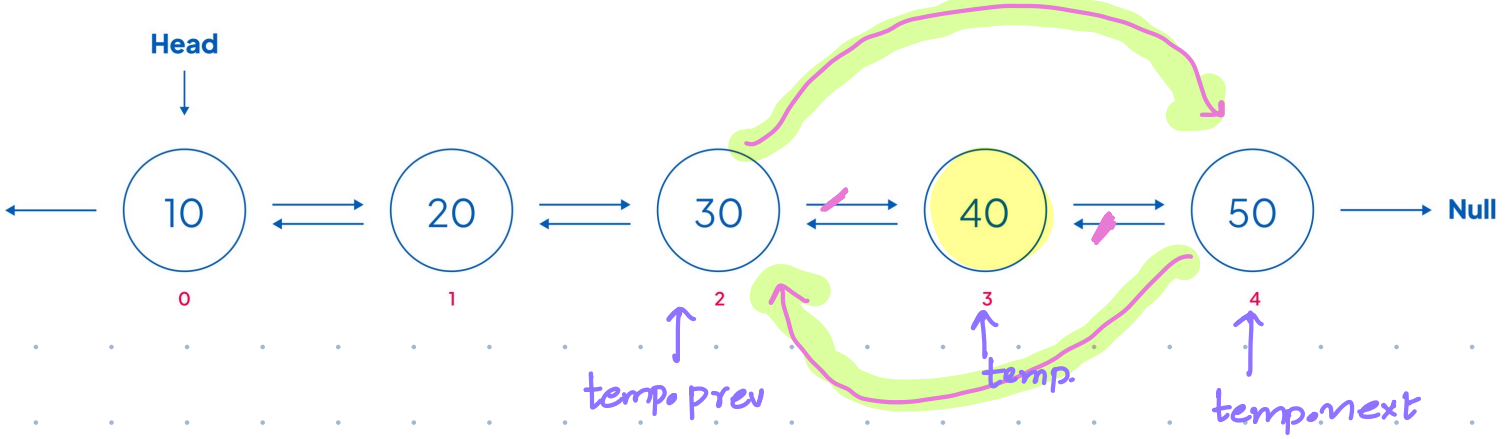
QUIZ

In a doubly linked list, the number of pointers affected for an insertion operation between two nodes will be? 4

Delete a node from D.L.L

Assume All values in DLL are distinct.
Delete the node with val = 40.





First iterate & check if the node with given val is present or not.

temp.prev.next = temp.next ;
(temp.prev = 30, temp.next = 50)

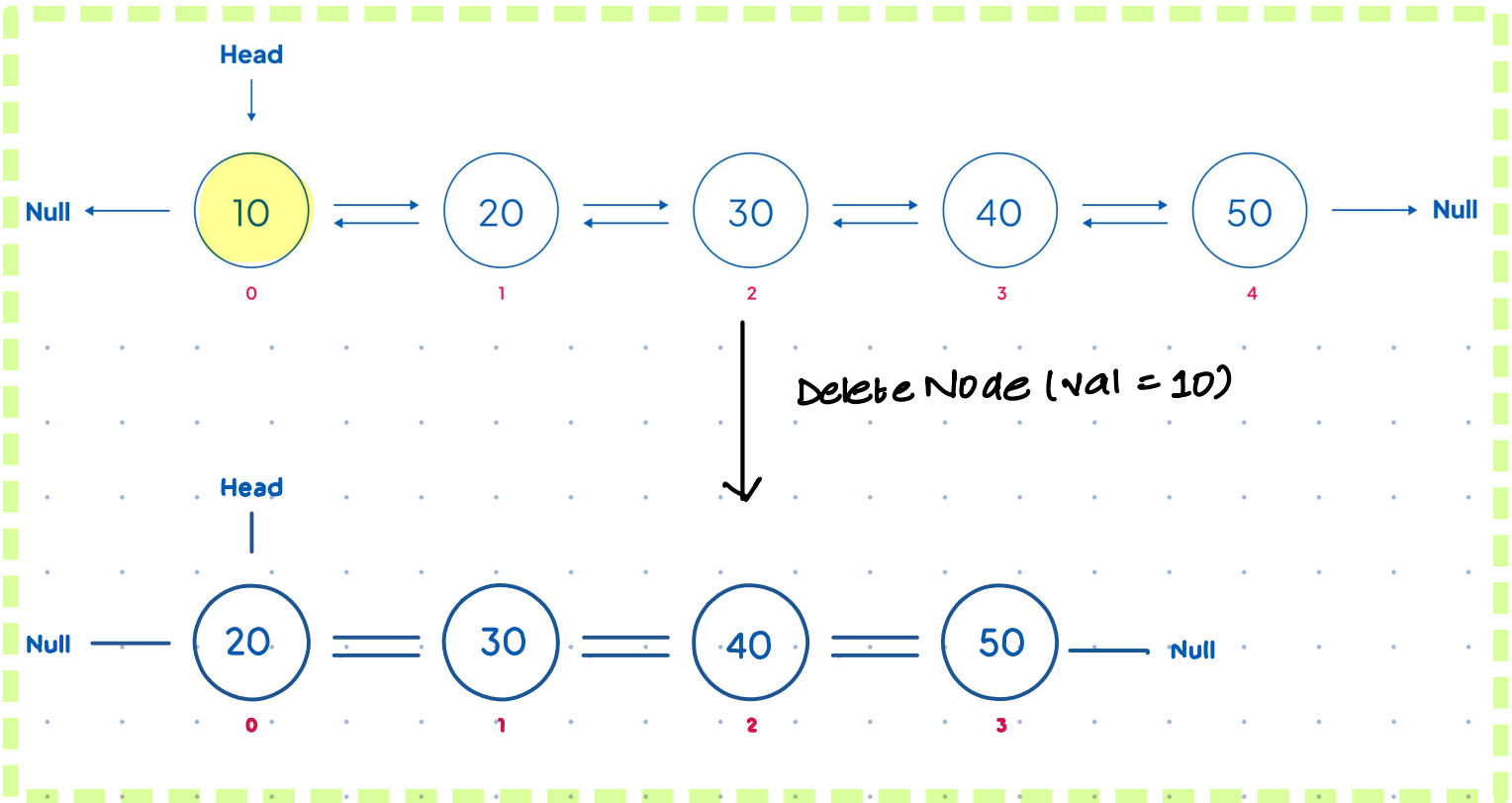
temp.next.prev = temp.prev ;
(temp.next = 50, temp.prev = 30)

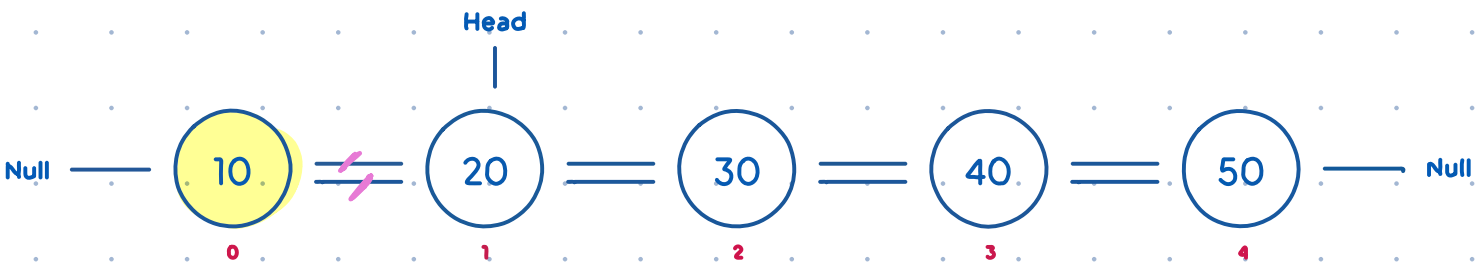
temp.next = NULL

temp.prev = NULL

Edge - Case

Case 1 : Delete Node at Start



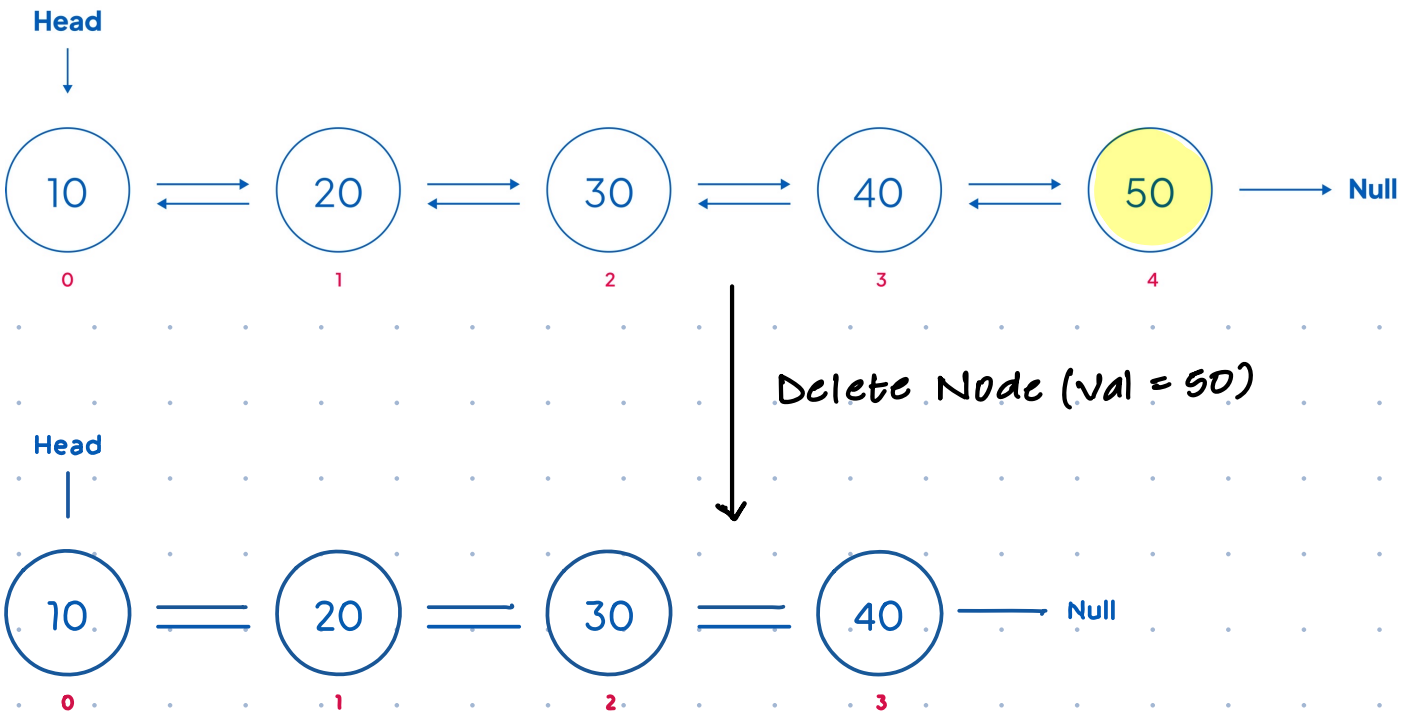


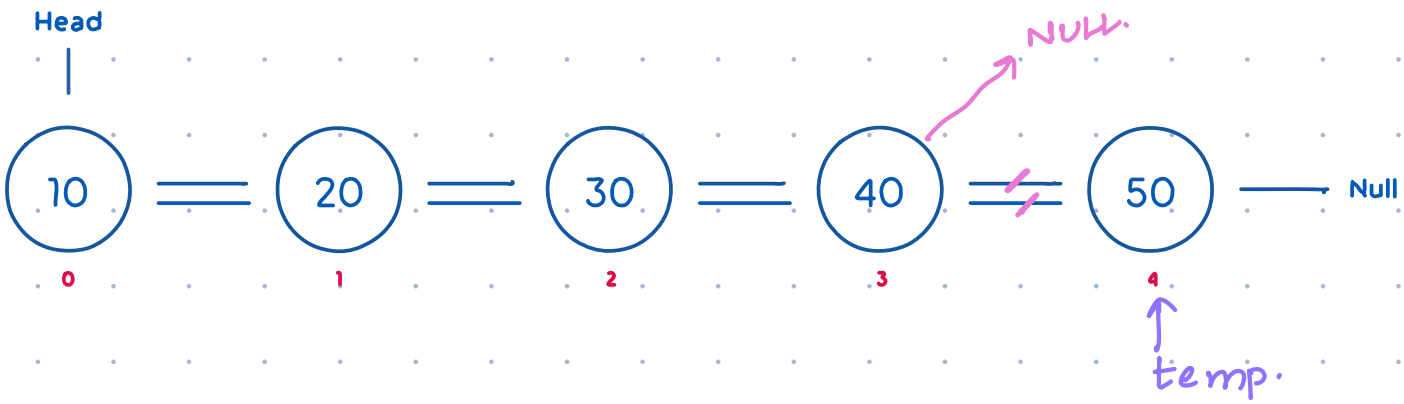
head = head.next

head.prev.next = NULL

head.prev = NULL

Case 2 : Delete Last Node





temp.prev.next = NULL

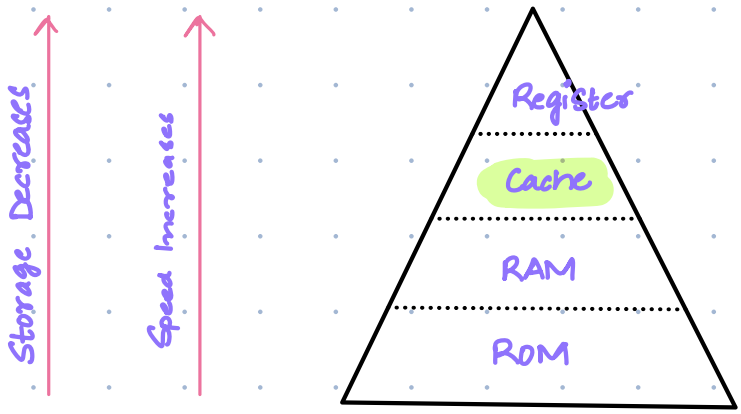
temp.prev = NULL

< / > Code

```
Node deleteNode(Node head, int val) {

    Node temp = head;

    // Search for node to be deleted

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

    if (temp == NULL)        // Target is not present
        return head;

    // Case 1: Only 1 Node in LL. & that is the one to be deleted.

    if (temp.prev == NULL && temp.next == NULL) {
        head = NULL;
        return head;
    }

    // Case 2: Target is at the start

    else if (temp.prev == NULL) {
        head = head.next
        head.prev.next = NULL
        head.prev = NULL
    }

    // Case 3: Target is at the end
    else if (temp.next == NULL) {
        temp.prev.next = NULL;
        temp.prev = NULL;
    }

    else {        // Target is in the middle
        temp.prev.next = temp.next;
        temp.next.prev = temp.prev;
        temp.next = NULL, temp.prev = NULL;
    }

    return head;
}
```

Implement L.R.U Cache

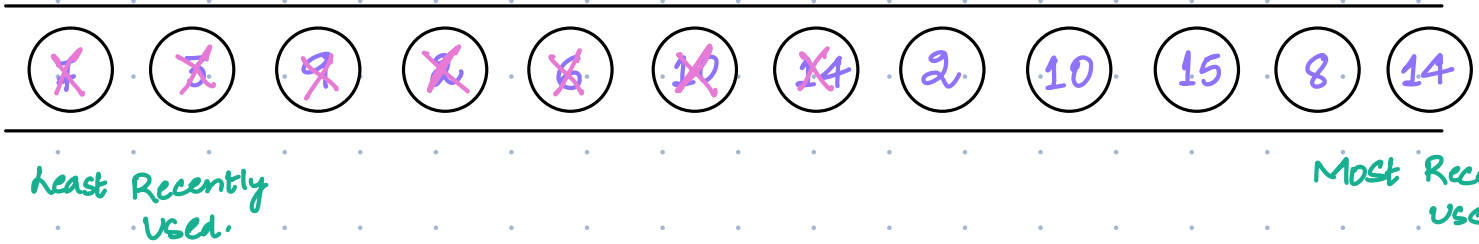
We have been given a running stream of integers and the fixed memory size of M, we have to maintain the most recent M elements. In case the current memory is full, we have to delete the least recent element and insert the current data into the memory (as the most recent item).



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

cache size = 5

size = ∅ 1 2 3 4 5

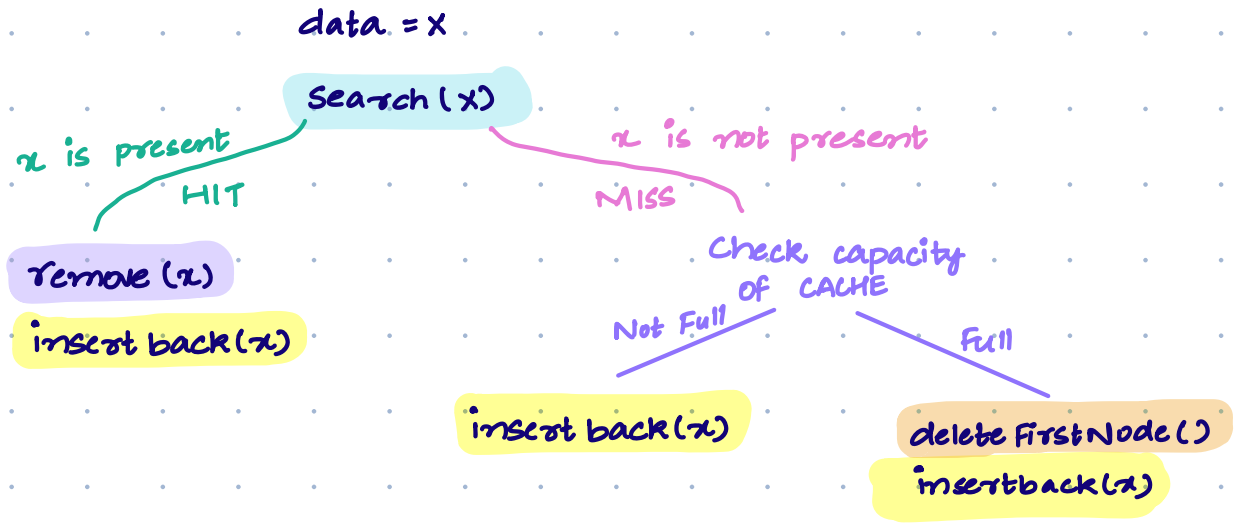


Least Recently Used.

Most Recently Used

QUIZ

Flowchart



Operations Required :

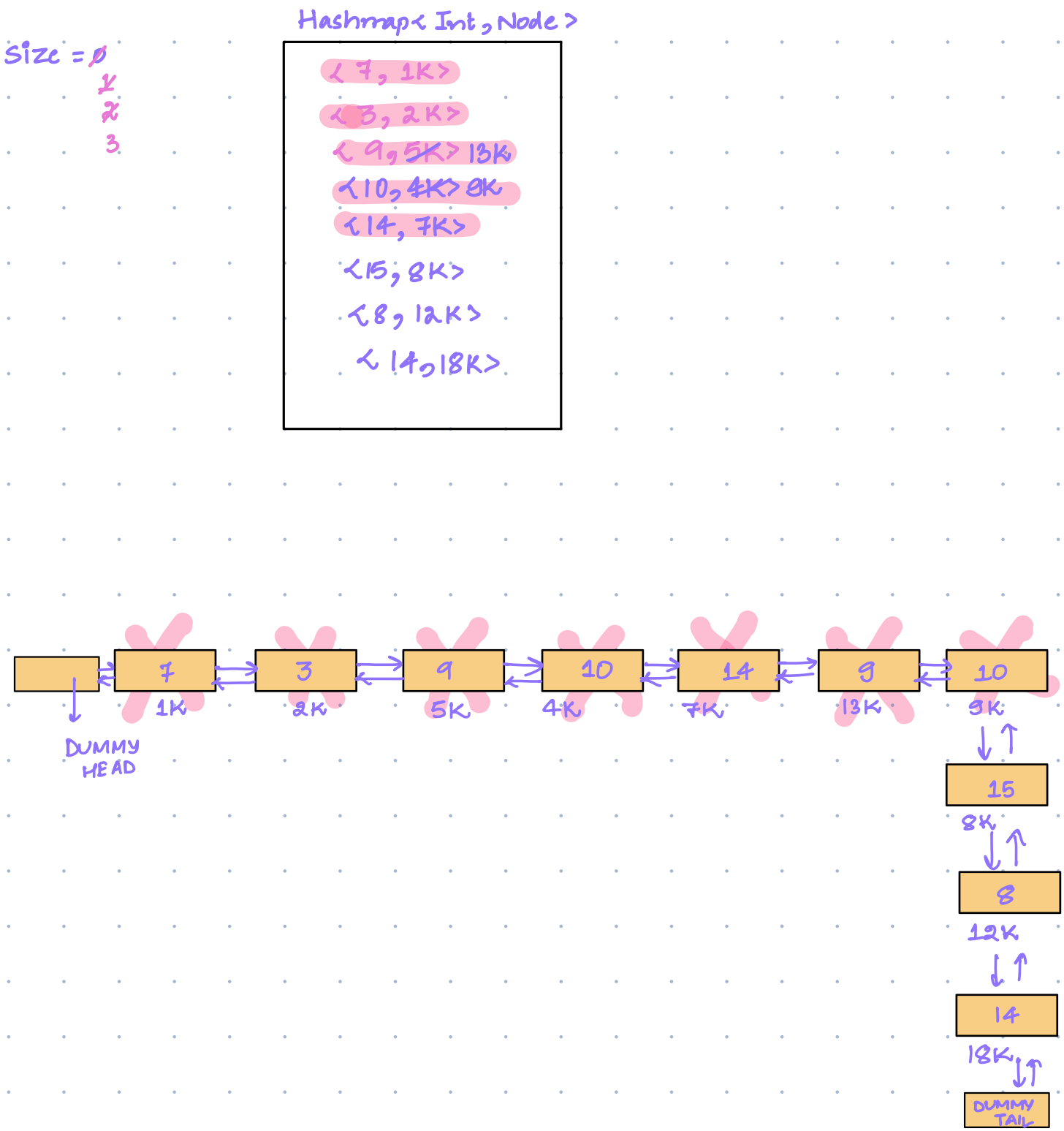
Complexities of analysing LRU through various Data Structures

Operations	ArrayList	Singely LL	Singely LL + Hashmap	DLL + Hashmap
Search (x)	O(N)	O(N)	O(1)	O(1)
Remove (x)	O(N)	O(N)	O(N)	O(1)
Insertback (x)	O(1)	O(1) → If tail is given	O(1)	O(1)
DeletefirstNode()	O(N)	O(1)	O(1)	O(1)

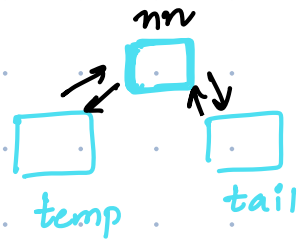
Hashmap< Integer, Node >

Data: 7 3 9 10 14 9 10 15 8 14

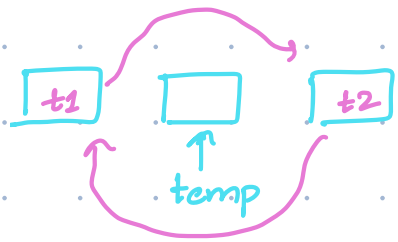
Cache size: 3



```
void InsertBack(Node nn, Node tail) {   // Insert before tail
    Node temp = tail.prev;
    temp.next = nn;
    nn.prev = temp;
    nn.next = tail;
    tail.prev = nn;
}
```



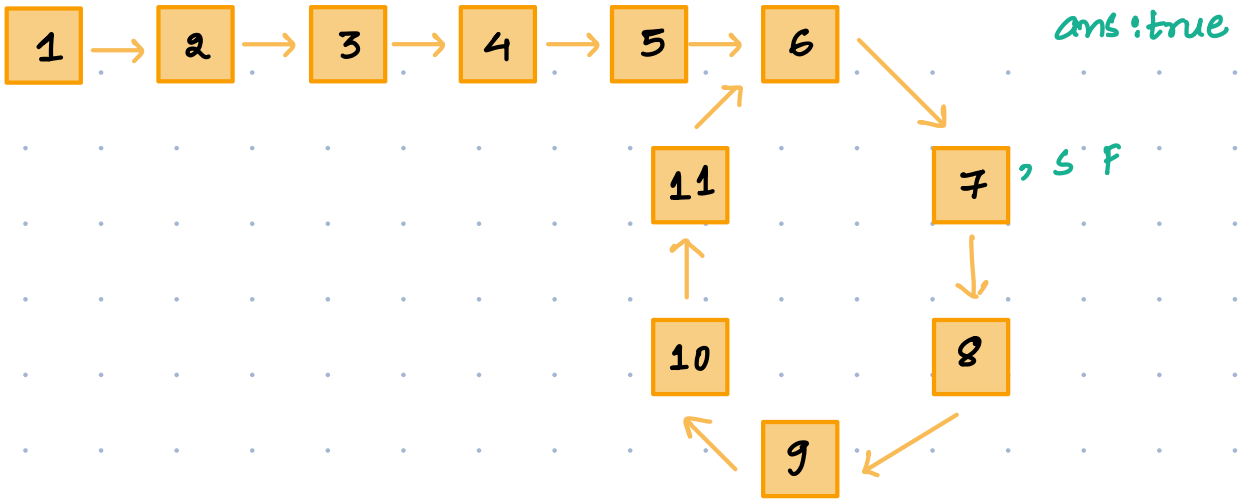

```
void deleteNode(Node temp) {
    Node t1 = temp.prev;
    Node t2 = temp.next;
    t1.next = t2;
    t2.prev = t1;
    temp.next = NULL;
    temp.prev = NULL;
}
```

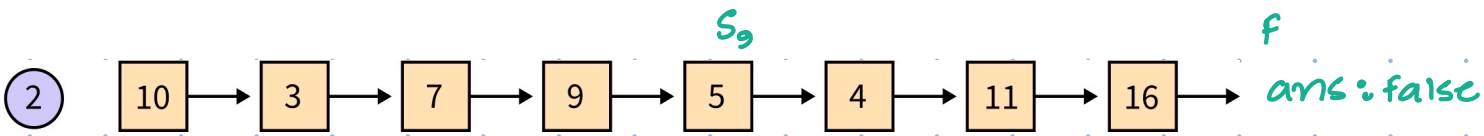


< / > Code

```
Hashmap<Int, Node> hm ;
Node head = new Node(-1);
Node tail = new Node(-1);
    head.next = tail;
    tail.prev = head;

function LRU (int x, int capacity) {
    // If x is already present
    if (hm.containsKey(x) == true) {                    // HIT case.
        // Remove
        Node temp = hm.get(x);
        deleteNode(temp);
        // Insert back
        Node nn = new Node(x);
        InsertBack(nn, tail);
        // Update in hashmap.
        hm.put(x, nn);
    }
    else {  // If x is not already present
        if (hm.size == capacity) {      // Cache is full
            // delete Node from start of LL
            Node temp = head.next;
            deleteNode(temp);
            hm.remove(temp.data);
        }                                                   } MISS
        // Insertion at back
        Node nn = new Node(x);
        InsertBack(nn, tail);
    }     hm.put(x, nn);
}
```

Check if there is a loop





BF Idea

Iterate & store address in a hashset.

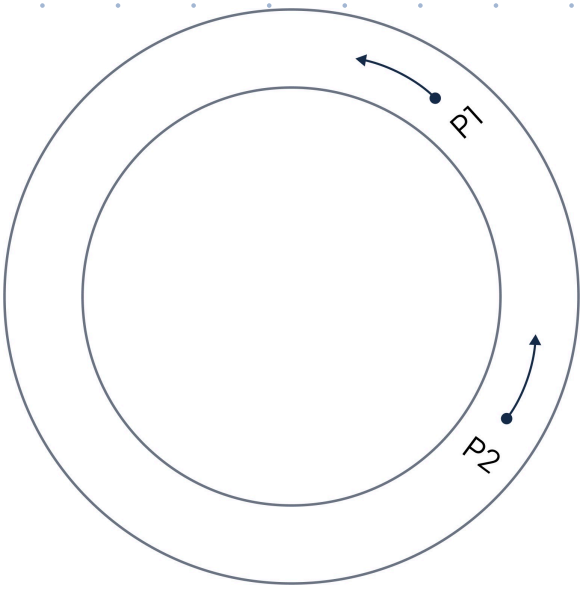
If any address is already present before → CYCLE.

If we reach NULL → NO CYCLE.

T.C : O(N) S.C : O(N)

Do it in const space.

Idea -2 If two people are running with different speeds on a circular track, they will 100% meet at some point.



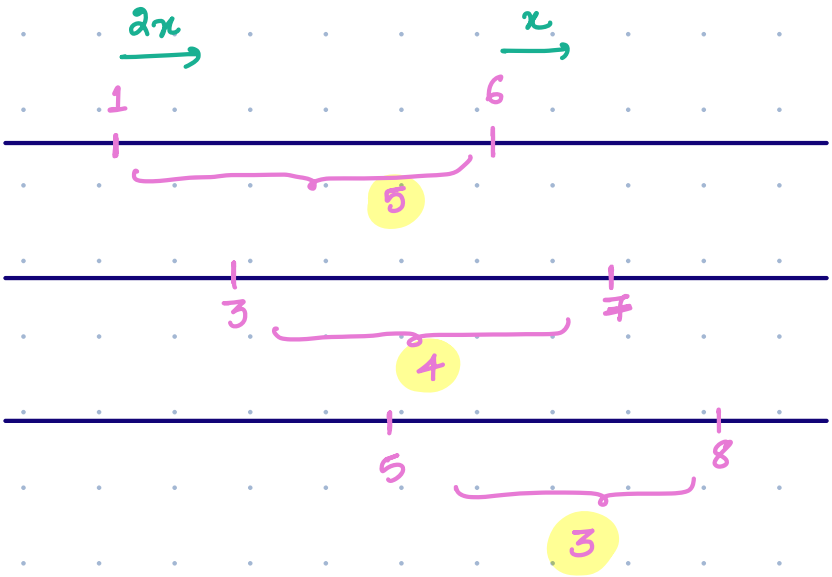
$P_1 = 2P_2$

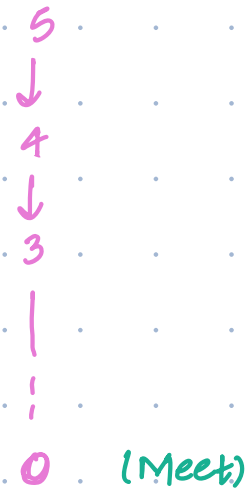
Hare & Tortoise Algo.

< / > Pseudo-Code

```
boolean detectCycle(Node head){
    Node slow = head; Node fast = head;
    while( fast != NULL && fast.next != NULL){
        slow = slow.next;
        fast = fast.next.next;
        if(slow == fast)
            return true;
    }
    return false;
}
```

T.C: O(N)
S.C: O(1)





x $2x$