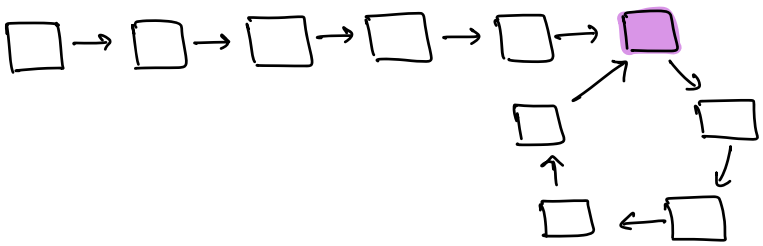
Q:1 Cycle Detection in L·L

- Detect if there's a cycle in L·L
- find the starting node of the cycle.



Approach 1 :-

HashMap / HashSet

⇒ HashSet<Node> set;

Iterate over the L·L :

for every node :

check if it is already present in the set :

[if present : return true;
else : add node to set.

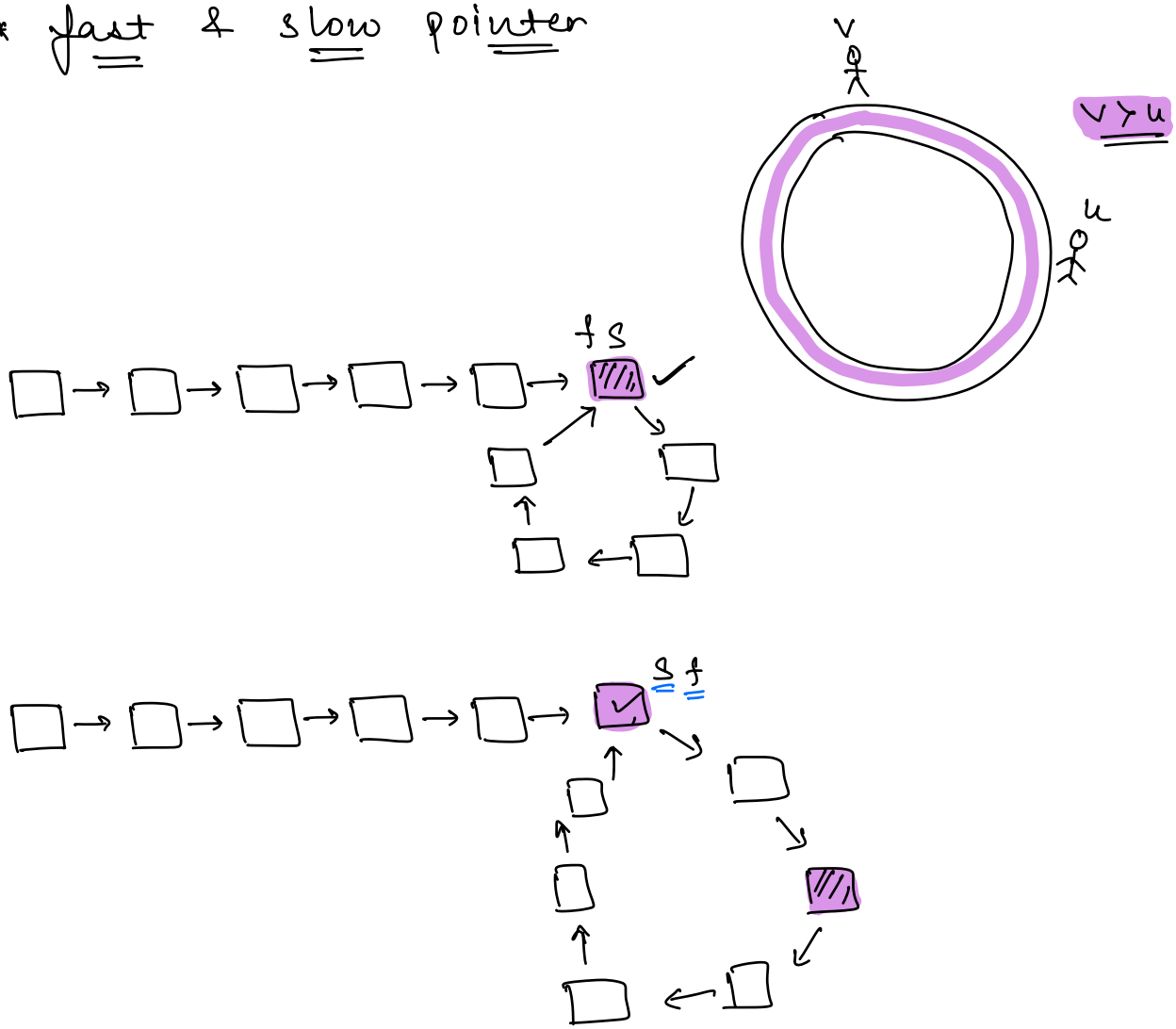
Node which is already present in the set is the starting node of the cycle.

TC : O(N)

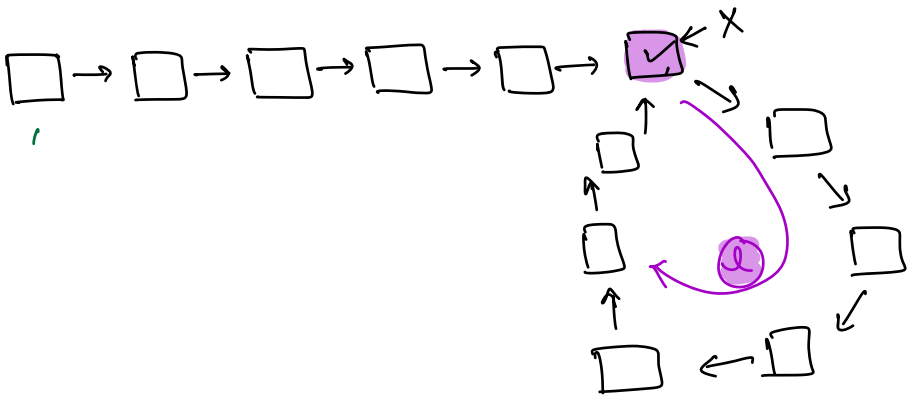
SC : O(N)

Approach 2:-

* fast & slow pointer




```
if (head == Null) return false;
slow = head, fast = head;
while (fast != Null && fast.next != Null){
    ① slow = slow.next
    ② fast = fast.next.next;
    ③ if (slow == fast) return true;
}
```



Length of cycle = l

* Starting node of cycle.

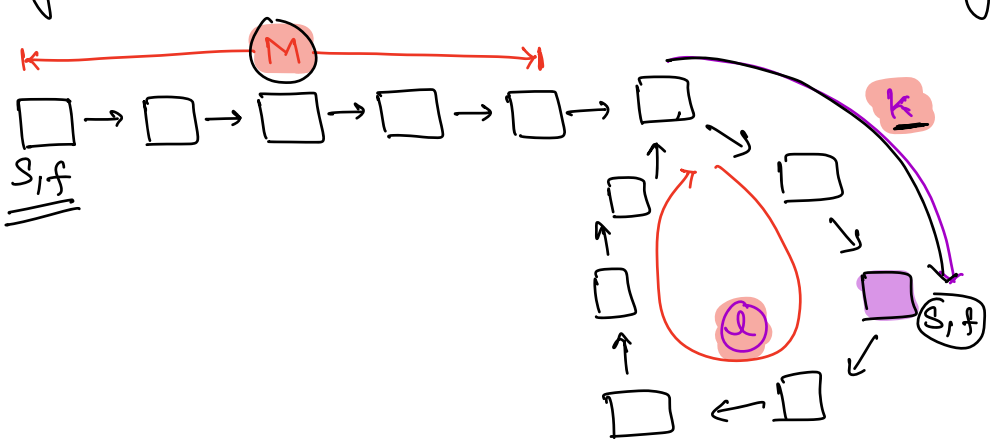
after l iterations ⟶ Starting node

after $2l$ iterations ⟶ Starting node

after $3l$ iterations ⟶ Starting node

⋮ ⋮

after $x \cdot l$ iterations ⟶ Starting node



$$dist(fast) = M + xl + K$$

$$dist(slow) = M + yl + K$$

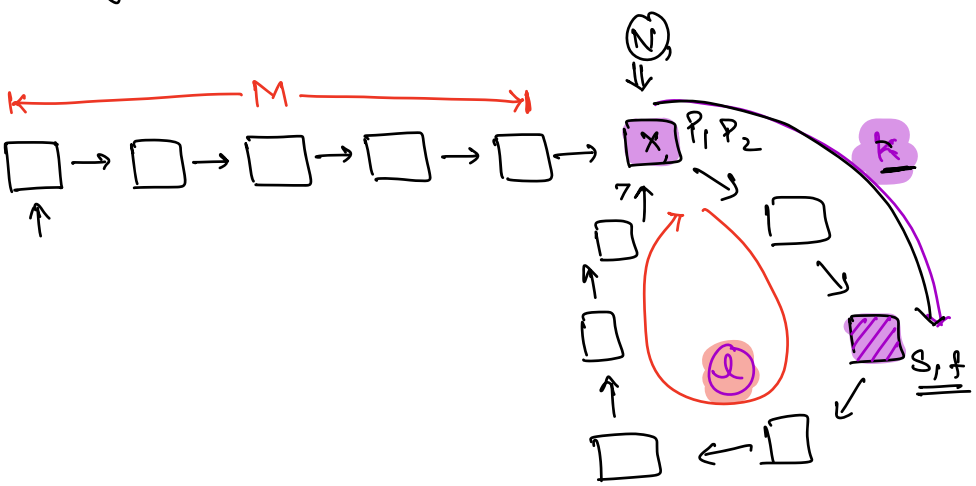
$$\text{dist(fast)} = 2 * \text{dist(slow)}$$

$$M + xl + K = 2(M + yl + K)$$

$$M + xl + K = 2M + 2yl + 2K$$

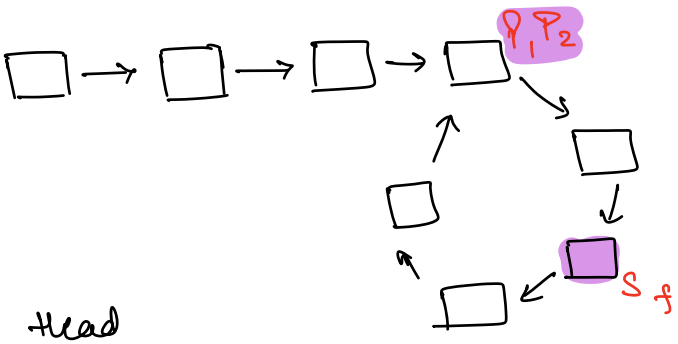
$$xl - 2yl = M + K$$

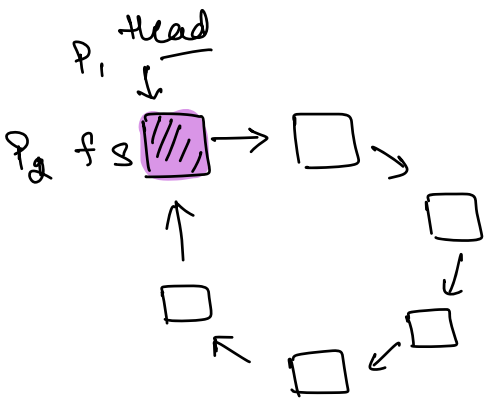
$$2 * y = (x - 2y)\,l = M + K$$

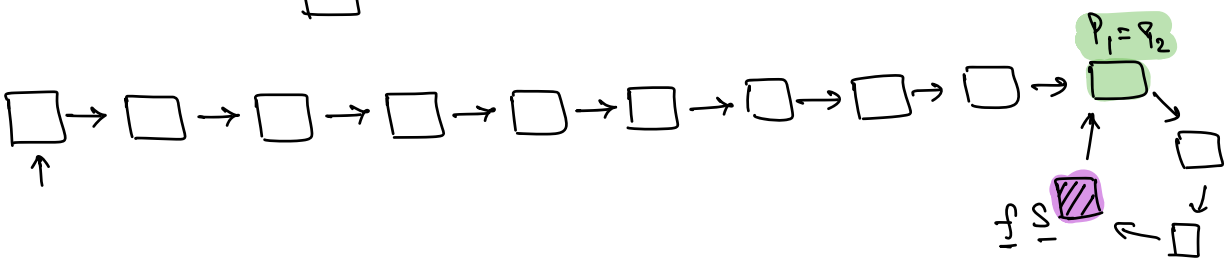


* If we start iterating from 1st node of cycle and do $(M+K)$ iterations :-
  ⇒ Then we'll end up at the start node only.

* Meeting pt. is already (K) distance from start node of cycle. So (M) iterations from meeting point will take to the start of cycle.







* Floyd's Cycle Detection Algorithm

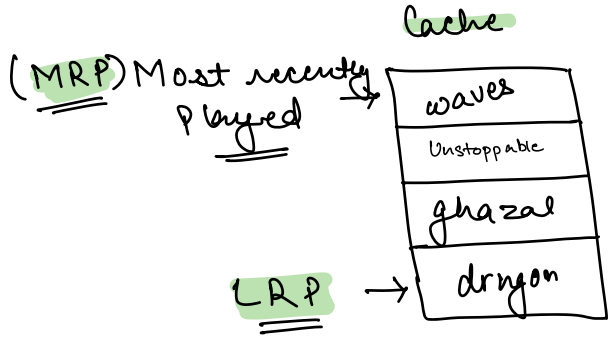
TC : O(N)

SC : O(1)

* Cache :-

⇒ Small DB which is very fast access.

⇒ Cache HW is very costly.

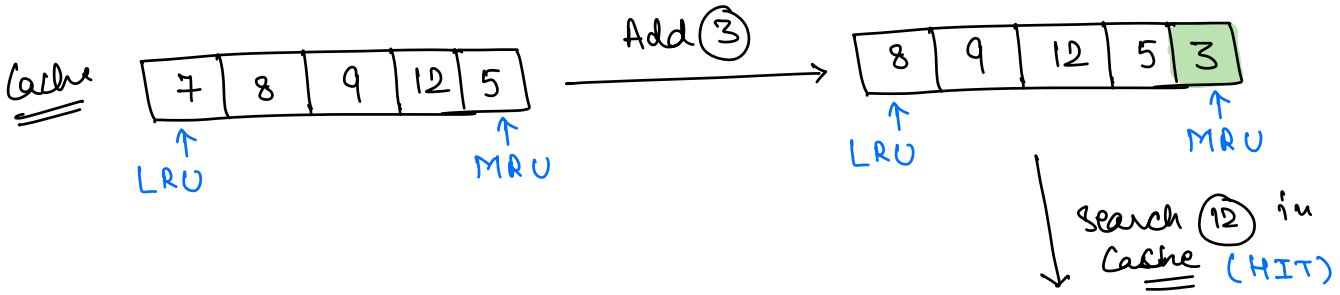


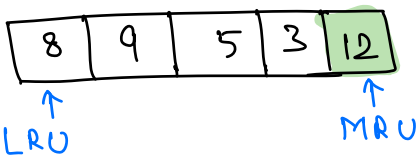
Cache Eviction : deleting from the cache when cache is full.

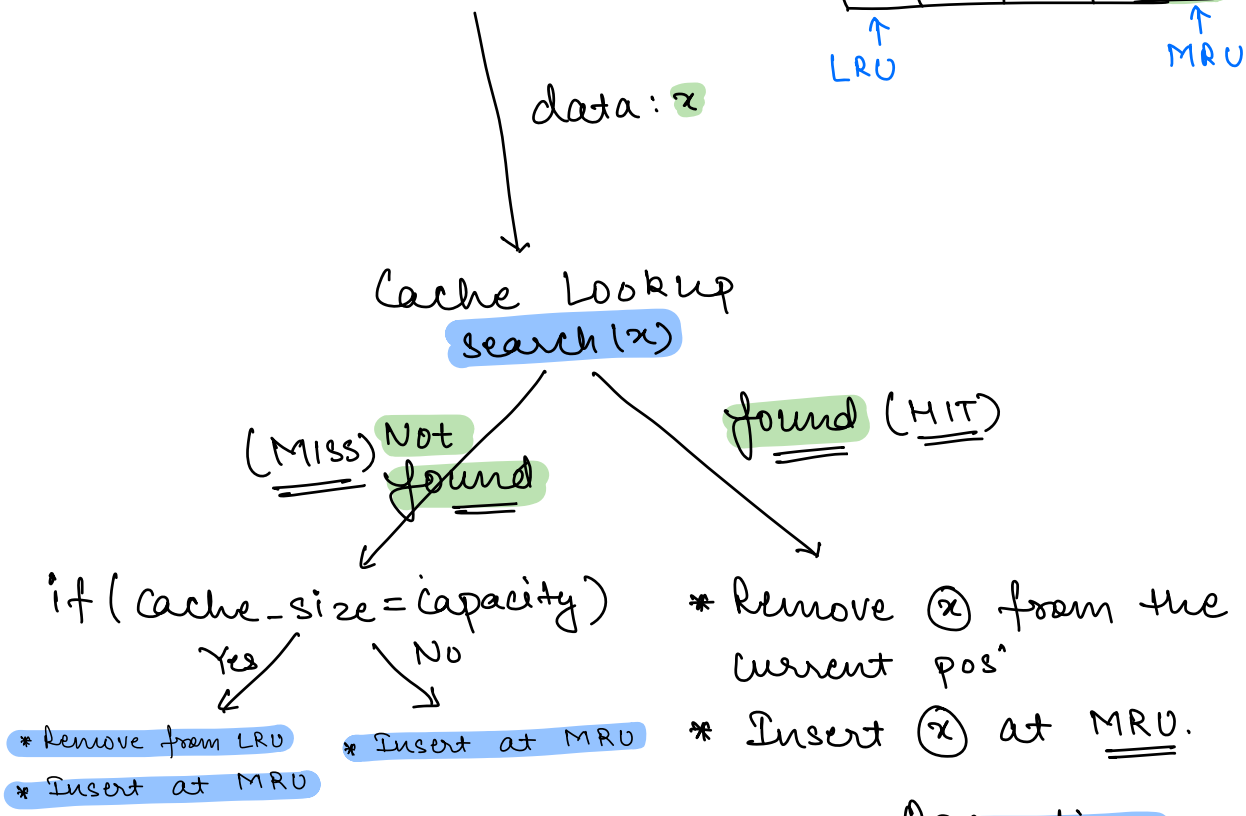
- FIFO
- Least frequently used
- Randomly.
- Least Recently Used (LRU)

* LRU Cache

id : 7, 8, 9, 12, 5, 3, 12







Operations

1) Search
2) Delete
3) Insert

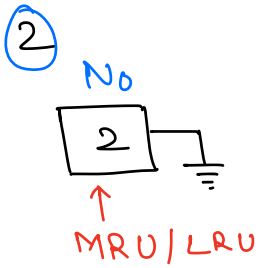
	Arrays	L.L
Search	O(N)	O(N) → O(1) (Set/Map)
delete	O(N)	O(1) (Search is already done)
insert	O(1)	O(1)

* L·L is more suitable.

* Searching can be optimized in L·L using Set / Map.

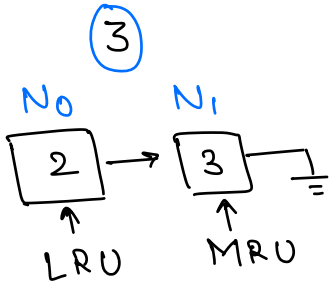
HashMap < int, ListNode >

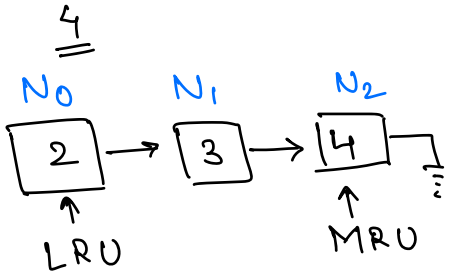
Cache_size = 3

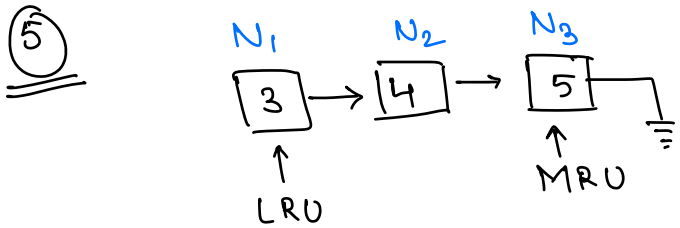


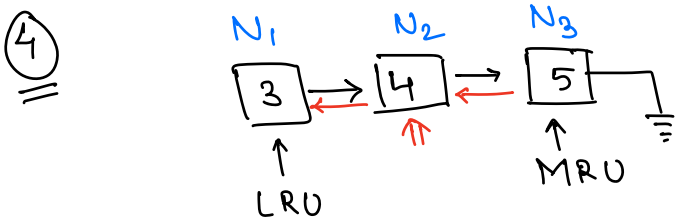
HashMap

id	:	Node
~~2~~		~~N0~~
3	:	N1
4	:	N2
5	:	N3

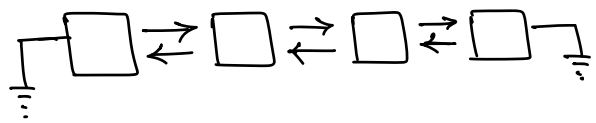




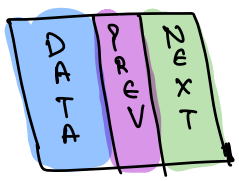




* Doubly L.L



Node




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

HW
* Implement LRU cache using DLL + Map.

———— * ————