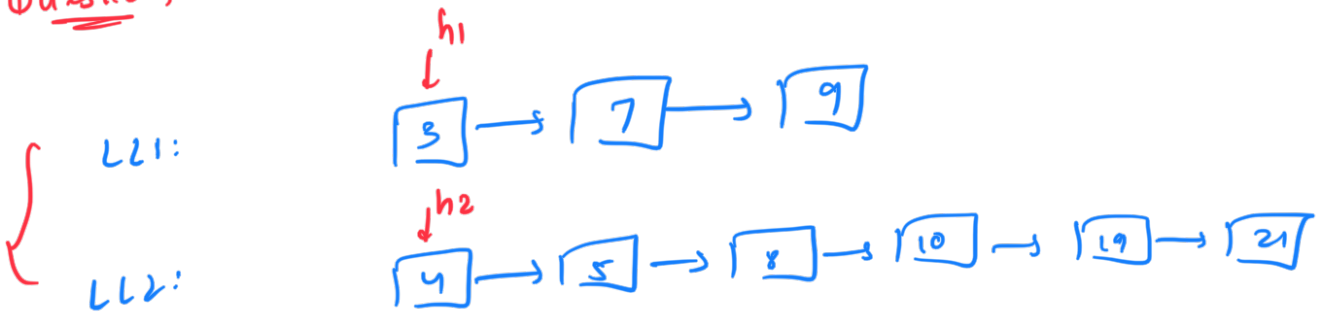
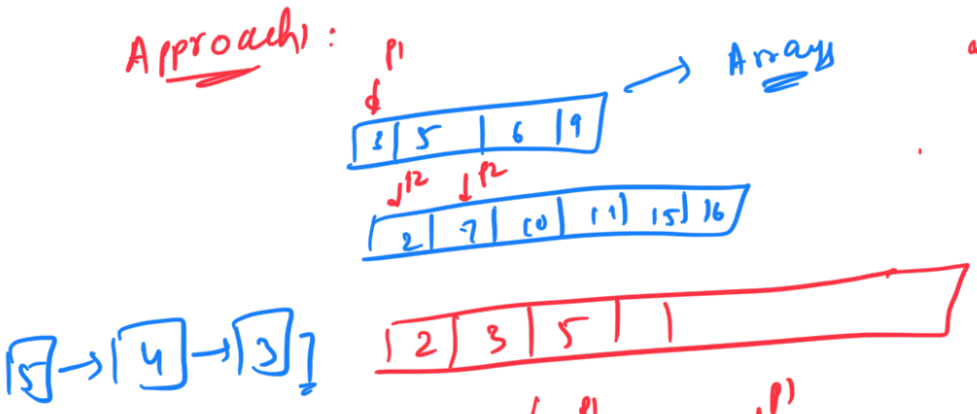
Linked List - 2

Question: Merge 2 sorted Linked Lists

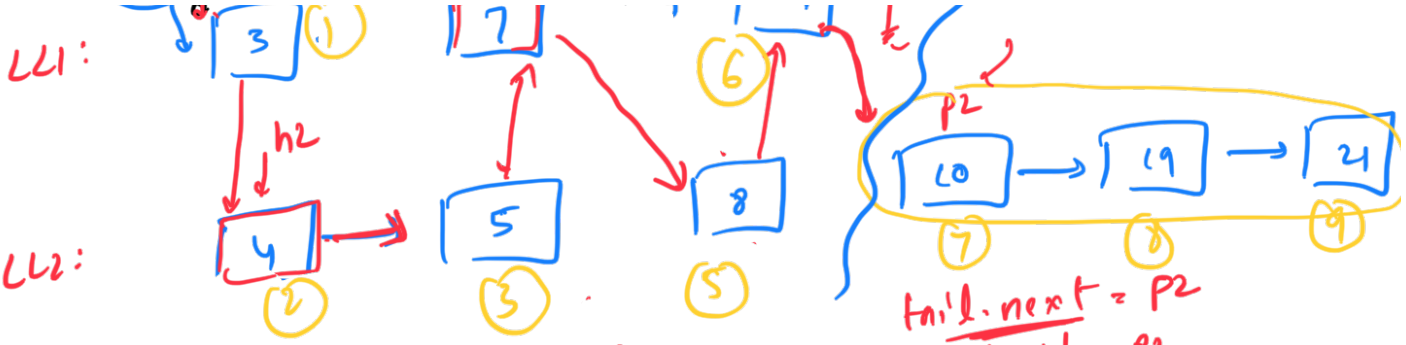


head = NULL

Approach 1:



Approach 2:



head.next = p2;

head.next = p2 ?

tail.next = p2
tail = p2
if(p1 == NULL)
tail.next = p2

reverse (output)

if(p1 == NULL)
tail.next = p2;

if(p2 == NULL)
tail.next = p1;

h1, h2

Base case:
if(h1 == NULL)
return h2;
if(h2 == NULL)
return h1;

T.C: O(m+n)

O(min(m,n))

LL1: 1 2 10

LL2: 2 3 4 5 6 7 8 9

T.C: O(m+n)

Extension: Merge Lists in Descending Order

Approach1:

in Increasing Order

1) Merge

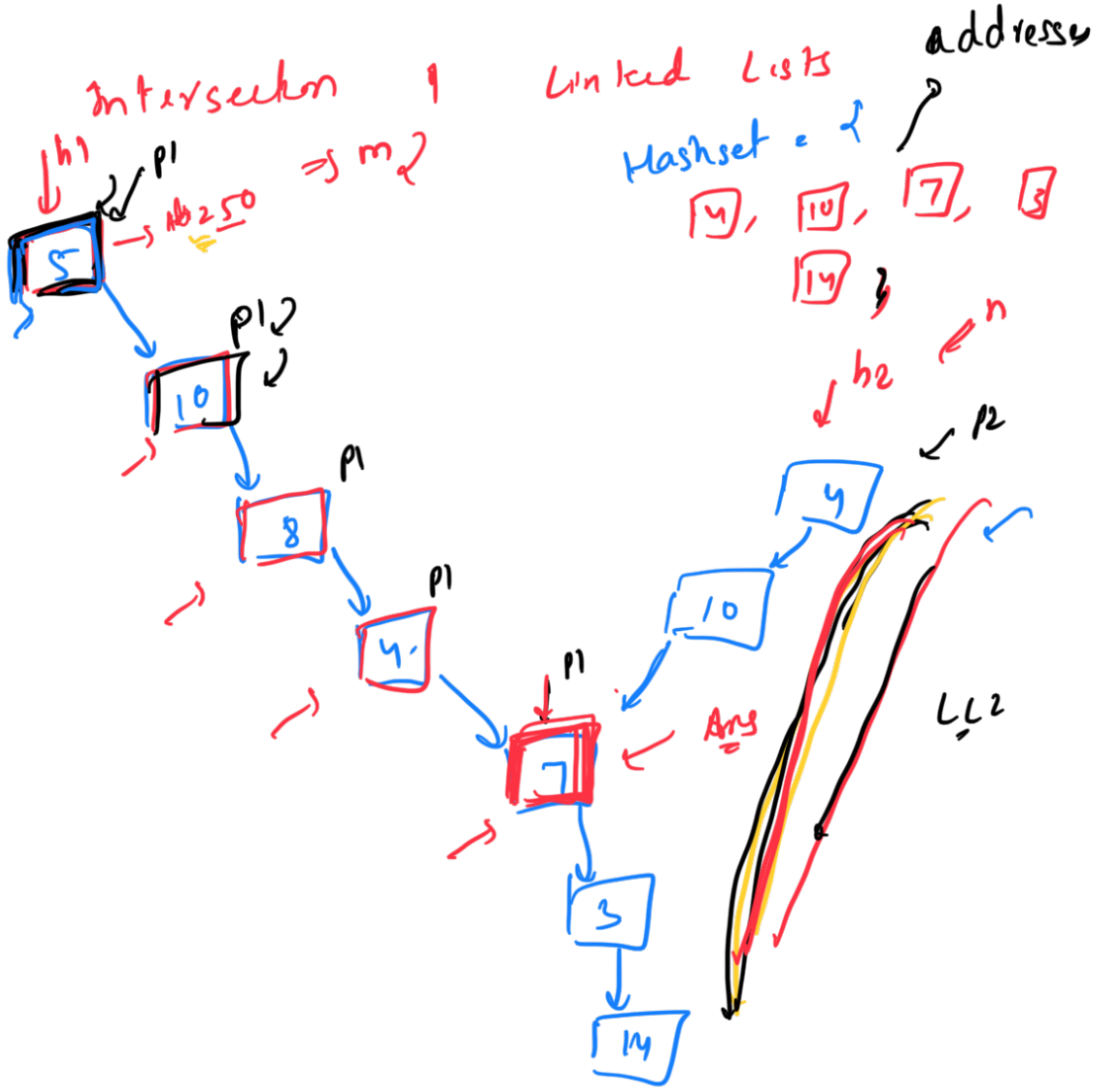
2) Reverse

O(1)

Approach2:

-> Insert at the Beginning of the Linked list

Question: Intersection of Linked Lists => m2



T.C:

Brute Force:

For every node in LL1, check if it is the intersection by iterating in LL2:

T.C: O(m·n)

S.C: O(1)

Approach2:

1) Iterate LL2 and hash the addresses using hashset. O(N)

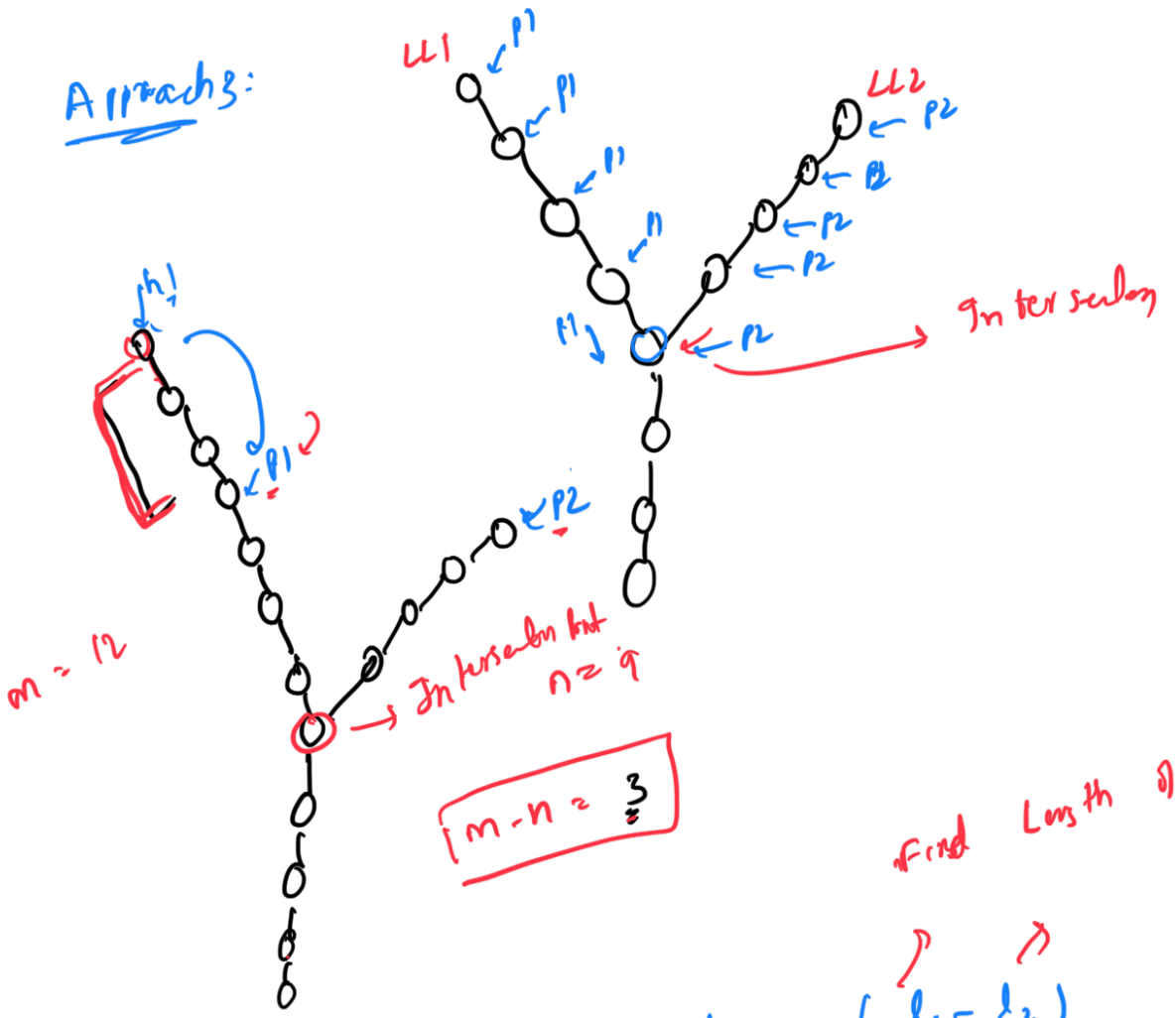
2) Iterate LL1 and check if node exists in the hashset. O(M)

⇒ O(M+N)

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

⇒ unordered_set

Approach3:



Find Length of LL1, LL2

$(l_1 - l_2)$

1) Move P1 by
2) Move P1, P2 until they meet
3) Meeting point is intersection

O(m+n)

T.C. :

S.C: O(1)

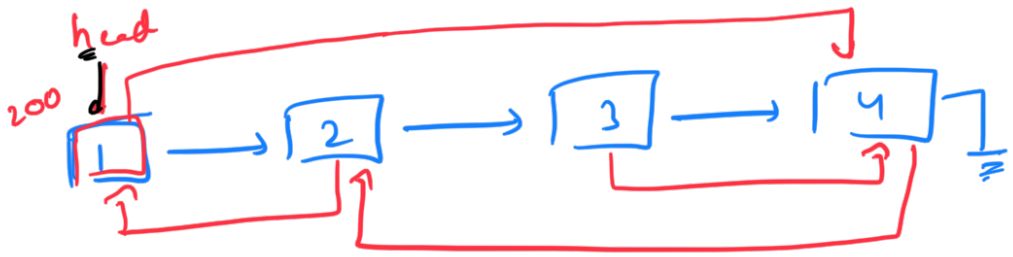
Question: Clone a Linked List (Deep Copy) with random pointers

3 round

3-4

10-12

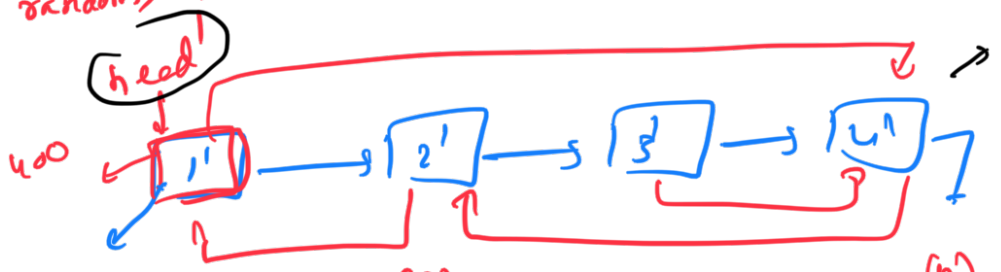
5-6




```
class Node {
    int data;
    Node next;
    Node random;
}
```

2 Pointers

Clone this LL

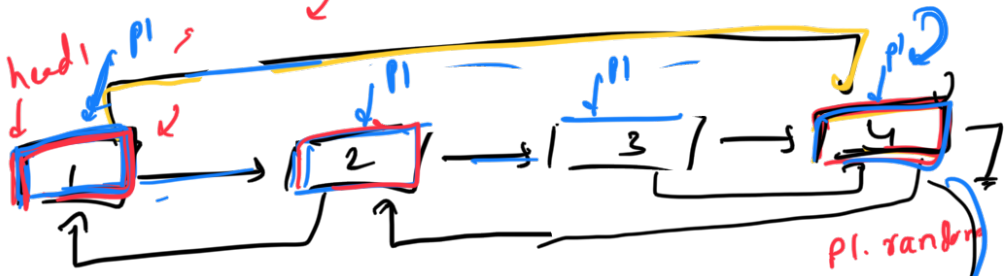


O(n) O(n) O(n)

dist = 3

Approach:

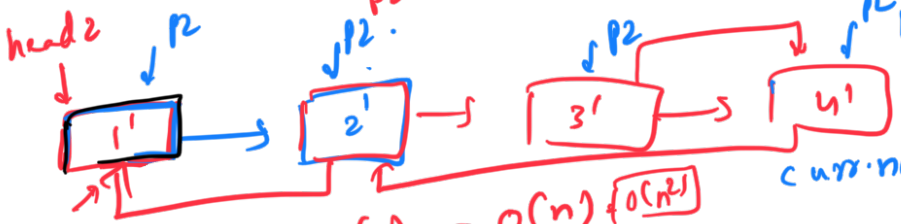
LL1.



p1.random

p2.random = p1.random

LL2:

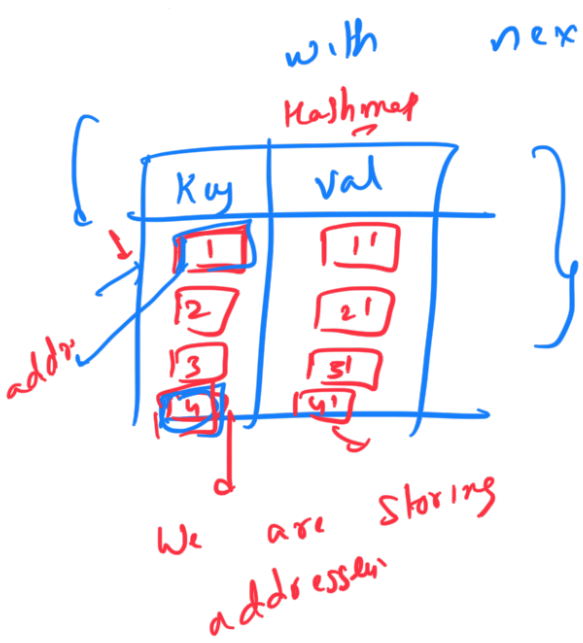


O(n)+O(n) = O(n) O(n²)

curr.next = 2'

curr.random = 4'

First, Let us create all the nodes pointers.



p2.random = mp[p1.random]

T.C: O(n) + O(n) = O(n)

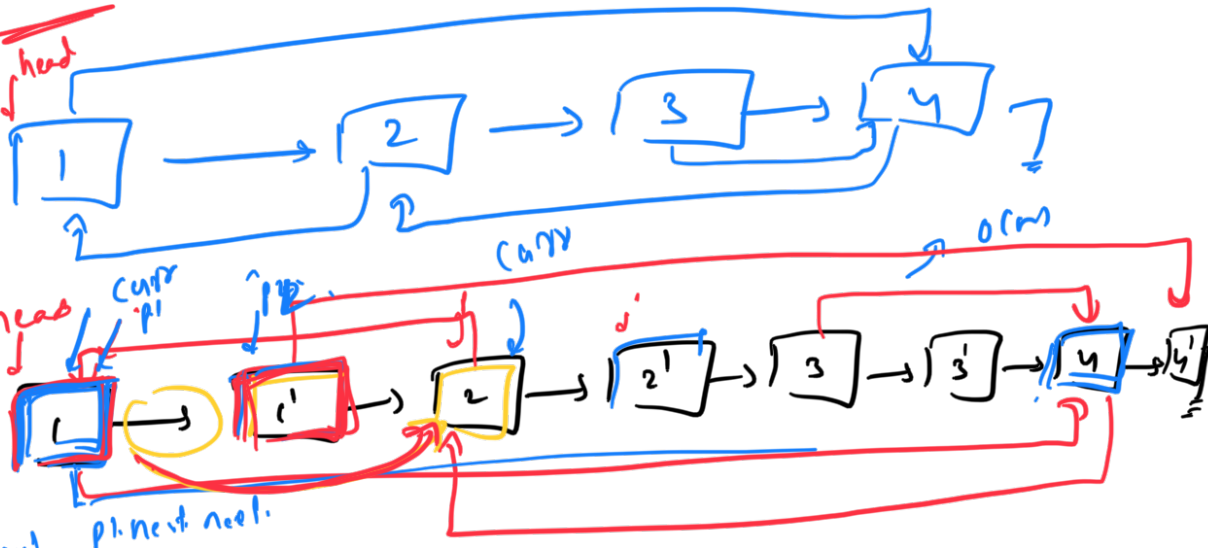
S.C: O(n)

HashMap<Node, Node> mp; → Java

map<Node*, Node*> mp

O(1) space

Optimised Approach



p1.next = p1.next.next

p2.next = p1.next.next

Copied Node curr = (curr.next)

curr.next.random = ?

curr.random.next

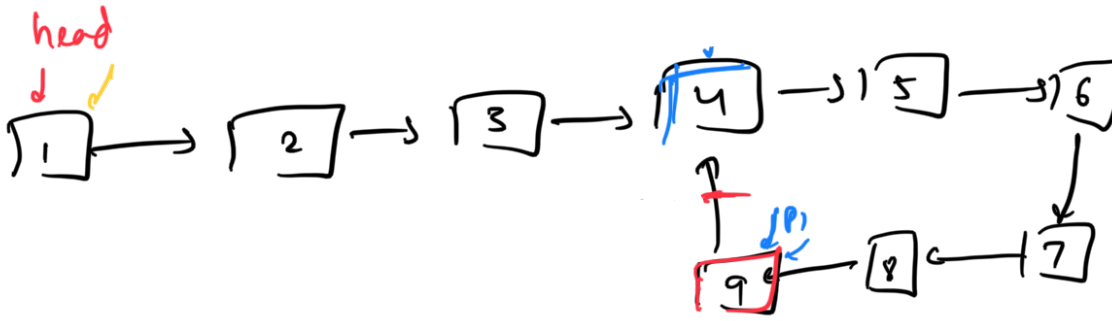
curr = curr.next.next

T.C: O(n)

S.C: O(1)

Curr-random.next

Question: Detect and Remove loop from a Linked List.



Approach:

If we visit a node which is already visited ⇒ LOOP

Hashset : { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

→ Find the first node whose next is already visited

```
while (curr.next is not in set){
    set.insert(curr);
    curr = curr.next;
}
```

curr != NULL

curr.next = NULL → Edge Case

T.C: O(n)

S.C: O(n) → Hashset

Floyd's cycle detection Algorithm

Tortouse Algo

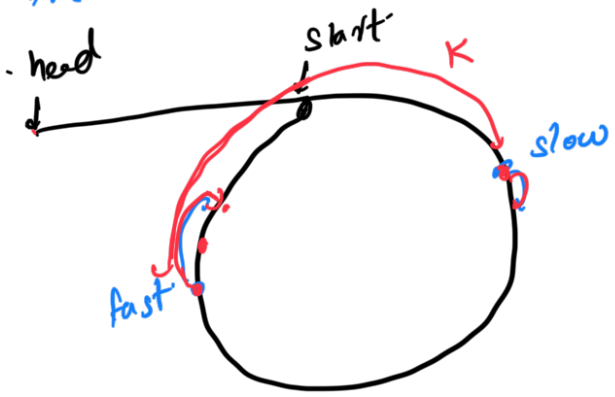
Hare and

slow, fast

x, 2x

If slow & fast pointers meet ⇒ LOOP

How can we guarantee these pointers if there is a loop?

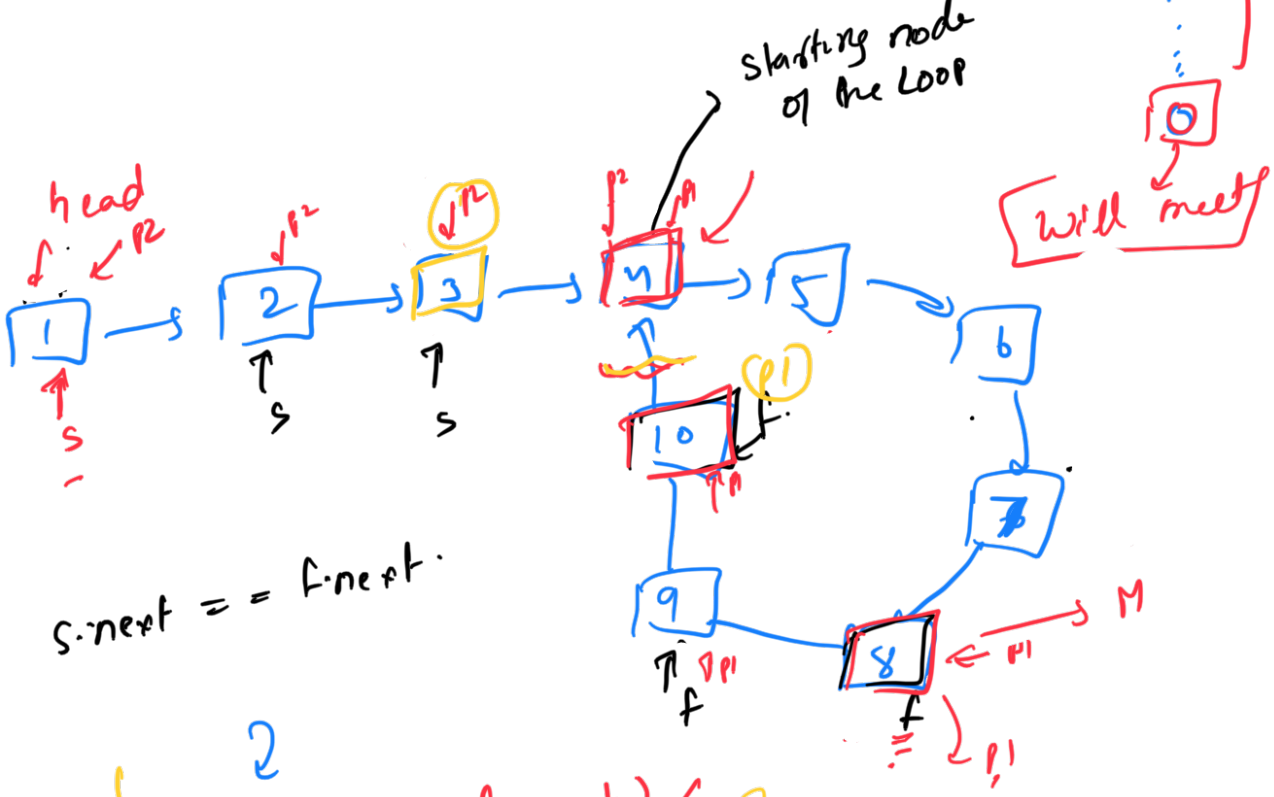


$d_1 = K$

$d_2 = K-1$

$d_3 = K-2$

$K-3$



s.next == f.next.

2

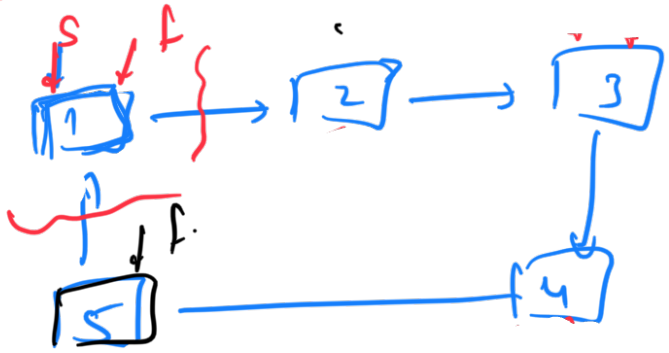
```
while ( s.next != f.next) {
    s = s.next;
    f = f.next;
}
```

3

meeting

if (head == fast)

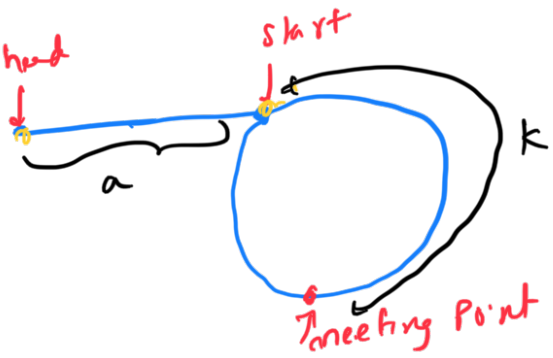
f.next = new



while (fast.next != new)
fast = fast.next

f.next = null;

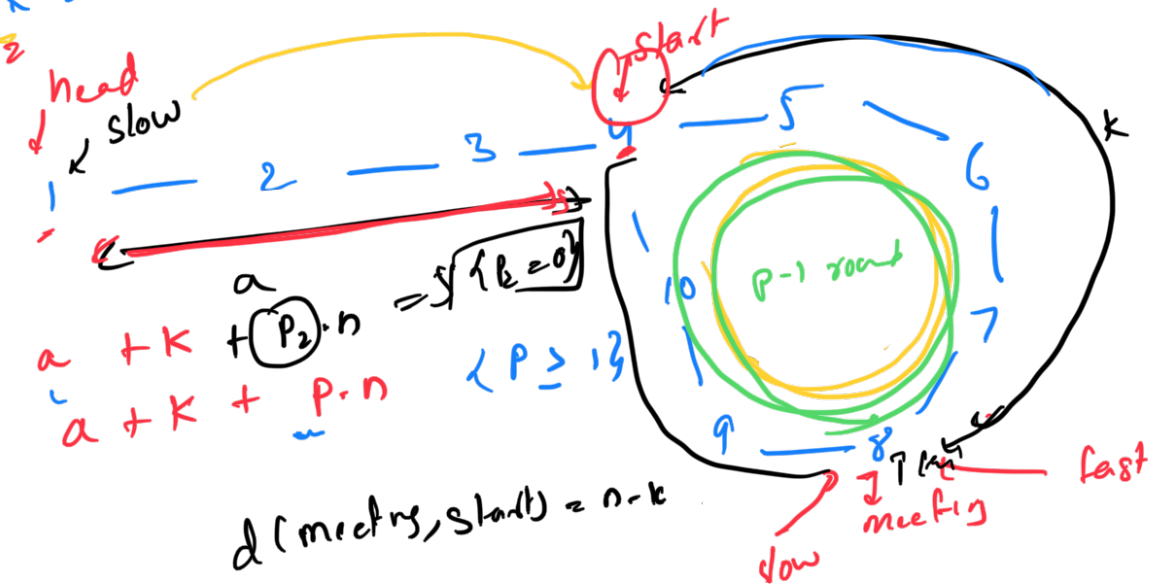
Proof :



a = Length from head to start. = 3

n = Length of Loop = 7

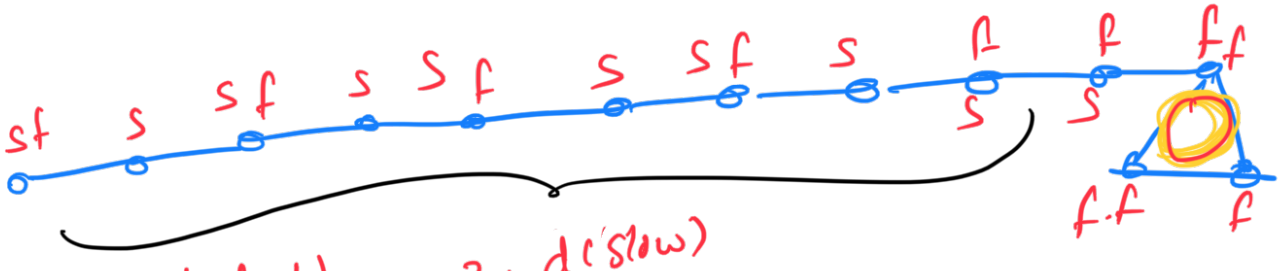
k = Length from start to meeting Point = 4



slow = a + k + $(p_2 \cdot n)$ $\Rightarrow$ if (k=0)

fast = a + k + p.n (p ≥ 1)

d(meeting, start) = n - k



d(fast) = 2 · d(slow)

a + k + p·n = 2a + 2k

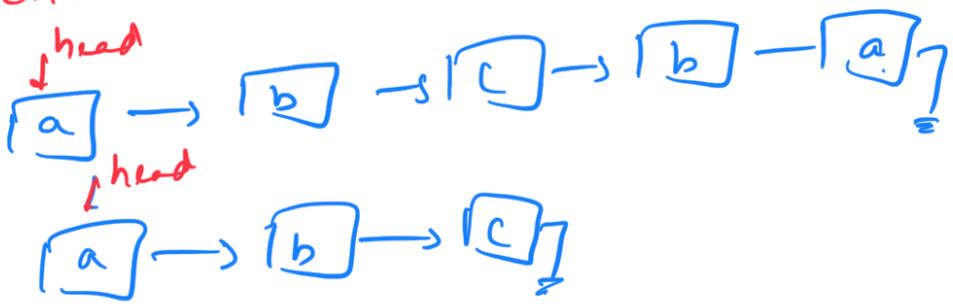
⇒ a + k = p · n { p ≥ 1 }

a = p·n − k

a = (p−1) n + (n − k) { p ≥ 1 }

a : distance travelled by slow pointer

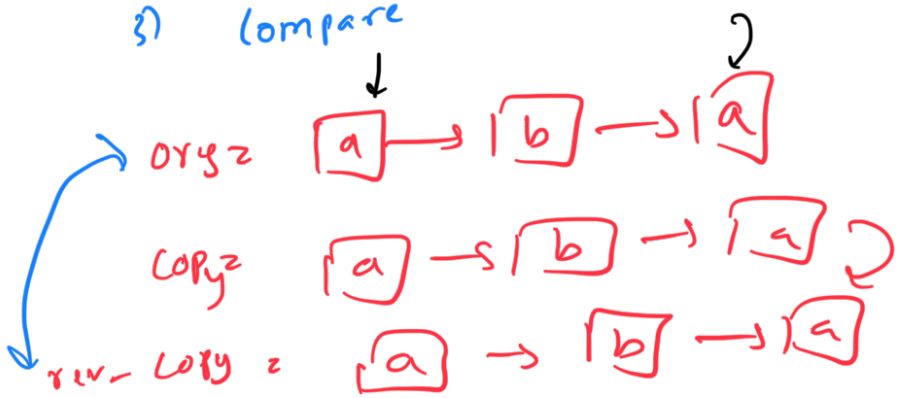
Question: check if a LL is palindrom



Approach 1:

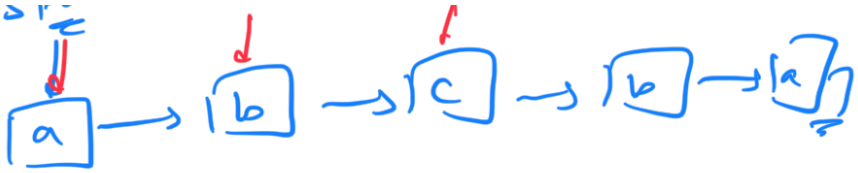
1) Make a copy of LL
2) Reverse
3) Compare

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



walk

Approach:

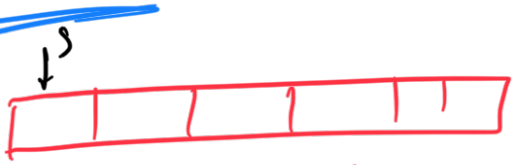


Stack.

T.C: O(n)

S.C: O(n) → Stack

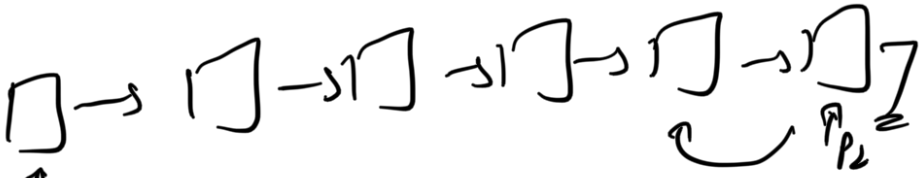
Efficient Approach:

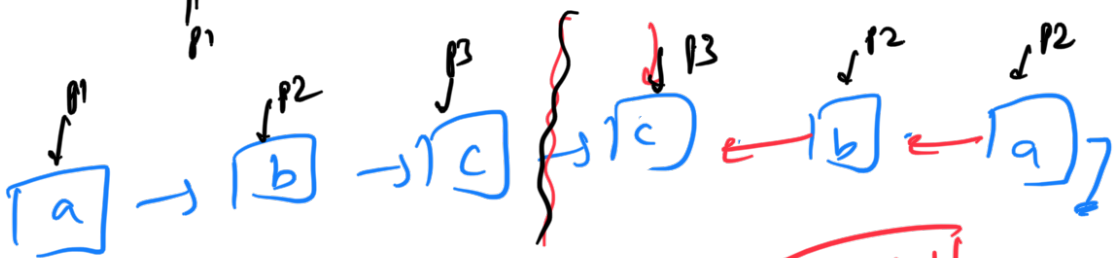


string = a b c d c b a

p1 p2

First half should match with reverse of second half.





Slow, fast

Even, odd

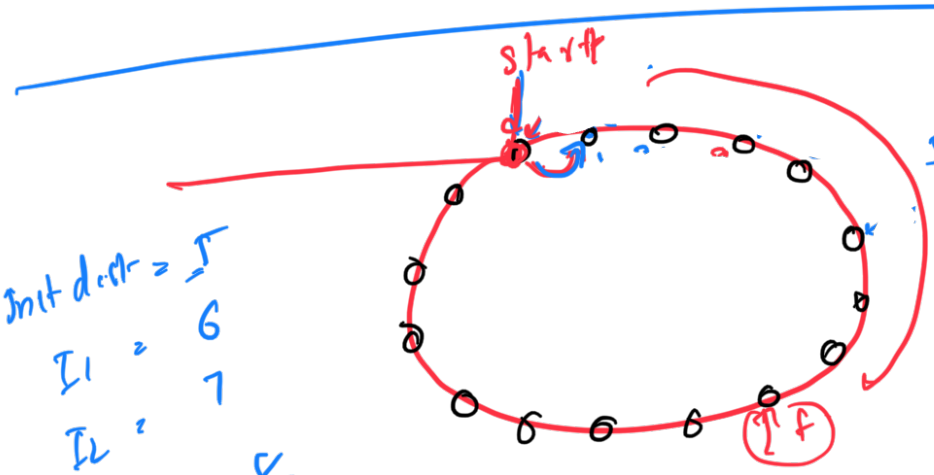
T.C:

Steps

1) Find Middle : O(n), O(1)
2) Reversing the 2nd half : O(n), O(1)
3) Match using P_1, P_2 : O(n), O(1)
4) Revert the 2nd half : O(n), O(1)

T.C: O(n)

S.C: O(1)



Length of loop = k (2, 3, 4 ... ?)

Init dist = 7

I1 = 6

I2 = 7

:

= k

s & f have met

dist(slow, fast) = 1

$d_2$ist(s, f) = 2

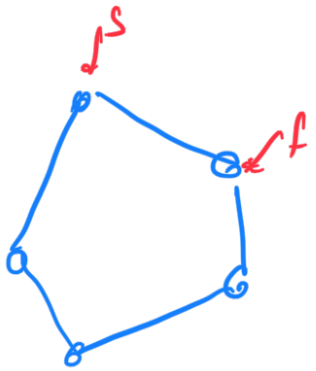
d_3(s, f) = 3

:

d(s, f) =

After (k-1) steps Meeting

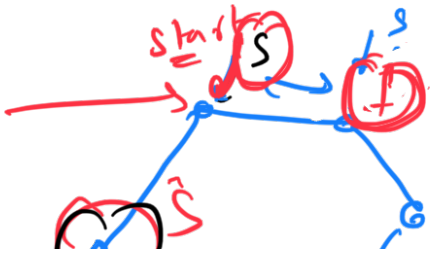
k =



(k-1) steps

T_0 = 1

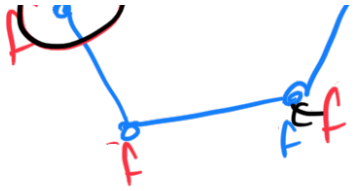
T_1 = 2



k = 6

T_0 = 1

(K-1)



$T_1 = 2$

$T_2 = 3$

$T_3 = 4$

$T_4 = 5$

$T_5 = 6$

After (K-1) iterations => slow & fast are meeting

$T_0 = 3$

$T_1 = 4$

$T_2 = 5$

$T_3 = 6$

3 iterations << K, they are meeting

Question: Merge 2 sorted linked lists

```
Node* mergeTwoLists(Node* A, Node* B){
      Node *p1 = A, *p2 = B;
      Node *head = *tail = NULL;

      if(A == null) return B;
      if(B == null) return A;

      if(p1->data < p2 ->data)
            head = tail = p1; p1 = p1->next;
      else
            head = tail = p2; p2 = p2->next;

      while(p1 != NULL && p2!= NULL){
            if(p1->data < p2->data){
                  tail -> next = p1;
                  tail = p1;
                  p1 = p1->next;
            }
            else{
                  tail -> next = p2;
                  tail = p2;
                  p2 = p2->next;
            }
      }
      if(p1 != NULL)  tail->next = p1;
      if(p2 != NULL) tail->next = p2;
return newHead;
}
```

Question: Clone a linked list

```
Node* clone(Node* head){
      unordered_map<Node*, Node*> mp;

      Node* curr = head;
      while(cur != NULL){
            Node* cloneNode = new Node(cur->data);
            mp[curr] = cloneNode;
            curr = curr->next;
      }

      curr = head;
      while(curr != NULL){
            cloneNode = mp[curr];
            cloneNode->next = mp[curr->next];
            cloneNode->random = mp[curr->random];
            curr = curr->next;
      }

return mp[head];
}
```

Detect and remove loop : Hashing approach

```
Node* removeLoopHashing(Node* head){
      Node* curr = head;
      unordered_set<Node*> st;
      while(curr && curr->next){
            if(curr not in set)){
                  st.insert(cur);
            }
            if(curr.next in set){
                  cur->next = NULL;
                  break;
            }
            cur = cur->next;
      }
 return head;
 }
```

Detect and remove loop

```
Node* removeLoopFloyd(Node* head){
      if(head == NULL || head->next = NULL)
            return head;

      Node* slow = head, fast = head;
      while(fast!=NULL && fast->next!=NULL){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast)
                  break;
      }

      if(slow != fast) NO LOOP!
      else if(slow == fast){
            slow = head;
            // When slow and fast meet at the head of LL
            if(head == fast){
                  while(fast->next != head)
                        fast = fast->next;
            }
            else{
                  while(slow->next != fast->next){
                        slow = slow->next;
                        fast = fast->next;
                  }
            }
            fast->next = NULL;
      }
return head;
}
```