Todays Content:

a) Merge

b) Merge sort

c) Re-arrange Linked list

d) Cycle detection

   i) Detect cycle

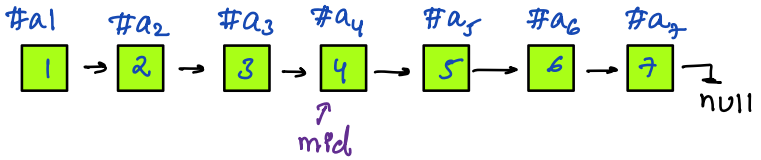
   ii) Find start of cycle

   iii) Remove cycle

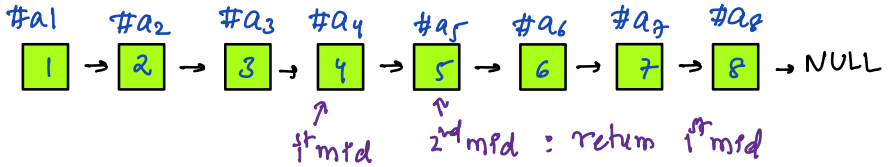
e) Find Intersection of linked list

10) Given head node find mid of linkedlist

Ex1:



Ex2:



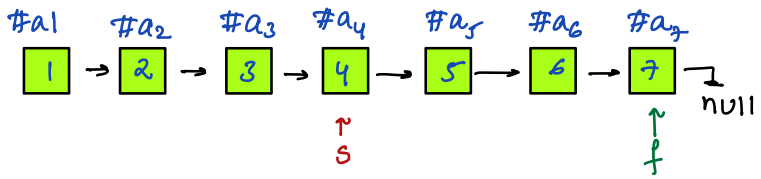
Idea: Iterate & get length, & goto to center node

TC: O(N) SC: O(1), Note: Traversing 2 times on linkedlist

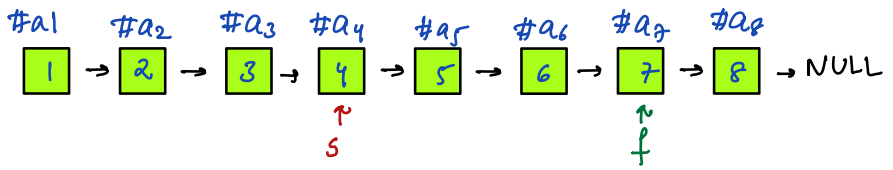
Idea2: Usage of slow and fast pointer

slow = slow.next

fast = fast.next.next

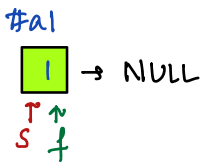


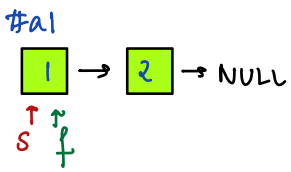
obs1: If f.next == null, s is at centre



obs2: If f.next.next == null, s is at centre

```
Node mid (Node h){ TC: O(N) SC: O(1)
    s = h, f = h
    if(h == NULL) { return h;
    while( f.next != NULL && f.next.next != NULL){
        s = s.next
        f = f.next.next
    }
    return s;
}
```

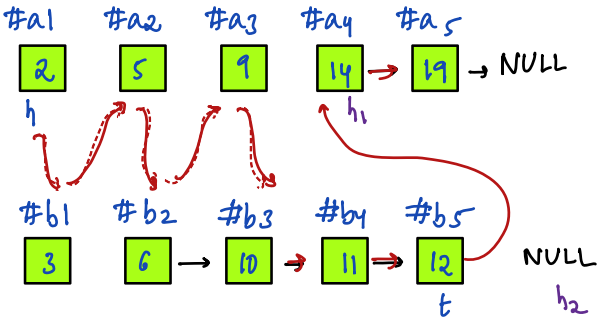




2Q) Given 2 sorted linked lists, Merge & get final sorted list

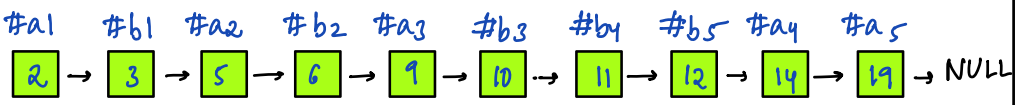
TC: O(N+M) SC: O(1)

Ex1:

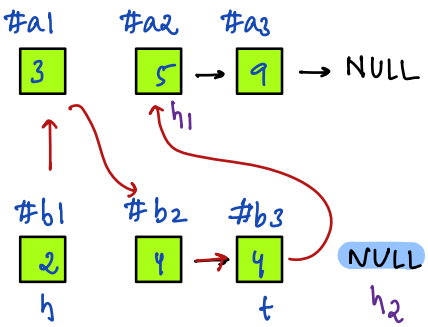


obs1: We will stop comparison when one of them reaches null

final:



Ex2:

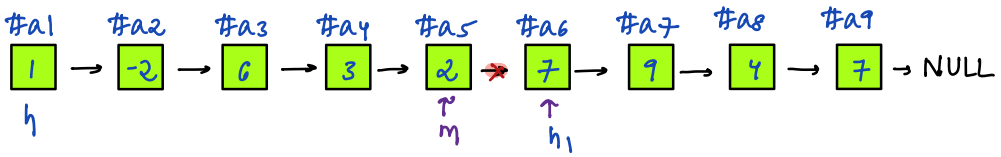


Psendo Code:

```
Node merge( Node h1, Node h2){   TC: O(N+M)  SC: O(1)

    if(h1 == null) { return h2 }

    if(h2 == null) { return h1}

    Node h, t;
    if( h1.data < h2.data){ h = h1, t = h1, h1 = h1.next }
    else { h = h2, t = h2 , h2 = h2.next }

    while( h1 != null && h2 != null) {

        if(h1.data < h2.data){ // h1 is min
            t.next = h1;  h1 = h1.next
            t = t.next
        }
        else { // h2 is min
            t.next = h2 ;  h2 = h2.next
            t = t.next
        }
    }
    // link left out linked list
    if( h1 != null) { t.next = h1}

    if( h2 != null) { t.next = h2}

    return h;
}
```

38) Merge Sort on Linked list:

Eg1:

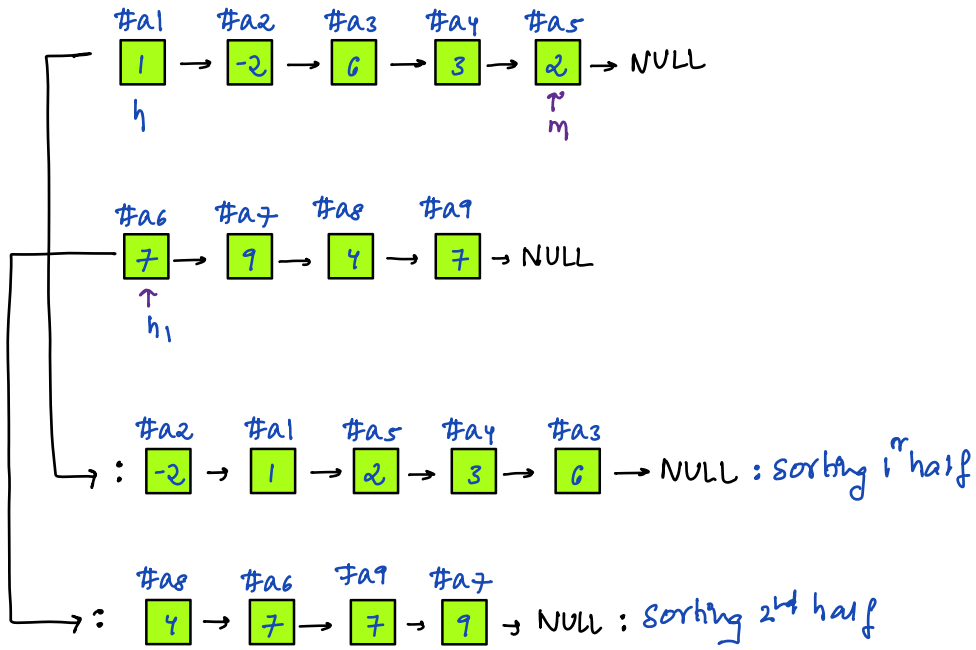


Step1: find the mid of linked list & divide into 2 linked list

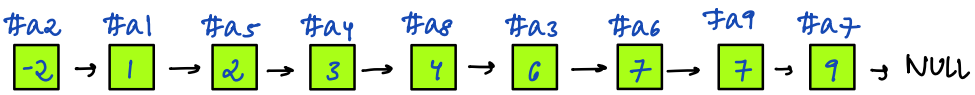
$h_1 = m.next$

$m.next = null$

Step2: Sort both linked lists using recursion



Step:3 Merge both sorted linked list

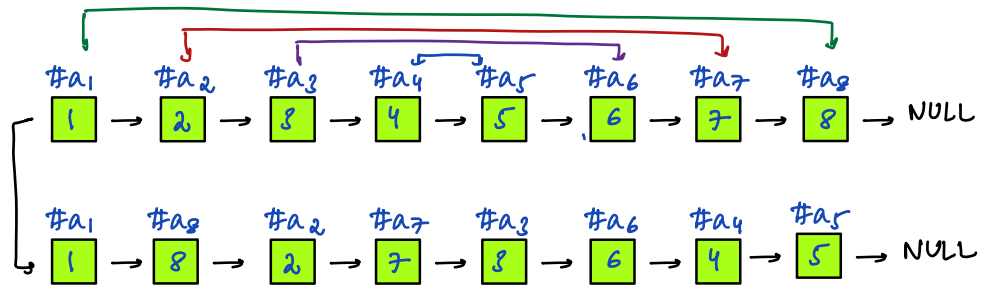


Ass: Given linked list, sort & return head node of list

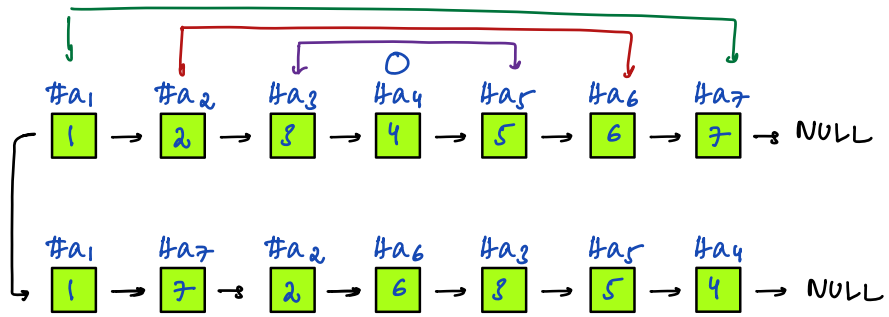
```
Node mergeSort(Node h){ TC: O(nlogn)  SC: O(logN) ≈ 0   // recursive stack spc
    if(h==null || h.next==null){return h;}
    Node m = middle(h) // it will return mid of linked list }
    Node h1 = m.next
    m.next = null
    Node t1 = mergeSort(h) // it will sort list & return head
    Node t2 = mergeSort(h1) // it will sort list & return head
    Node t3 = merge(t1, t2) // merge 2 sorted lists & return head
    return t3
}
```

Re-arrange : Expected TC: O(N) SC: O(1) TODO 10:30 → 10:40

Eg1:

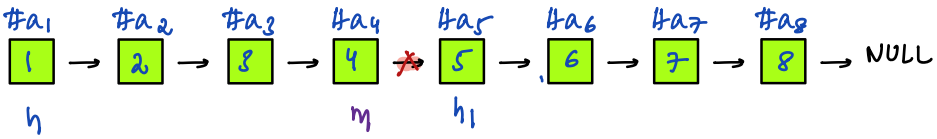


Eg2:

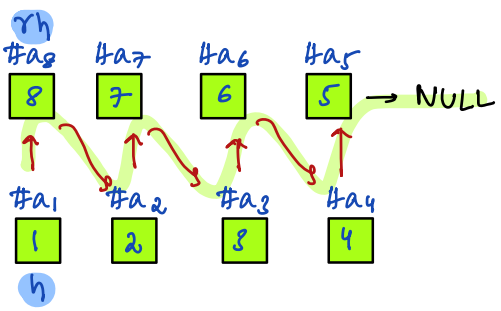


Idea:

Step1: Find mid & separate 2 linked lists



Step:2 Reverse 2nd linked list

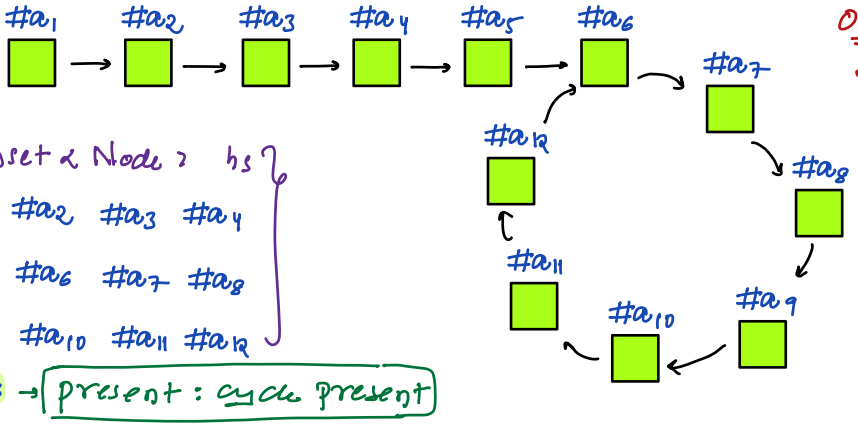


Step:3 Re-arrange links in h & rh so that {TODO}
we get 1 node from h & another from rh & so-on

4Q) Given a head node of linkedlist, check for cycle detection?

Extra Space:

Ex1:



Obs: If cycle is present, there is no null in linked list

hashset < Node > hs

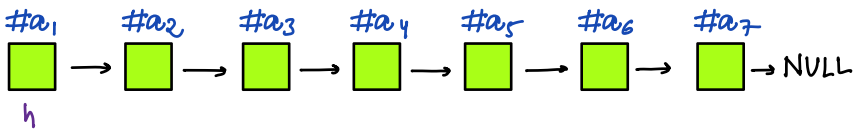
#a1 #a2 #a3 #a4

#a5 #a6 #a7 #a8

#a9 #a10 #a11 #a12

#a6 → present : cycle present

Ex2:



hashset < Node > hs { #a1 #a2 #a3 #a4 #a5 #a6 #a7 }

NULL : no cycle

Idea:

Iterate on listlist & insert all address

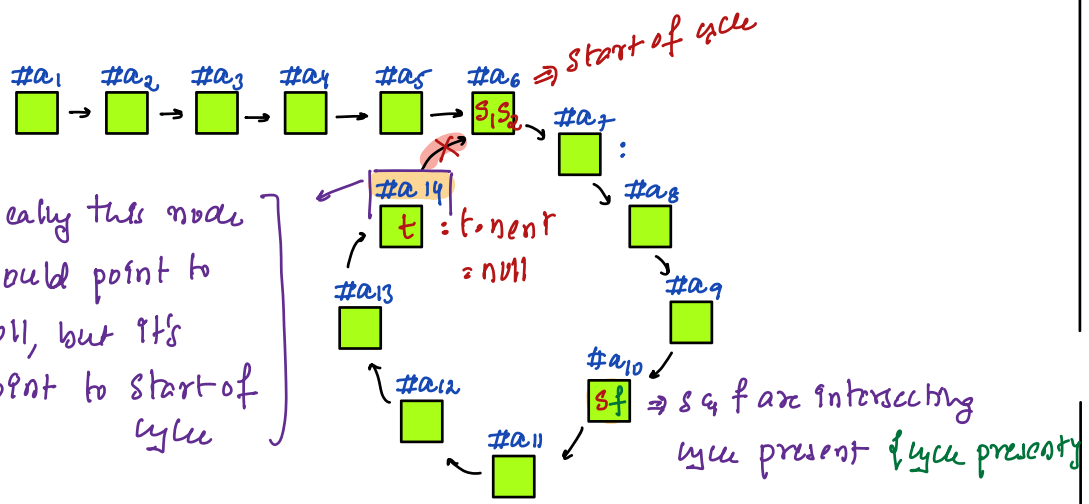
1) If an address is already present in hs : Cycle present

2) If we reach null : Cycle absent

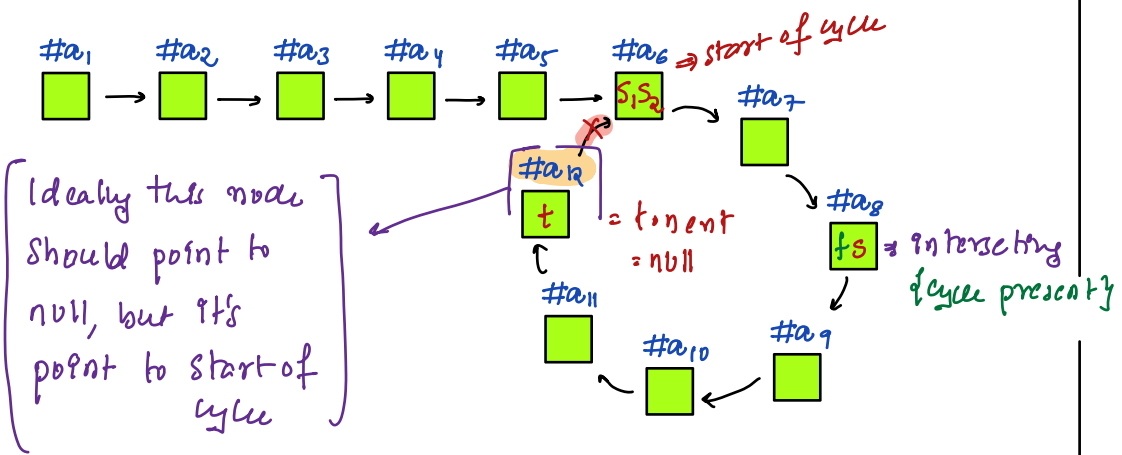
TC: O(N) SC: O(N) // Insert addres in hashset

Idea 2: Using slow & fast pointer

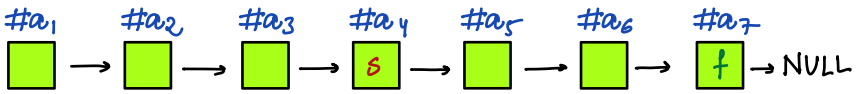
Eg 1:



Eg 2:



Eg 3:

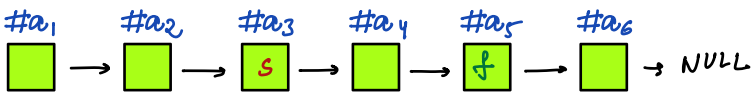


f.next == null & no cycle

& Use both conditions to stop}

f == null & no cycle

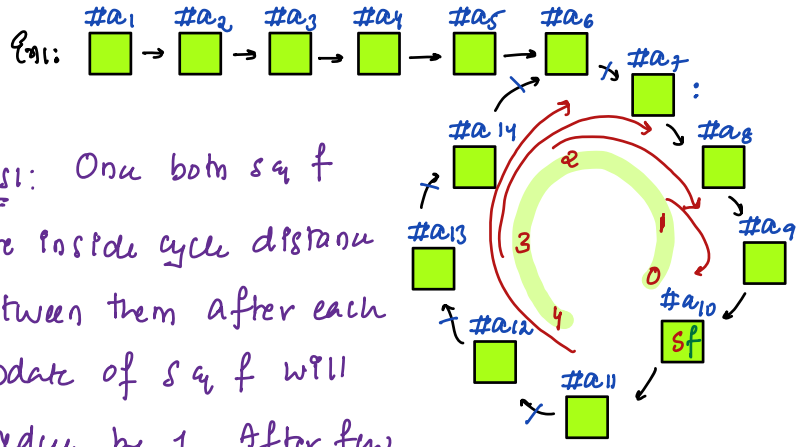
Eg 4



Idea: Take s & f pointers, initialize them to head

s = s.next
f = f.next.next
} If both are matching cycle present?

Why should it intersect? If there is a cycle?



Obs1: Once both s & f are inside cycle distance between them after each update of s & f will reduce by 1, After few iterations distance will finally 0, They are intersecting

Start of cycle? {Proof: doubt session}

: Take 2 slow pointers s_1 & s_2

: s_1 = h {head of list} s_2 = {Intersection of s & f}

: Keep updating till both of them intersect, Their intersection point is start of cycle

Remove cycle:

: Take t, initialize of cycle

: Iterate on cycle, untill t.next = start of cycle

: t.next = null / break the cycle

Q): Given linked list If cycle present:
: remove cycle : last node should point to null
: return start of cycle
If cycle not present : return null

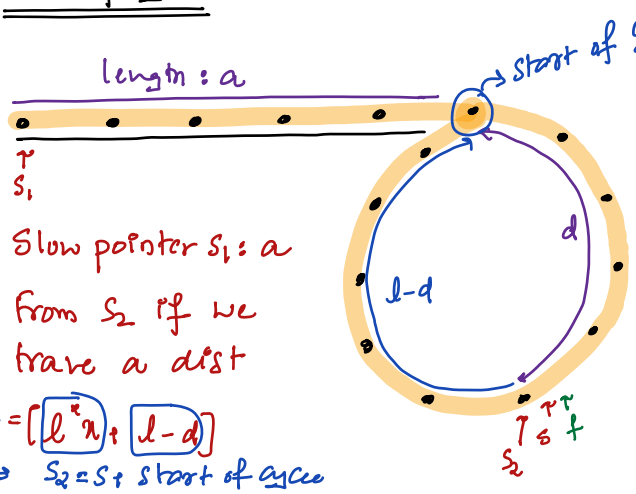
```
bool detectCycle(Node h){
    Node s = h
    Node f = h
    bool isCycle = false;
    while(f!=null && f.next!=NULL){
        s = s.next
        f = f.next.next
        if(s==f){
            isCycle = True, break;
        }
    }
    if(isCycle == false){return null}
    // find start of cycle?
      Node s1 = h, s2 = s or f {Both same}
      while(s1 != s2){
          s1 = s1.next
          s2 = s2.next
      }
      Node t = s1 // start of cycle
      while(t.next != s1){
          t=t.next // Keep iterating till we find node whose next as start of cycle
      }
      t.next = null // break cycle
    return s1 // start of cycle
}
```

Detect Cycle

find start of cycle

removing cycle

Start of cycle? {proof : doubt session}



→ Slow pointer s_1 : a

→ From s_2 if we trave a dist

$= [l^* n + (l-d)]$

↳ $s_2 = s_f$ start of cycle

As1: length of cycle : l

As2: start to intersection of s/f : d

$$a = l^* n + l - d$$

→ some number

d_s = {distance travelled by slow pointer} = $a + c_s^* l + d$

d_f = {distance travelled by fast pointer} = $a + c_f^* l + d$

// c_s = times s pointer iterate in cycle c_f = times f pointer iterate on cycle

obs: $d_f = 2^* d_s$

$$a + c_f^* l + d = 2[a + c_s^* l + d]$$

$$a + c_f^* l + d = 2a + 2c_s^* l + 2d$$

$$c_f^* l - 2c_s^* l = a + d$$

$$a = c_f^* l - 2c_s^* l - d$$

// Add & subtract l

$$a = c_f^* l - 2c_s^* l - l + l - d$$

// Say some number n

$$= l[c_f - 2c_s - 1] + l - d$$

$$a = l^* n + l - d$$