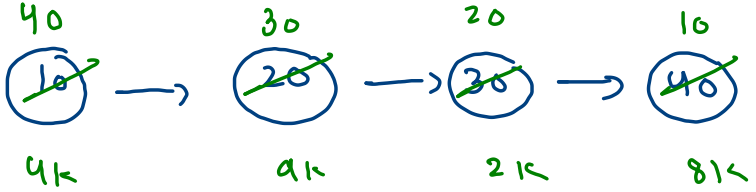
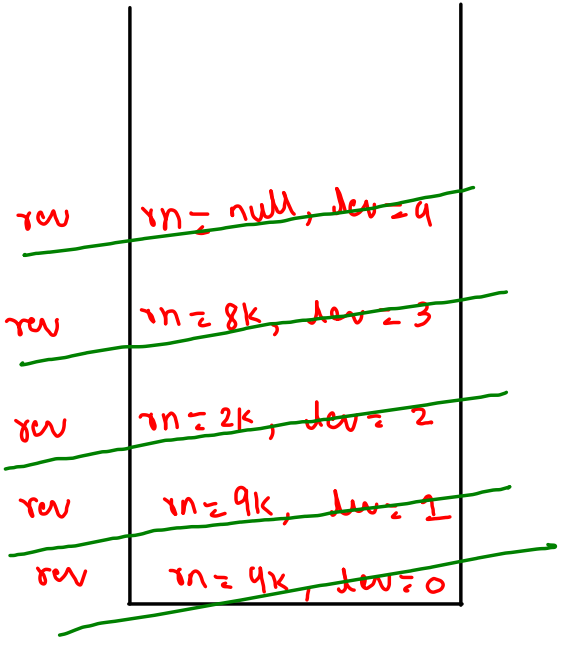
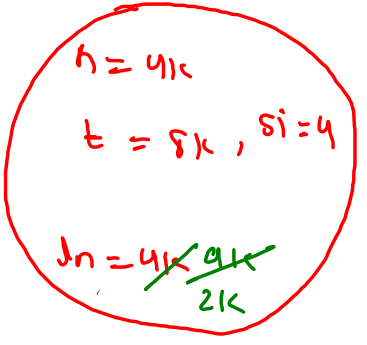
Reverse A Linked List (data Recursive)

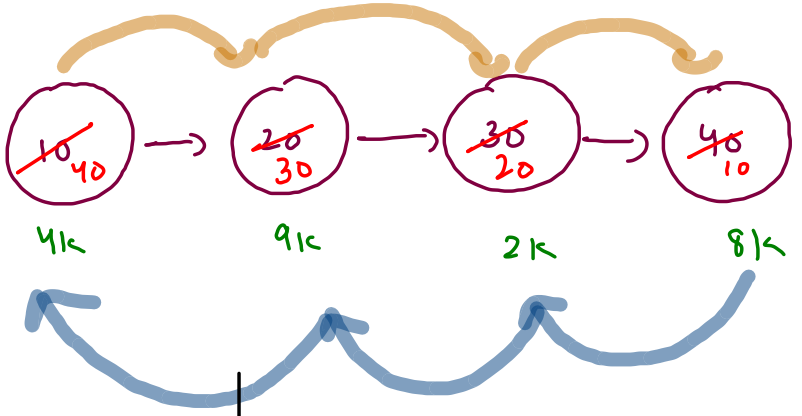



```java
Node ln;
public void reverseDR() {
  ln = head;
  reverseDRH(head,0);
}

private void reverseDRH(Node rn,int lev) {
    if(rn == null) {
        return;
    }

    reverseDRH(rn.next,lev + 1);

    if(lev >= (size/2)) {
        //swap
        int ld = ln.data;
        int rd = rn.data;
        ln.data = rd;
        rn.data = ld;

        ln = ln.next; //heap change (this.ln = this.ln.next)
    }
}
```

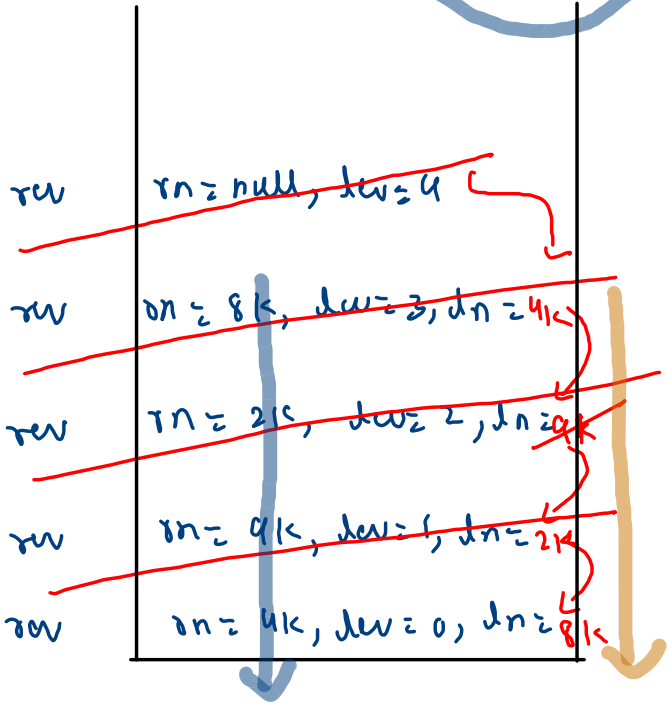




(size/2) = 2



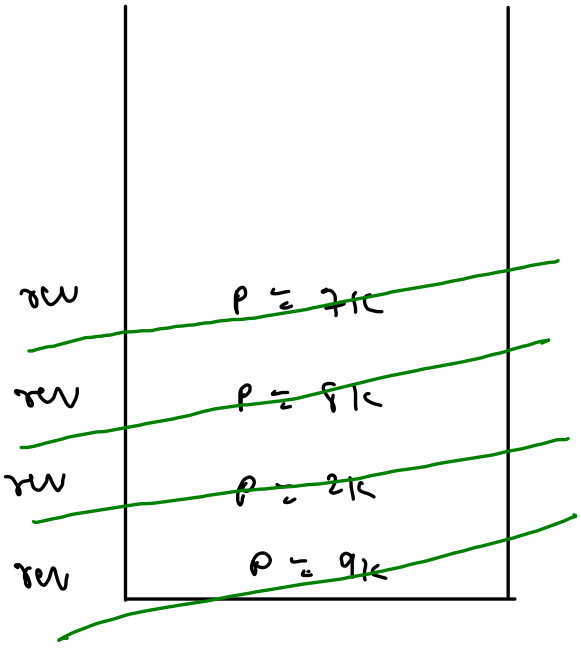
Way 2 : return

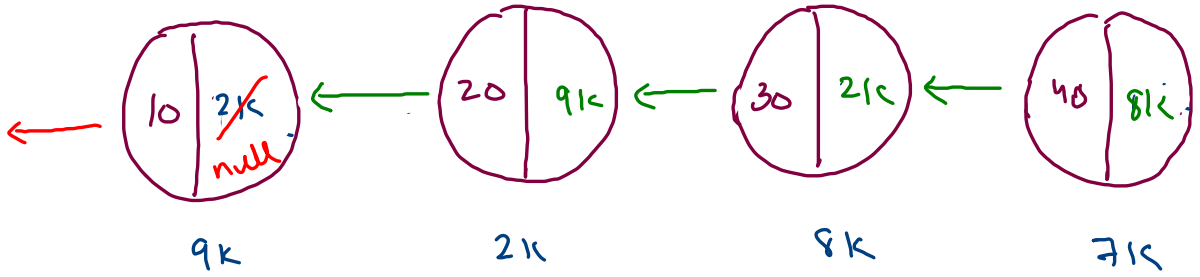



```
Node rev (Node rn, int dw){
    if (rn == null) {
        return head;
    }
    Node dn = rev (rn.next, dw+1);
    if (dw >= size/2) {
        swap dn, rn data;
    }
    return dn.next;
}
```

h = 4k
t = 9k
s = 4

Reverse Linked List (pointer - Recursive)

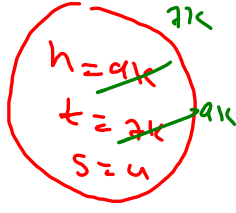




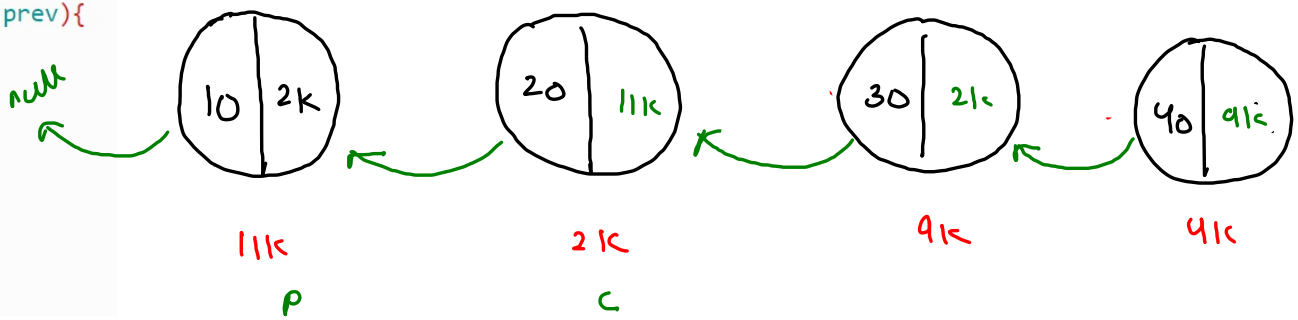
c = p.next

c.next = p

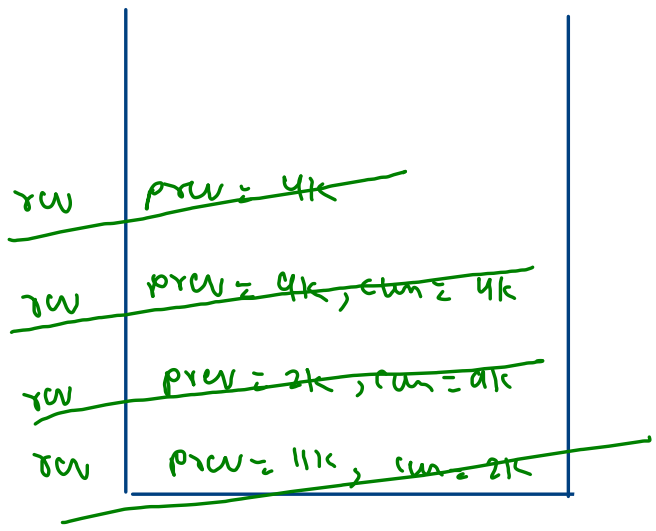
head.next = null

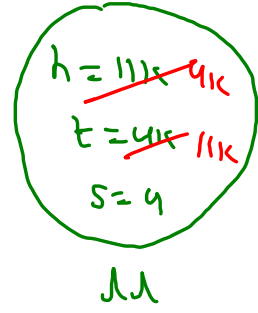


```
private void reversePRHelper(Node prev){
    if(prev.next == null) {
        return;
    }

    reversePRHelper(prev.next);

    Node curr = prev.next;
    curr.next = prev;
}
```




```
public void reversePR(){
  // write your code here
  reversePRHelper(head);
  head.next = null;

  //swap head & tail
  Node temp = head;
  head = tail;
  tail = temp;
}
```





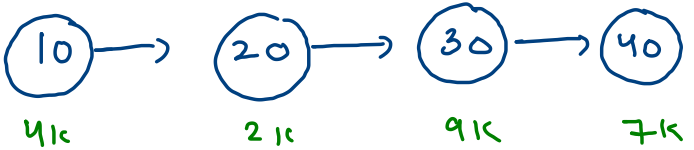
Reverse analysis:

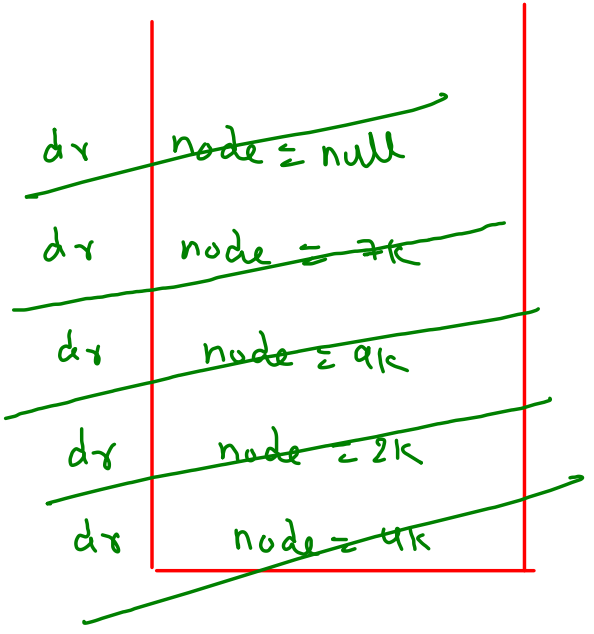
(best)

	data iterative	pointer iterative	data recursive	pointer recursive
T	$O(n^2)$	$O(n)$	$O(n)$	$O(n)$
S	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Display Reverse (recursive) - Linked List

```java
private void displayReverseHelper(Node node){
    if(node == null) {
        return;
    }

    displayReverseHelper(node.next);
    System.out.print(node.data + " ");
}
```





40 30 20 10

Static (independent)

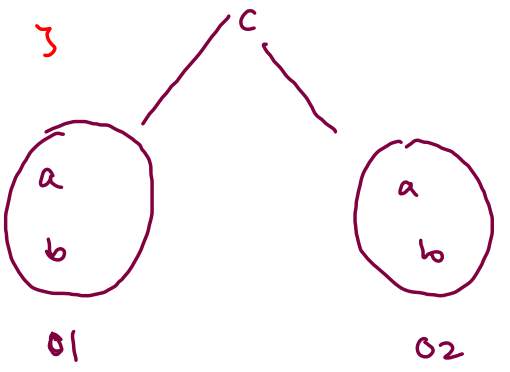
non-static (object bound)

```
class A {
    int a;
    int b;
    static int c;
}
```

data member

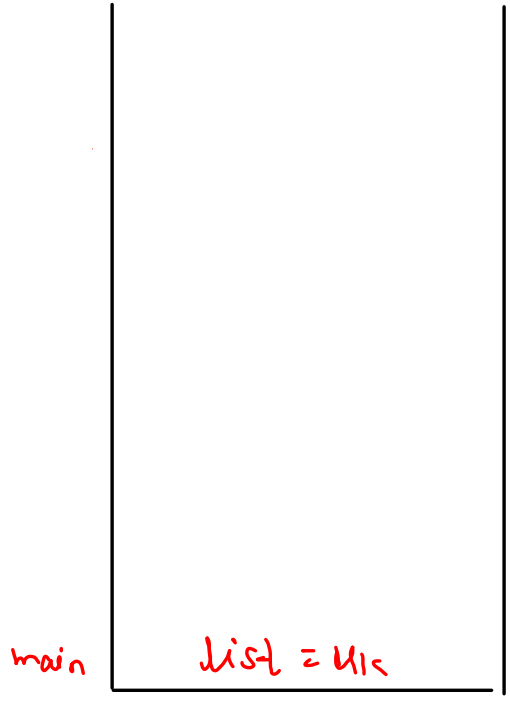
class variable

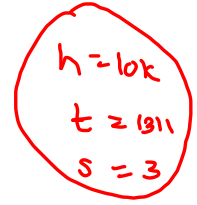
```
main() {
    A o1 = new A();
    A o2 = new A();
}
```

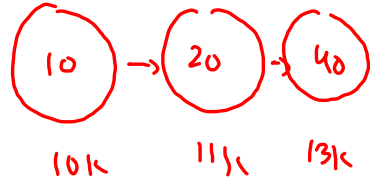


```
LL {

    Node head;
    Node tail;
    int size;

    void addLast() {
                    ---> non-static
    }

    LL add2LL(L1, L2) {
                    ---> static
    }
}
```

```
list.addLast()

LL res = LL.add2LL(list1, list);
```



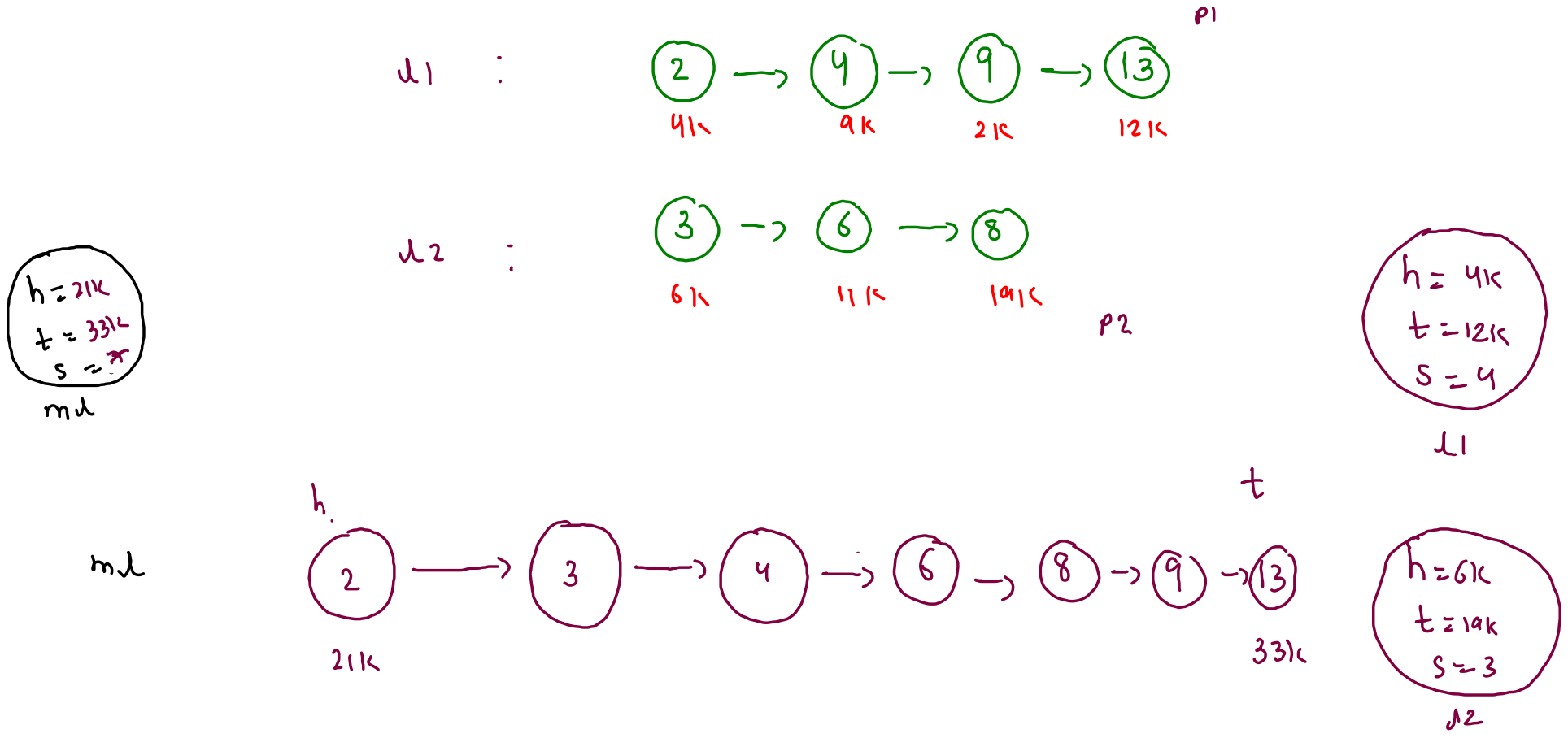




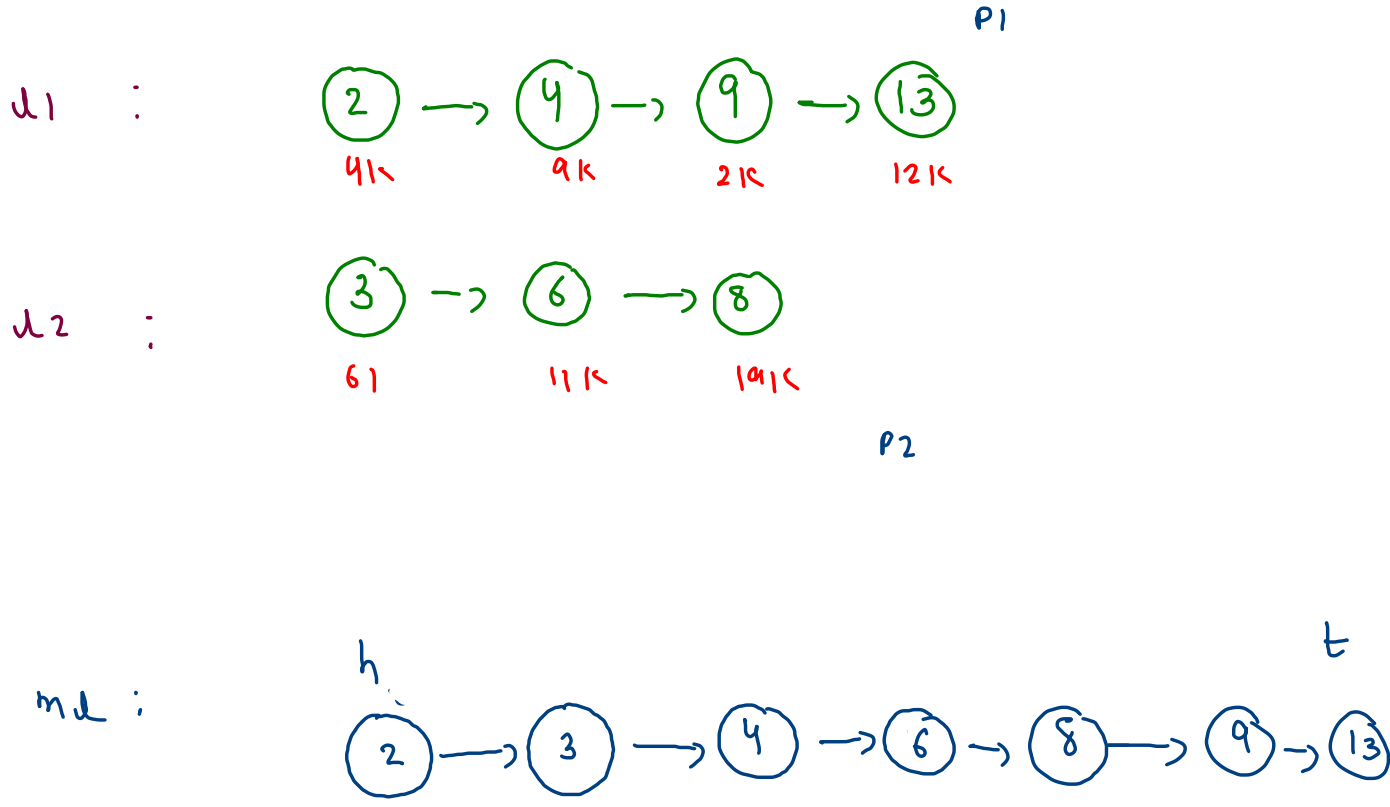
```
LL {
    non static fun3( ) { }

    non static fun1( ) {
        fun3( );    // this.fun3( );
    }

    static fun2( ) {

    }

}
```

```
main ( ) {

    LL list = new LL( );

    list. fun1( );

}
```

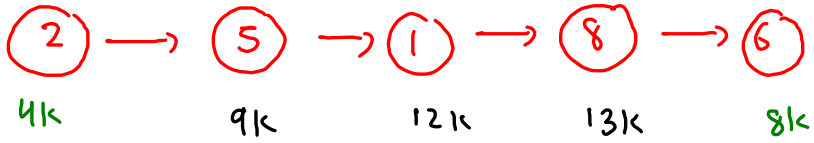
Merge Two Sorted Linked Lists

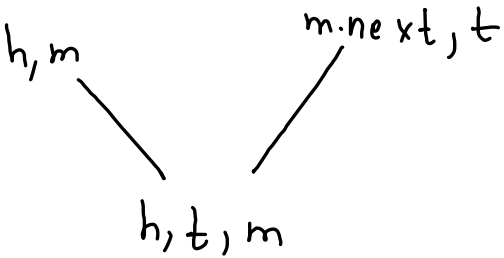


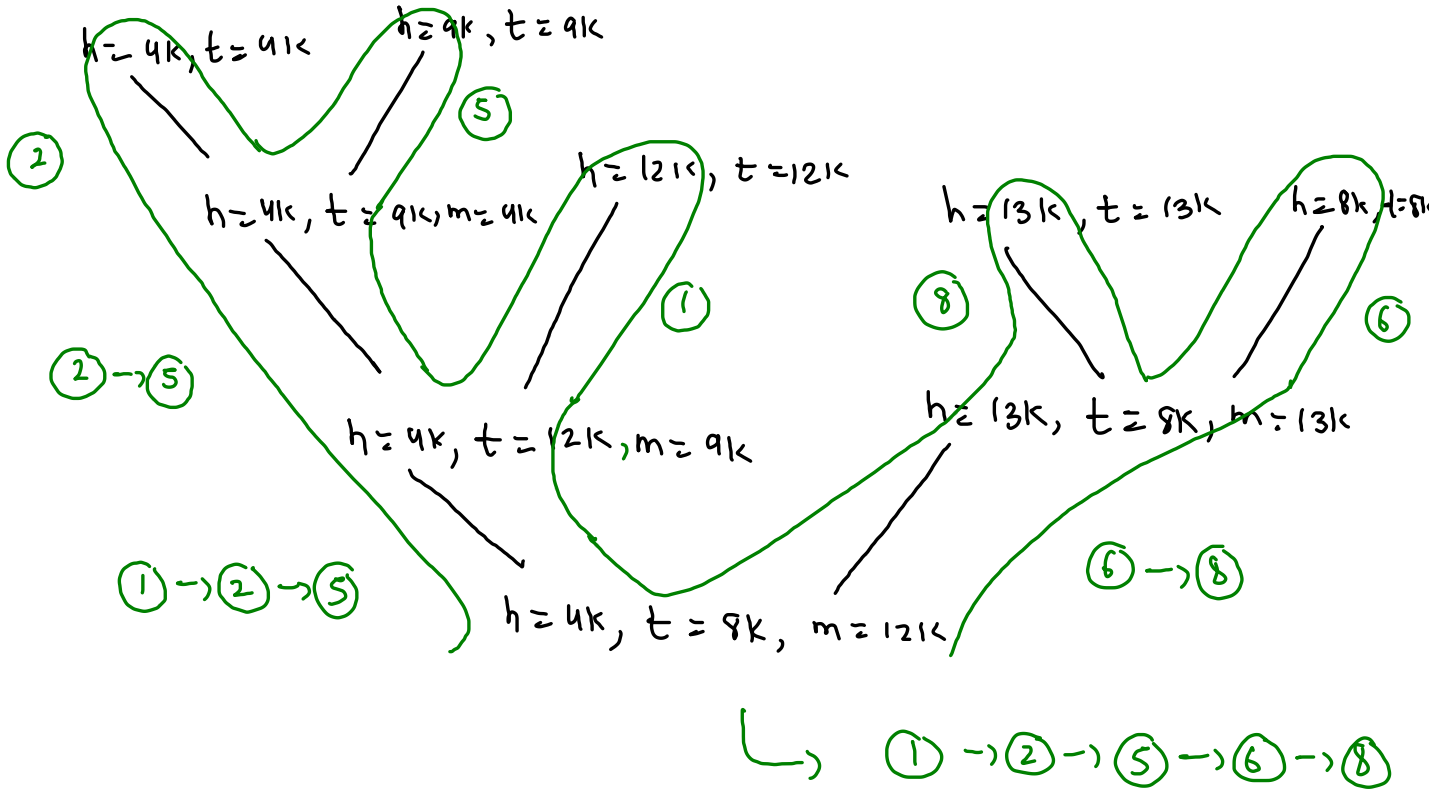
```java
LinkedList ml = new LinkedList();

Node p1 = l1.head;
Node p2 = l2.head;

while(p1 != null && p2 != null) {
    if(p1.data < p2.data) {
        ml.addLast(p1.data);
        p1 = p1.next;
    }
    else {
        ml.addLast(p2.data);
        p2 = p2.next;
    }
}

while(p1 != null) {
    ml.addLast(p1.data);
    p1 = p1.next;
}

while(p2 != null) {
    ml.addLast(p2.data);
    p2 = p2.next;
}

return ml;
```



Merge Sort A Linked List







T : nlogn

```
public static Node midNode(Node head,Node tail) {
    Node slow = head;
    Node fast = head;

    while(fast != tail && fast.next != tail) {
        slow = slow.next;
        fast = fast.next.next;
    }

    return slow;

}
```

```
public static LinkedList mergeSort(Node head, Node tail){
    if(head == tail) {
        LinkedList bl = new LinkedList();
        bl.addLast(head.data);
        return bl;
    }

    Node mid = midNode(head,tail);

    LinkedList left = mergeSort(head,mid);
    LinkedList right = mergeSort(mid.next,tail);

    LinkedList ans = mergeTwoSortedLists(left,right);
    return ans;
}
```