

Recursion

(8) Reverse a Linked List (V.V.Imp)

Consider an example bigger problem

`reverse([5 | 2004] → [9 | 2009] → [2 | 2013] → [3 | NULL])`

→ solution

new-head = 2013

Equivalent smaller prb

`reverse([5 | NULL] ← [9 | 2000] ← [2 | 2004] ← [3 | 2009])`

new-head = 2009

`reverse([9 | 2009] → [2 | 2013] → [3 | NULL])`

new-head = 2013

`reverse([9 | NULL] ← [2 | 2004] ← [3 | 2009])`

new-head = 2009

now think what you would do
on sol of equi smaller prb to
get sol of bigger prb

`reverse([5 | NULL] ← [9 | NULL] ← [2 | 2004] ← [3 | 2009])`

new-head = 2009

i.e. we will remove null and instead of null add ten address where head is pointing currently.

`reverse([3 | NULL])`

new-head = 2013

sol to equivalent smallest prb

→ $O(n)$ time complexity, $O(n)$ space comp
(recursive stack)

→ because we will return newhead
of reversed linked list which is a
pointer to
node datatype

```
node* reverseRecursive(node* &head){  
    base condition  
  
    if( head->next==NULL){  
        return head; } → newhead is  
        head itself  
  
    node* newhead= reverseRecursive(head->next);  
    head->next->next=head;  
    head->next=NULL;  
  
    return newhead;  
}  
  
int main()  
{  
    node* head=NULL;  
    insertAtTail(head,1);  
    insertAtTail(head,2);  
    insertAtTail(head,3);  
    insertAtTail(head,4);  
  
    display(head);  
    → node* newhead= reverse(head);  
    → display(newhead);  
  
    return 0;  
}
```

then code

(Q) Reverse k nodes in a linked list (V.V.V.Imp)

Input
Given, $LL \rightarrow$

$K=2$

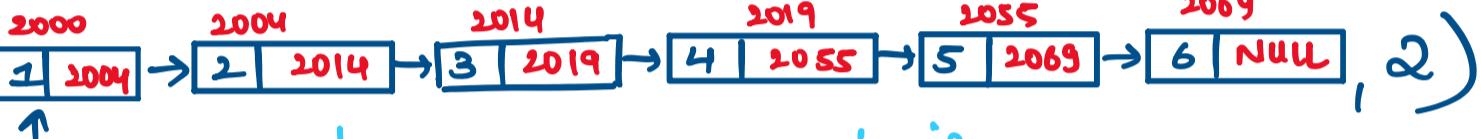
Output

Question

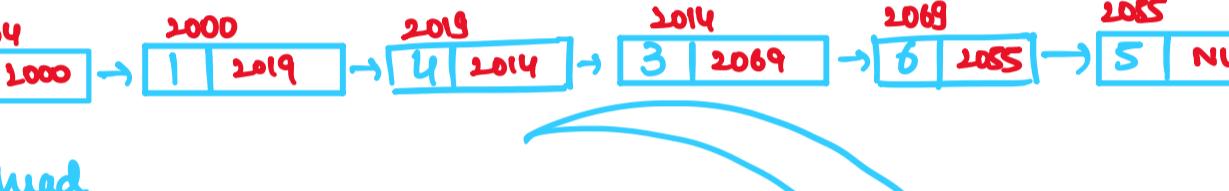
→ (n) time complexity

consider an example

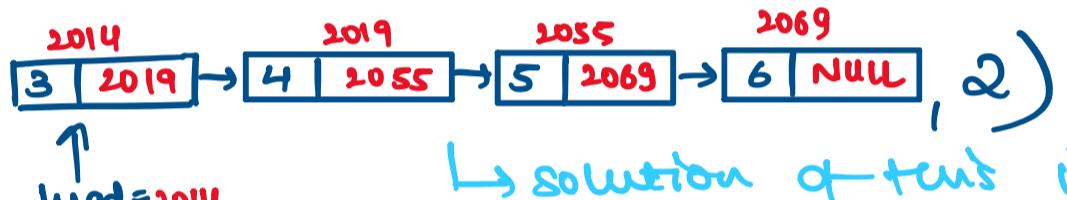
bigger prob

reverse k ( , 2)

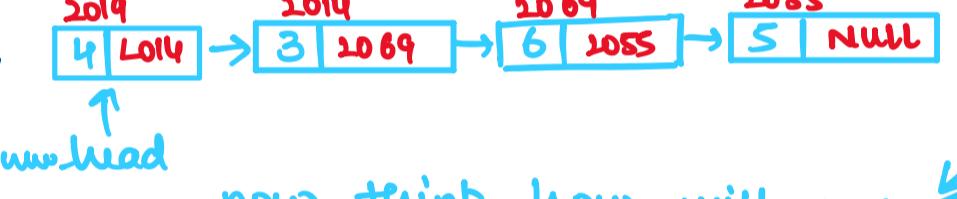
↳ solution of this is



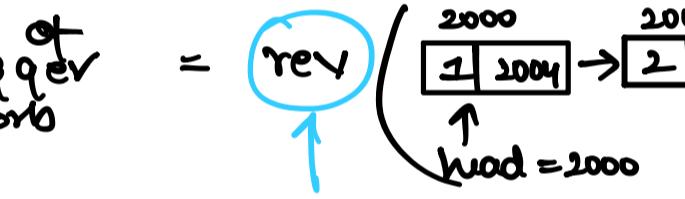
equivalent smaller prob

reverse k ( , 2)

↳ solution of this is

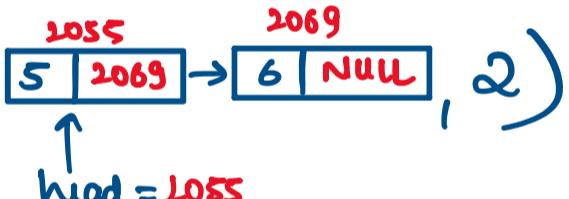


now think how will you get solution of bigger problem from sol of equivalent smaller problem!!

sol of bigger prob = rev () + sol of equi smaller prob

you can use iterative or above recursive fn for doing this reverse.

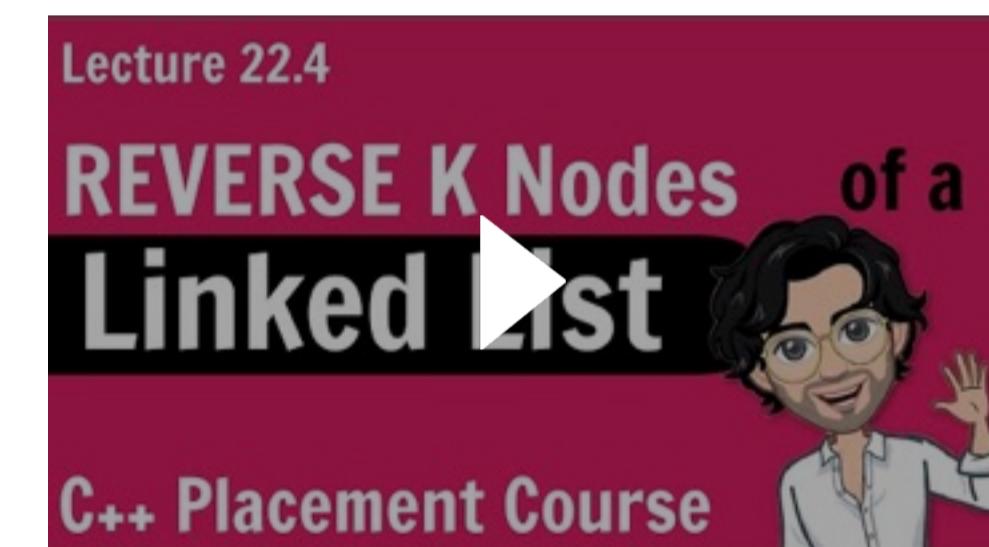
equivalent smaller prob

reverse k ( , 2)

reverse k (, , , ,) → equi smallest prob

→ sol to equi smallest prob

[Reverse K Nodes in a Linked List | C++ Placement Course | Lecture 22.4](#)



their code

(o) Sort a linked list using Merge sort

think write

similarly logic, what we saw in array,
usually interviewers never ask this
question.

all sorting techniques we saw
in array can be applied here

- 1. Bubble Sort $O(n^2)$
 - 2. Insertion Sort $O(n^2)$
 - 3. Selection Sort $O(n^2)$
 - 4. Heap Sort $O(n \log n)$
 - 5. Merge Sort $O(n \log n)$
 - 6. Quick Sort $O(n \log n)$
 - 7. Shell Sort $O(n^{3/2})$
 - 8. Count Sort $O(n)$
 - 9. Bucket / Bin Sort $O(n)$
 - 10. Radix Sort $O(n)$

→ comparison based sort algos

→ index based sort algos.

→ why quick sort is preferred to arrays over linked list??

- Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires $O(N)$ extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm.
- Comparing average complexity we find that both type of sorts have $O(N \log N)$ average complexity but the constants differ. For arrays, merge sort loses due to the use of extra $O(N)$ storage space.
- Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of $O(n \log n)$. The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

- In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory.
- Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time if we are given reference/pointer to the previous node. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
- In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of $A[0]$ be x then to access $A[i]$, we can directly access the memory at $(x + i * 4)$. Unlike arrays, we can not do random access in linked list.
- Quick Sort requires a lot of this kind of access. In linked list to access i 'th index, we have to travel each and every node from the head to i 'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort is more efficient and the need of random access is low.

- Insertion sort is more friendly to linked list compared to array cause in linked list to insert an element we need not shift element, we we just need to attach a node.
 - Similarly merge sort is also more friendly to linked list compared to array.