Linked list'

Thursday, 3 August 2023 6:32 AM

① Reverse a Linked List in groups of size k

Reversing first k Nodes

```
curr = head
prev, next = None, None
count = 1
while curr != None and count < k:
    next = curr.next
    curr.next = prev
    prev = curr
    curr = next
    count += 1
```

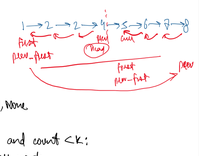
Recursive — Time: $O(n)$, Aux: $O(n/k)$

```
def reverseK(head, k):
    curr = head
    prev, next = None, None
    count = 0
    while curr != None and count < k:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
        count += 1
    if next != None:
        rem_head = reverseK(curr, k)
        head.next = rem_head
    return prev
```

1. Reverse first sublist of size k. While reversing keep track of next node and previous node. Let pointer of next node be next and previous node be prev.
2. head.next = reverse(next, k)
   Recursively call for rest of LL and link two sublists.
3. Return prev (prev becomes new head of list)

Iterative

```
def iterK(head, k):
    curr = head
    prev_first = None
    first_pass = True
    while curr != None:
        first, prev = curr, None
        count = 0
        while curr != None and count < k:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next
            count += 1
        if (first_pass):
            head = prev
            first_pass = False
        else:
            prev_first.next = prev
        prev_first = first
        return head
```



① Reverse a Linked List

— Iterative

use three pointers curr, prev and next to keep track of nodes to update

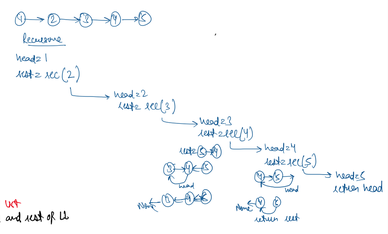
```
def reverseList(head):
    prev = None
    curr = head
    while curr:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    return prev
```

— Recursion

rest is last node of linked list using recursion then start reversing the linked list

- provide list with two pads: first node and rest of LL
- Call reverse for rest of LL
- link rest of LL to first
- fix head pointer to null

```
def reverseList(self, head):
    if not head or not head.next:
        return head
    rest = self.reverseList(head.next)
    head.next.next = head
    head.next = None
    return rest
```



② Detect Loop

Method 1 (Naive)

Run a loop from head. For every node x, do the following:

a) Let y be the next of x
b) If y is null, return false
c) Else run a loop from head to x.
   If any node is found to be same as y, return true.

Time: $O(n^2)$, Space: $O(1)$

Method 2 ($O(n)$ time but modifies linked list node structure)

→ We add a boolean field "visited" in the node. We initialize "visited" as "false" in the Node's constructor.
→ We traverse the list and mark "visited" of the visited nodes as "true"
   If it is false.
→ If already true, we return true.
→ If we reach null, we return false

Method 3 (modifies linked list) — $O(n)$: Time

1) Create a dummy Node
2) Traverse the list and do following for each node x:
   a) If next of x is not dummy, make it dummy
   b) Else if next of x is None, return False
   c) Else return True

Method 4 (Using Hash Set) — $O(n)$: Time, $O(n)$: Space

1) Create an empty set.
2) Traverse through the list and do following for every node:
   a) If x is already in the set, return True
   b) Else put x in the set
3) Return False

— Idea is to insert nodes in hashmap and whenever node is encountered that is already present, return true

```
s = set(), curr = head
while curr:
    if curr in s: return True
    s.add(curr)
    curr = curr.next
return false
```

③ Detect loop using Floyd's cycle detection algorithm

Algorithm

① Initialize slow_p = head, fast_p = head
② Move slow_p by one and fast_p by two. If these point meet, then there is a loop.

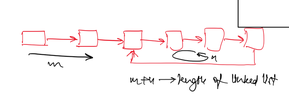
```
def isLoop(head):
    slow_p = head
    fast_p = head
    while fast_p != None and fast_p.next != None:
        slow_p = slow_p.next
        fast_p = fast_p.next.next
        if slow_p == fast_p:
            return True
    return false
```

How does this algorithm work?

Points to Note:

→ fast_p will enter into the loop before (or at same time as slow_p).
→ let fast_p be k distance ahead of slow_p, when slow_p enters the loop where $k \geq 0$.
→ this distance keeps on increasing by one in every movement of both pointers.
→ when distance becomes length of cycle, they meet.

Time: $O(n+m)$, Space: $O(1)$



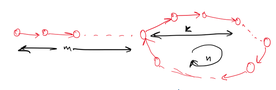
④ Detect and remove loop in linked list

① → Detect loop using floyd's detection algorithm.
② → Move "slow_p" to the beginning of linked list and keep "fast_p" at meeting point.
③ → Now one by one move slow and fast (at same speed).
The point where they meet now is the first node of the loop.

```
def detectRemoveLoop(head):
    slow = head
    fast = head
    while fast != None and fast.next != None:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            break
    if slow != fast:
        return
    slow = head
    while slow != fast.next:
        slow = slow.next
        fast = fast.next
    fast.next = None
```

(Detect loop / Remove loop)

How does this algo work?



Before the first meeting point:

- (Distance travelled by slow) * 2 = (Distance travelled by fast)

$(m + k + x * n) * 2 = m + k + y * n$

$\Rightarrow m + k = n(y - 2x)$

$\Rightarrow (m+k)$ is a multiple of n

x → No. of iterations made by slow before the first meeting point
y → " " " " fast

If (m+k) is a multiple of n, then second meeting point is going to be first node of loop.

Variation:
① Find length of loop.
② Find the first node of loop.

```
def removeLoop(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next
        if slow == fast:
            break
    if slow != fast:
        return
    if slow == head:
        while slow.next != fast:
            slow = slow.next
        slow.next = None
    elif slow == fast:
        slow = head
        while slow.next != fast.next:
            slow = slow.next
            fast = fast.next
        fast.next = None
    return head
```

(* if slow is head)

Find first node of loop in linked list

```
def findFirstNode(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            break
    if slow != fast:
        break
    slow = head
    while slow != fast:
        slow = slow.next
        fast = fast.next
    return slow.data
```

⑤ Intersection Point of two linked lists

Method 1

```
def intersection(head1, head2):
    s = set
    curr = head1
    while curr != None:
        s.add(curr)
        curr = curr.next
    curr = head2
    while curr != None:
        if curr. in s:
            return curr.data
        curr = curr.next
    return -1
```

— using two-pointer approach
Time: $O(m+n)$
Space: $O(m)$
where m is length of 1st linked list and n " " " 2nd " "

Method 2

```
def intersection(d, head1, head2):
    curr1 = head1
    curr2 = head2
    for i in range(d):
        if curr1 == None:
            return -1
    while curr1 != None and curr2 != None:
        if curr1 == curr2:
            return curr1.data
        curr1 = curr1.next
        curr2 = curr2.next
```

Time: $O(m+n)$, Space: $O(1)$

① Intersection of two LL
— using dummy node

```
dummy = curr2 = Node(-1)
while head1 and head2:
    if head1.data == head2.data:
        curr.next = Node(head1.data)
        head1 = head1.next
        head2 = head2.next
        curr = curr.next
    elif head1.data < head2.data:
        head1 = head1.next
    else:
        head2 = head2.next
return dummy.next
```

⑥ Segregate Even and Odd Nodes

Naive (Requires two traversals) — $\Theta(n)$ time / $\Theta(1)$ space

① Find the last node reference/pointer by doing a traversal.
② Traverse the linked list again. For every odd node, insert it after the last node and make the newly inserted node as the new node.

Efficient (Idea for the one traversal solution)

→ We maintain 4 variables:
- es → reference/pointer to start of the even sublist
- ee → " " " end " " " "
- os → " " " start " " odd "
- oe → " " " end " " " "

All the above are initialized as null/None.

While traversing:
(a) If current node is even, we insert it after ee and update ee. Also update es if it first node.
(b) Similar to (a) for odd nodes.

After the loop, connect the two lists.

```
def segregate(head):
    es, ee, os, oe = None, None, None, None
    curr = head
    while curr != None:
        x = curr.data
        if x % 2 == 0:
            if es == None:
                es = curr
                ee = es
            else:
                ee.next = curr
                ee = ee.next
        else:
            if os == None:
                os = curr
                oe = os
            else:
                oe.next = curr
                oe = oe.next
        curr = curr.next
    if os == None or es == None:
        return head
    ee.next = os
    oe.next = None
    return es.
```

⑦ Pairwise Swap Nodes

Naive (Swapping data) — $\Theta(n)$ time

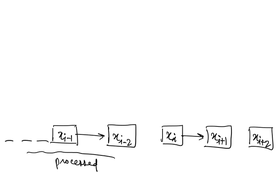
→ Run a loop while we have at least one node ahead:
   ① swap data of current node with the next node
   ② move current two nodes ahead

(* Swapping data is not good idea because there can be "data structures of objects".)

```
def pairWiseSwap(head):
    curr = head
    while curr != None and curr.next != None:
        curr.data, curr.next.data = curr.next.data, curr.data
        curr = curr.next.next
    return head
```

Efficient (Change links and don't touch data)

```
def pairWiseSwap(head):
    if head == None or head.next == None:
        return head
    curr = head.next.next
    prev = head
    head = head.next
    head.next = prev
    while curr != None and curr.next != None:
        prev.next = curr.next
        prev = curr
        next = curr.next.next
        curr.next.next = curr
        curr = next
    prev.next = curr
    return head
```



⑧ Clone a linked list with random connections

Method 1 (Use Dictionary) — $\Theta(n)$ time / $\Theta(n)$ space

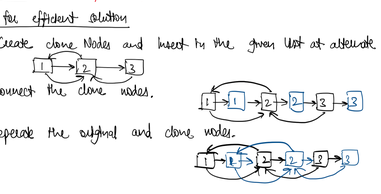
① Create an empty dictionary d
② Traverse through the list and do the following for every node:
   d[curr] = Node(curr.data)
③ Traverse through the list again and do:
   d[curr].next = d[curr.next]
   d[curr].random = d[curr.random]
④ Return d[h1]

```
def clone(h1):
    d = {None: None}
    curr = head
    while curr != None:
        d[curr] = Node(curr.data)
        curr = curr.next
    curr = head
    while curr != None:
        d[curr].next = d[curr.next]
        d[curr].random = d[curr.random]
        curr = curr.next
    return d[h1]
```

Method 2 — $\Theta(n)$ time / $\Theta(1)$ space

Idea for efficient solution

① Create clone Nodes and insert in the given list at alternate positions.
② Connect the clone nodes.
③ Seperate the original and clone nodes.



```
def clone(h1):
    curr = h1
    while curr != None:
        next = curr.next
        curr.next = Node(curr.data)
        curr.next.next = next
        curr = next
    curr = h1
    while curr != None:
        curr.next.random = None if curr.rab == None else curr.rab.next
        curr = curr.next.next
    h2 = h1.next
    clone = h2
    curr = h1
    while curr != None:
        curr.next = curr.next.next
        clone.next = None if (clone.next == None) else clone.next.next
        clone = clone.next
        curr = curr.next
    return h2
```

⑨ Merge two sorted linked lists

```
def SortedMerge(a, b):
    if (a == None):
        return b
    if (b == None):
        return a
    head, tail = None, None
    if a.data <= b.data:
        head = tail = a
        a = a.next
    else:
        head = tail = b
        b = b.next
    while (a != None and b != None):
        if (a.data <= b.data):
            tail.next = a
            tail = a
            a = a.next
        else:
            tail.next = b
            tail = b
            b = b.next
    if (a == None):
        tail.next = b
    else:
        tail.next = a
    return head
```

Time: $\Theta(m+n)$, Space: $\Theta(1)$

Remove duplicates from a sorted list

```
def removeDuplicates(head):
    curr = head
    while curr.next:
        if curr.data == curr.next.data:
            curr.next = curr.next.next
        else:
            curr = curr.next
    return head
```

(* Remove duplicates from unsorted LL — using set)

```
def removeDuplicates(head):
    curr = head
    count = set()
    count.add(curr.data)
    while curr.next:
        if curr.next.data in count:
            curr.next = curr.next.next
        else:
            count.add(curr.next.data)
            curr = curr.next
    return head
```

Merge Sort for linked list

```
def merge(self, left, right):
    dummy = curr = Node(-1)
    while left and right:
        if left.data <= right.data:
            curr.next = left
            left = left.next
            curr = curr.next
        else:
            curr.next = right
            right = right.next
            curr = curr.next
    if left or right:
        if left:
            curr.next = left
        else:
            curr.next = right
    return dummy.next
```

```
def mergeSort(self, head):
    if not head or not head.next:
        return head
    mid = findMid(head)
    head2 = mid.next
    mid.next = None
    head = mergeSort(head)
    head2 = mergeSort(head2)
    return self.merge(head, head2)
```

```
def findMid(head):
    slow = head
    fast = head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

⑩ Palindrome linked list

Naive (Using stack) — Time: $\Theta(n)$, Space: $\Theta(n)$

```
def isPalindrome(head):
    stack = []
    curr = head
    while (curr != None):
        stack.append(curr.data)
        curr = curr.next
    curr = head
    while (curr != None):
        if (stack.pop() != curr.data):
            return false
        curr = curr.next
    return True
```

Efficient

```
def isPalindrome(head):
    if (head == None):
        return True
    slow, fast = head, head.next
    while (fast.next != None and fast.next.next != None):
        slow = slow.next
        fast = fast.next.next
    rev = reverseList(slow.next)
    curr = head
    while (rev != head):
        if (rev.data != curr.data):
            return false
        rev = rev.next
        curr = curr.next
    return True
```

---

① Add 1 to a number represented as linked list
— Reverse a given linked list
— Start traversing it from leftmost and add 1 to it. If there is a carry move to next node. Keep moving to next while there is a carry.
— Reverse modified LL and return

```
def addOne(head):
    head = reverse(head)
    k = head
    carry = 0
    prev = None
    head.data += 1
    while head and (head.data > 9 or carry > 0):
        prev = head
        head.data += carry
        carry = head.data // 10
        head.data = head.data % 10
        head = head.next
    if carry:
        prev.next = Node(carry)
    return reverse(k)
```

② Add two numbers represented by LL
— without extra space

```
def add(first, second):
    head = first = reverse(first)
    second = reverse(second)
    carry = 0
    total = 0
    while first and second:
        total = carry + first.data + second.data
        carry = total // 10
        first.data = total % 10
        prev = first
        first = first.next
        second = second.next
        total = 0
    if first or second:
        if second:
            prev.next = second
        first = prev.next
        while first:
            total = first.data + carry
            first.data = total % 10
            carry = total // 10
            prev = first
            first = first.next
    if carry:
        prev.next = Node(carry)
    return (reverse(head))
```

③ Check if LL is circular
— store head of LL and traverse it. If iterator reaches NULL, LL is not circular else it reaches head.

④ Split circular into two halves

⑤ Deletion and Reversal of a circular LL

⑥ Reverse a doubly LL
— traverse LL using a pointer
— swap prev and next for all nodes

```
def reverseDLL(head):
    curr = head
    prev = None
    while curr:
        prev = curr
        curr.next, curr.prev = curr.prev, curr.next
        curr = curr.prev
    return prev
```

⑦ Find pairs with given sum in DLL (sorted DLL)
— loop terminates when two pointers cross each other (second.next = first) or they become same (first = second)

```
def findPairs(target, head):
    first, second = head, head
    ans = []
    while second.next:
        second = second.next
    while first != second and first.prev != second:
        if first.data + second.data == target:
            ans.append([first.data, second.data])
            first = first.next
            second = second.prev
        elif first.data + second.data < target:
            first = first.next
        else:
            second = second.prev
    return ans
```

⑧ Count triplets in a sorted DLL whose sum is equal to x
→ $O(n^2)$ (two pointer approach)

⑨ Sort a K sorted DLL
→ Using min-Heap
Same as K sorted LL

```
class Node:
    def __init__(val):
        self.val = val
        self.next = None
    def __lt__(self, other):
        return self.val < other.val
from heap import heapq, heappush, heappop

def mergeKlists(lists):
    def solve(lists, k):
        heap = []
        for i in range(k):
            if lists[i]:
                heappush(heap, (lists[i].val, lists[i]))
        dummy = Node(0)
        last = dummy
        while heap:
            curr = heappop(heap)[1]
            last.next = curr
            last = last.next
            if curr.next:
                heappush(heap, (curr.next.val, curr.next))
        return dummy.next
    return solve(lists, len(lists))
```

⑩ Rotate a linked list by k places
— traverse list from start and stop at k^{th} node. Store k's next in a temp pointer and point k's next to NULL then start traversing from temp till end, and point end's node next to start.
— Make temp as new head.

→ traverse given list to find last element and store it in a node.
— Make last element of current's head which we can do by storing head in temp node. Repeat k times.