hiimdaosui Follow
Jul 19, 2018 · 5 min read · Listen

Remove Linked List Elements

> *Remove all elements from a linked list of integers that have value* **val**.
>
> ***Example:***

```
Input:  1->2->6->3->4->5->6, val = 6
Output: 1->2->3->4->5
```

This problem asks us to remove all nodes in a singly linked list whose val is equal to another input `val`. This turns out to be a very straightforward question. We will also propose both iterative and recursive solutions here.

- **Iterative Solution**

To solve this problem iteratively, we can simply traverse the list once, extracting the node whose value is different from `val` and delete others.

Since this is quite obvious, there are only two things that I think are worth noticing is that we'd better use a dummy head as a holder to hold the filtered valid nodes. It's also mainly because the head might be deleted, so this makes our implementation more convenient.

Second, at the end of traversal, make sure that the last node in our result list is pointing to `NULL` to indicate the end of the list. What problem would occur if we don't do that? Imagine the case where we need to remove the last element in the input list. Then it will

random address in the memory. And it could be very risky and dangerous. So make sure to set it empty.

```
ListNode* removeElements(ListNode* head, int val) {
    ListNode dummy(0), *it = &dummy;
    while (head) {
        if (head->val == val) {
            ListNode* toDel = head;
            head = head->next;
            delete toDel;
        } else {
            it->next = head;
            head = head->next;
            it = it->next;
        }
    }
    it->next = NULL;

    return dummy.next;
}
```

Also, note that in C++, we need to delete the node and free the memory if they are not used any more. It's not a problem in Java or Python and etc., which have automatic garbage collection.

Let's go through an example whose last element needs to be removed:

```
Suppose that we are given a list:

1->2->1->3->1->NULL, val = 1

Initialization: dummy -> 1, it = dummy

Current List:       1->2->1->3->1->NULL
Result List:        dummy->NULL
1st Iteration:      head points to the first 1, remove
                    move head to 2
                    delete the first 1
```


```
                    move it to 2
                    move head to the second 1

Current List:       2->1->3->1->NULL
Result List:        dummy->2->(1->3->1)
1st Iteration:      head points to the second 1, remove
                    move head to 2
                    delete the second 1

Current List:       2->, 3->1
Result List:        dummy->2->
2nd Iteration:      head points to 3, keep
                    add 3 to result
                    move it to 3
                    move head to the last 1

Current List:       2->3->1
Result List:        dummy->2->3->(1)
1st Iteration:      head points to the last 1, remove
                    move head to NULL
                    delete the last 1

Current List:       2->3->
Result List:        dummy->2->3->

Post Process:       Point it (3) to NULL

Result List:        dummy->2->3->NULL

return dummy.next = 2
```

Time Complexity: O(n) — We need to go through the list for once, so the time complexity is apparently linear.

Space Complexity: O(1) — Only a dummy head and a few additional pointers are used in this implementation, so the space cost is constant.

- **Recursive Solution**

Still, we can try to use recursion to solve this problem since it could be easily broken down into multiple sub-problems. Note that the sub-problem is quite intuitive. We can simply

So the recursive rule is supposed to be:

1. Base Case: The input list is empty.
2. Recursive Rules: 1) Get the result from the sub-list starting from the next node. 2) Determine if the current should be kept or removed. If it is kept, then connect it with the result returned from the following list. Otherwise, discard the current node and return the subsequent result.

```
ListNode* removeElements(ListNode* head, int val) {
    if (!head) {
        return NULL;
    }
    ListNode* post = removeElements(head->next, val);
    if (head->val != val) {
        head->next = post;
        return head;
    }

    delete head;
    return post;
}
```

The code just straightly demonstrates the algorithm that we described above. `post` is the result we obtain from the subsequent list. It should contain all the nodes whose values are not equal to `val`. Also, note that if the current head's value is the same as `val`, remember to delete the current node to avoid memory leak.

Let's run the same example as the one in the iterative solution:

```
Suppose that we have an input list:

1->2->1->3->1->NULL

Current List:               1->2->1->3->1->NULL
In Main Function:           call removeElements(1)
```


```
Current List:               1->2->1->3->1->NULL
In removeElements(2):       call removeElements(1)

Current List:               1->2->1->3->1->NULL
In removeElements(1):       call removeElements(3)

Current List:               1->2->1->3->1->NULL
In removeElements(3):       call removeElements(1)

Current List:               1->2->1->3->1->NULL
In removeElements(1):       call removeElements(NULL)

Current List:               1->2->1->3->1->NULL
In removeElements(NULL):    hit base case
                            return NULL

Current List:               1->2->1->3->1->NULL
In removeElements(1):       get NULL
                            1 is equal to 1
                            delete 1
                            return NULL

Current List:               1->2->1->3->, NULL
In removeElements(3):       get NULL
                            3 is not equal to 1
                            point 3 to NULL
                            return 3

Current List:               1->2->1->, 3->NULL
In removeElements(1):       get 3
                            1 is equal to 1
                            delete 1
                            return 3

Current List:               1->2->, 3->NULL
In removeElements(2):       get 3
                            2 is not equal to 1
                            point 2 to 3
                            return 2

Current List:               1->2->3->NULL
In removeElements(1):       get 2
                            1 is equal to 1
                            delete 1
                            return 2
```

Time Complexity: O(n) — Same as the iterative solution, we need to go through the list for once, so the time complexity is apparently linear.

Space Complexity: O(n) —Every time we need to call the recursion for the next node, so for each node there will be an individual call to handle. So the cost of call stacks is linear. In each function, we only use a couple of local pointers, which are constant. So the entire space complexity is O(n).