Delete greater nodes in the right of linked list

2 approaches discussed here

GFG Link : Click

Leetcode Link : Click

$Approach-1$ (nsq approach) - TLE ⚠

Time : $O(n^2)$

> for because in worst case for n nodes we will check every n-1 nodes that if they are greater then the left nodes or not

Space : $O(1)$

> constant space

approach/steps :-

```
    /*⚠️✔️[TLE] Approach - 1 (simple O(nsq sol))

        explanation :

        // -> Main fun

         step 1 : create 3 pointers tempHead = head, temp = head -> next, tempHeadPrv
= null
         step 2 : run a loop while thempHead & temp both are non null
             step 3 : when the temp's data is smaller than tempHeads data ,
                 step 4 : when the tempHead and head are on same node
                     step 4.1 : head = head -> next
                     step 4.2 : create a node 'a' and store tempHead into it, then
delete 'a' node
                     step 4.3 : update tempHead = head
                     step 4.4 : temp = tempHead -> next
                 step 5 : else when the tempHead and heaad are not on same node
                     step 5.1 : create a node 'a' and store tempHead into it
                     step 5.2 : update tempHead = tempHead -> next
                     step 5.3 : update tempHeadPrv -> next = tempHead
                     step 5.4 : temp = tempHead -> next

            step 6 :  when the temps data is not smaller than tempHead's data
                 step 6.1 : set temp = temp -> next
                 step 6.2 : if tempHead = null then
                     step 6.3 : update tempHEadPrv = tempHead, tempHead = tempHead ->
next,
                     step 6.4 : if tempHead == null then break
                     step 6.5 : else temp = tempHead -> next
         step 7 : return the head node

        ⚠️T : O(N^2)  (worst case) -> n is total nodes in linked list
        ☑️S : O(1)


    */
```

code :-

```
public:
    Node *compute(Node *head)
    {
        // step 1 : create 3 pointers tempHead = head, temp = head -> next,
tempHeadPrv = null
        Node* tempHead = head;
        Node* temp = head -> next;
        Node* tempHeadPrv =nullptr;

        // step 2 : run a loop while thempHead & temp both are non null
        while(tempHead && temp){

            // step 3 : when the temp's data is smaller than tempHeads data ,
            if(temp != nullptr && (temp -> data > tempHead -> data)){

                // step 4 : when the tempHead and head are on same node
                if(tempHead == head){
                    // step 4.1 : head = head -> next
                    head = head -> next;
                    // step 4.2 : create a node 'a' and store tempHead into it, then
delete 'a' node
                    Node* a = tempHead;
                    a -> next = nullptr;
                    delete a;
                    // step 4.3 : update tempHead = head
                    tempHead = head;
                    // step 4.4 : temp = tempHead -> next
                    temp = tempHead -> next;
                }

                // step 5 : else when the tempHead and heaad are not on same node
                else{
                    // step 5.1 : create a node 'a' and store tempHead into it
                    Node* a = tempHead;
                    // step 5.2 : update tempHead = tempHead -> next
                    tempHead = tempHead -> next;
                    // step 5.3 : update tempHeadPrv -> next = tempHead
                    tempHeadPrv -> next = tempHead;
                    // step 5.4 : temp = tempHead -> next
                    temp = tempHead -> next;
                }
            }

            //step 6 :  when the temps data is not smaller than tempHead's data
            else{
                // step 6.1 : set temp = temp -> next
                temp = temp -> next;
                // step 6.2 : if tempHead = null then
                if(temp == nullptr){
                    // step 6.3 : update tempHEadPrv = tempHead, tempHead = tempHead -
```

```
> next,
                    tempHeadPrv = tempHead;
                    tempHead = tempHead -> next;
                    // step 6.4 : if tempHead == null then break
                    if(tempHead == nullptr) break;
                    // step 6.5 : else temp = tempHead -> next
                    else temp = tempHead -> next;
                }

            }
        }

        // step 7 : return the head node
        return head;
    }
```

★ $Approach - 2$ (using deque) - BEST

Time : $O(n)$

> n for inserting nodes into dq, then n for changing links to make a diffrt linked list

Space : $O(n)$

> to store n number of nodes in doubly ended queue

Approach :-

```
/* ✔⭐Approach - 2 (using deque) - Optimal

        explanation :-

            Fun.1 : main fun

                 step 91 : create a doubly ended q dq<node*>, create temp node to store head

                 step2 : run a loop while temp is not null
                     step 3 : if dq is empty, push temp node from back of dq and update the temp = temp -> next
                     step 4 : else if the dq is non  empty
                     then if the dqs top is greater then dq back then pop the front node from 'dq'
                     step 5 : else if the temps data is smaller then the dq's back node then push the temp into the dq, and set temp = temp -> next

                 step 6 : create 2 nodes 'mainHead = dq.front()' and 'tail = mainHead' , pop the front of dq
                 Step 7 : run a loop while the dq is not empty
                     step 7.1 : create a temp node and save the front of dq intp it, now pop fron front
                     step 7.2 : set tail -> next = temp, tail = temp
                 step 8 : return the mainHead


    */
```

Code :-

```
public:
    Node* compute(Node* head) {

        // step 1 : create a doubly ended q dq<node*>, create temp node to store head
        deque<Node*> dq;
        Node* temp = head;

        // step2 : run a loop while temp is not null
        while(temp){
            // step 3 : if dq is empty, push temp node from back of dq and update the
temp = temp -> next
            if(dq.empty()){
                dq.push_back(temp);
                temp = temp -> next;
            }

            // step 4 : else if the dq is non  empty
            // then if the dqs top is greater then dq back then pop the front node
from 'dq'
            else if(temp -> data > dq.back() -> data){
                dq.pop_back();
            }

            // step 5 : else if the temps data is smaller then the dq's back node then
push the temp into the dq, and set temp = temp -> next
            else{
                dq.push_back(temp);
                temp = temp -> next;
            }
        }

        // step 6 : create 2 nodes 'mainHead = dq.front()' and 'tail = mainHead' , pop
the front of dq
        Node* mainHead = dq.front();
        Node* tail = mainHead;
        dq.pop_front();

        // Step 7 : run a loop while the dq is not empty
        while(!dq.empty()){
            // step 7.1 : create a temp node and save the front of dq intp it, now pop
fron front
            Node* temp = dq.front();
            dq.pop_front();
            // step 7.2 : set tail -> next = temp, tail = temp
            tail -> next = temp;
            tail = temp;
        }

        // step 8 : return the mainHead
        return mainHead;
```

```
    }

};
```

$Approach - 3$ (recursive) - BEST

Time : $O(N)$

Space : $O(N)$

code :-

```
public:
    ListNode* removeNodes(ListNode* head) {

        // step1 : base case - if the head is null then return null
        if(head == nullptr) return nullptr;
        if(head -> next == nullptr) return head;

        // step 2 : recursion will somehow remove greater nodes for right linked list,
and will return the head of new right part, connect the head and right head
        ListNode* rightHead = removeNodes(head -> next);
        head -> next = rightHead;

        // step 3 : if right head is greater then 'head' then return the righthHead,
else return head
        if(rightHead -> val > head -> val) return rightHead;
        else return head;

    }
```