Flatten a linked list (with bottom & next ptr)

Date : 7th Jan 23

GFG Link : [Click here](Flattening a Linked List | Practice | GeeksforGeeks)

$Approach - 1$ (brute force)

Time : $O(NlogN)$

> O(n) for traversing and storing all nodes in 'ans' vector + O(nLogn) fir sorting the vector ans + O(n) for creating brand new linked lists from the ans vector

Space : $O(N)$

> for storing n nodes into the vector 'ans'

Approach Steps :

```
/* ✔️Approach - 1 (brute force)

            explanation :-

        -> // Fun.1 : main function

             step 1 : create a vector 'ans'
             step 2 : store the root nodes into a queue (FIFO)
             step 3 : run a loop while the queue is non empty
                 step 4 : fetch out the top node of queue and pop it from the queue
and store it in 'temp' node
                 step 5 : run a loop while temp is not null
                   step 5.1 : store the data of temp into 'ans' vector and set temp =
temp -> bottom
             step 6 : sort the ans vector
             step 7 : now create a whole new linked list from the vector 'ans' data
(make sure to insert data at the bottom of each node)
             step 8 : delete the dummy node & return the bottom node of head (because
head is dummy node with data -1)

    */
```

Code :

```
public:
    // Fun.1 : main function
    Node *flatten(Node *root)
    {
        // step 1 : create a vector 'ans'
        vector<int> ans;

        // step 2 : store the root nodes into a queue (FIFO)
        queue<Node*> q;
        Node* temp = root;
        while(temp){
            q.push(temp);
            temp = temp -> next;
        }

        // step 3 : run a loop while the queue is non empty
        while(!q.empty()){
            // step 4 : fetch out the top node of queue and pop it from the queue and
store it in 'temp' node
            temp = q.front();
            q.pop();

            // step 5 : run a loop while temp is not null
            while(temp){
                // step 5.1 : store the data of temp into 'ans' vector and set temp =
temp -> bottom
                ans.push_back(temp -> data);
                temp = temp -> bottom;
            }
        }

        // step 6 : sort the ans vector
        sort(ans.begin(), ans.end());

        // step 7 : now create a whole new linked list from the vector 'ans' data
(make sure to insert data at the bottom of each node)
        int size = ans.size();
        Node* head = new Node(-1);
        Node* tail = head;

        for(int i = 0; i < size; i++){
            Node* newNode = new Node(ans[i]);
            tail -> bottom = newNode;
            tail = newNode;
        }

        // step 8 : delete the dummy node & return the bottom node of head (because
head is dummy node with data -1)
        Node* dummyHead = head;
        Node* realHead = head -> bottom;
```

```
        dummyHead -> bottom = nullptr;
        delete dummyHead;

        return realHead;



    }
```

$Approach - 2$ (using recursion) - BEST

Time : $O(n)$

> n is total number of nodes in all linked lists

Space : $O(k)$

> k is total number of root nodes - for recursive call stack

idea :- here we will merge the right sorted linked lists using recursion, and then merge the left 1st list with right one and return the root

note : we start our merging of sorted linked lists from left to right but for that we will need to store the root nodes (k nodes)

Approach / steps :-

```
    explanation :

            step 1 :base case : if the root is null or the roots next is null then
return root, i.e cant merge empty or single linked list
            step 2 : reursion will merge the right linked lists and then return the
root after nerging so store it into 'rightRoot'
            step 3 : store the current root as 'leftRoot'
            step : merge the left and right linked lists with 2 pointer logic for
linked lists
            step 4 : create 2 points to nodes 'temp1' and 'temp2' and store the roots
of left and right roots into them
            step 5 : create a head node and a tail pointer with data -1
            step 5 : run a loop while the temp1 not null && temp2 is non null
                step 6 : check if temp1s data is lesser then temp2 them point it at
the bottom of tail & set temp1 = temp1 -> bottom
                step 7 : else repeat step6 for 'temp2'
            step 8 : if temp1 is non null them, attach temp1 at the tail's bottom and
set tails next as nullptr
            step 9 : else if temp2 is non null then attack temp2 at the bottom of
tail and set temp2's next as null
            step 10 : return the bottom node of head (because head is dummy head with
data -1)
```

code :

```
public:
    // Fun.1 : main function
    Node *flatten(Node *root)
    {
        //step 1 :base case : if the root is null or the roots next is null then return root, i.e cant merge empty or single linked list
        if(!root || !(root -> next)) return root;

        // step 2 : reursion will merge the right linked lists and then return the root after nerging so store it into 'rightRoot'
        Node* rightRoot = flatten(root -> next);

        // step 3 : store the current root as 'leftRoot'
        Node* leftRoot = root;

        // step : merge the left and right linked lists with 2 pointer logic for linked lists
        // step 4 : create 2 points to nodes 'temp1' and 'temp2' and store the roots of left and right roots into them
        Node* temp1 = leftRoot;
        Node* temp2 = rightRoot;

        // step 5 : create a head node and a tail pointer with data -1
        Node* head = new Node(-1);
        Node* tail = head;

        // step 5 : run a loop while the temp1 not null && temp2 is non null
        while(temp1 && temp2){
            // step 6 : check if temp1s data is lesser then temp2 them point it at the bottom of tail & set temp1 = temp1 -> bottom
            if(temp1 -> data < temp2 -> data){
                tail -> bottom = temp1;
                temp1 -> next = nullptr;
                tail = temp1;
                temp1 = temp1 -> bottom;
            }
            // step 7 : else repeat step6 for 'temp2'
            else{
                tail -> bottom = temp2;
                temp2 -> next = nullptr;
                tail = temp2;
                temp2 = temp2 -> bottom;
            }
        }

        // step 8 : if temp1 is non null them, attach temp1 at the tail's bottom and set tails next as nullptr
        if(temp1){
            tail -> bottom = temp1;
```

```
            temp1 -> next = nullptr;
        }

        // step 9 : else if temp2 is non null then attack temp2 at the bottom of tail
and set temp2's next as null
        if(temp2){
            tail -> bottom = temp2;
            temp2 -> next = nullptr;
        }

        // step 10 : return the bottom node of head (because head is dummy head with
data -1)
        return head -> bottom;
    }
```

END