LINKED LISTS - SET 2 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/convert-binary-number-in-a-linked-list-to-integer/
- **Difficulty level**: Easy

Explanation:

Initialize result number to be equal to head value: `num = head.val`. This operation is safe because the list is guaranteed to be non-empty.

Parse linked list starting from the head: `while head.next` :
The current value is `head.next.val`. Update the result by shifting it by one to the left and adding the current value using logical OR: `num = (num << 1) | head.next.val`.
Return `num`.

Solution:

```
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        int ret = 0;
        while(head){
            ret <<= 1;
            ret |= head->val;
            head = head->next;
        }
        return ret;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/delete-node-in-a-linked-list/
- **Difficulty level**: Easy

Explanation:

This problem forces you to think in a dereferencing sense. If you are, then it is a very silly question. Copy next node values into that node by dereferencing that's it guys.

Solution:

```
class Solution {
public:
    void deleteNode(ListNode* node) {
        // deleting node is not possible without pre pointer
        // use dereferencing concept
        *node=*(node->next);
    }
};
```

Complexity:

- Time: O(1)
- Extra space: O(1)

Question 3:

- **Problem link**: https://leetcode.com/problems/palindrome-linked-list/
- **Difficulty level**: Easy

Explanation:

Reach to the middle element using two pointers. Slow and fast where slow goes one by one and fast jump one node in between. When fast pointing to last or second last slow points to middle(odd case) or first middle(even case). Now reverse the next half list. Now compare the node values if all the values match then return true else false.

Solution:

```
ListNode *slow=head , *fast=head;

        // slow points to middle node after loop end
        while(fast->next and fast->next->next){
            slow=slow->next;
            fast=fast->next->next;
        }

        // reverse right half list
        ListNode *temp;
        fast=slow->next , slow=NULL;
        while(fast){
            temp=fast;
            fast=fast->next;
            temp->next=slow;
            slow=temp;
        }

        // compare the values
        temp=head;
        while(slow){
            if(slow->val!=temp->val)
```

```
                return false;

            slow=slow->next;
            temp=temp->next;
        }

        return true;
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 4:

- **Problem link**: https://leetcode.com/problems/reverse-linked-list-ii/
- **Difficulty level**: Medium

Explanation:

The detailed solution can be found here.

Solution:

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        // Empty list
        if (head == NULL) {
            return NULL;
        }
```

```
        // Move the two pointers until they reach the proper starting point
        // in the list.
        ListNode *cur = head, *prev = NULL;
        while (left > 1) {
            prev = cur;
            cur = cur->next;
            left--;
            right--;
        }

        // The two pointers that will fix the final connections.
        ListNode *con = prev, *tail = cur;

        // Iteratively reverse the nodes until n becomes 0.
        ListNode *third = NULL;
        while (right > 0) {
            third = cur->next;
            cur->next = prev;
            prev = cur;
            cur = third;
            right--;
        }

        // Adjust the final connections as explained in the algorithm
        if (con != NULL) {
            con->next = prev;
        } else {
            head = prev;
        }

        tail->next = cur;
        return head;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 5:

- **Problem link**: https://leetcode.com/problems/insertion-sort-list/
- **Difficulty level**: Medium

Explanation:

Keep a sorted partial list (head) and start from the second node (head -> next), each time when we see a node with value smaller than its previous node, we scan from the head and find the position that the node should be inserted. Since a node may be inserted before 'head', we create a dummy head that points to 'head'.

Solution:

```
class Solution {
public:
    ListNode* insertionSortList(ListNode* head) {
        ListNode* dummy = new ListNode(0);
        dummy -> next = head;
        ListNode *pre = dummy, *cur = head;
        while (cur) {
            if ((cur -> next) && (cur -> next -> val < cur -> val)) {
                while ((pre -> next) && (pre -> next -> val < cur -> next -> val))
                {
                    pre = pre -> next;
                }
                ListNode* temp = pre -> next;
                pre -> next = cur -> next;
                cur -> next = cur -> next -> next;
                pre -> next -> next = temp;
```

```
                pre = dummy;
            }
            else {
                cur = cur -> next;
            }
        }
        return dummy -> next;
    }
};
```

Complexity:

- Time: O(n^2)
- Space: O(1)

Question 6:

- **Problem link**: https://practice.geeksforgeeks.org/problems/merge-two-sorted-linked-lists/1
- **Difficulty level**: Medium

Explanation:

Merge is one of those recursive problems where the recursive solution code is much cleaner than the iterative code. In this problem, the base case includes one of the two linked lists being empty. Then we just return the non-empty linked list. Otherwise, initialize an empty linked list result, compare the values of current node of list a and current node of list b and assign the node which has lesser value to this new list (say a) and recursively repeat this process with next node of the lesser value current node list (i.e. a->next) and the current node of the other list (i.e. b) to get the next node of the result list.

Solution:

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;

    // Base cases
    if (a == NULL)
        return(b);
    else if (b == NULL)
        return(a);

    // Pick either a or b, and recur
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 7:

- **Problem link**: https://practice.geeksforgeeks.org/problems/swap-kth-node-from-beginning-and-kth-node-from-end-in-a-singly-linked-list/1
- **Difficulty level**: Medium

Explanation:

Find the k th node from the start and the kth node from last is n-k+1 th node from the start. Swap both the nodes.

However there are some corner cases, which must be handled

1. Y is next to X
2. X is next to Y
3. X and Y are same
4. X and Y don't exist (k is more than number of nodes in linked list)

Solution:

```
Node *swapkthnode(Node* head, int num, int K)
{
    // Count nodes in linked list
    int count = 0;
    Node* t = head;
    while (t != NULL) {
        count++;
        t = t->next;
    }
    int n = count;

    // Check if k is valid
    if (n < K)
        return head;

    // If x (kth node from start) and y(kth node from end) are same
```

```
    if (2 * K - 1 == n)
        return head;

    /* Find the kth node from the beginning of the linked list. We
also find the previous of kth node because we need to update the next
pointer of the previous.*/
    Node* x = head;
    Node* x_prev = NULL;
    for (int i = 1; i < K; i++) {
        x_prev = x;
        x = x->next;
    }

    /* Similarly, find the kth node from end and its previous. kth
node from end is (n-k+1)th node from beginning */
    Node* y = head;
    Node* y_prev = NULL;
    for (int i = 1; i < n - K + 1; i++) {
        y_prev = y;
        y = y->next;
    }

    /* If x_prev exists, then the new next of it will be y. Consider
the case when y->next is x, in this case, x_prev and y are the same.
So the statement "x_prev->next = y" creates a self loop. This self
loop will be broken when we change y->next. */
    if (x_prev)
        x_prev->next = y;

    // Same thing applies to y_prev
    if (y_prev)
        y_prev->next = x;

    /* Swap next pointers of x and y. These statements also break
self loop if x->next is y or y->next is x */
```

```
    Node* temp = x->next;
    x->next = y->next;
    y->next = temp;

    // Change head pointers when k is 1 or n
    if (K == 1)
        head = y;
    if (K == n)
        head = x;
    return head;
}
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 8:

- **Problem link**: https://leetcode.com/problems/merge-k-sorted-lists/
- **Difficulty level**: Hard

Explanation:

We Compare every k node (head of every linked list) and get the node with the smallest value and then we extend the final sorted linked list with the selected nodes. We use priority queue to optimize the comparison process.

Solution:

```
class Solution {
public:
    struct compare
```

```
    {
        bool operator()(ListNode* &a,ListNode* &b)
        {
            return a->val>b->val;
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*,vector<ListNode*>,compare>minh;
        for(int i=0;i<lists.size();i++)
        {
           if(lists[i]!=NULL) minh.push(lists[i]);
        }
        ListNode* head=new ListNode(0);
        ListNode* temp=head;
        while(minh.size()>0)
        {
            ListNode* p=minh.top();
            minh.pop();
            temp->next=new ListNode(p->val);
            temp=temp->next;
            if(p->next!=NULL) minh.push(p->next);
        }
        return head->next;
    }
};
```

Complexity:

- Time: O(n log k)
- Space: O(n)

Question 9:

- **Problem link**: https://practice.geeksforgeeks.org/problems/quicksort-on-doubly-linked-list/1/
- **Difficulty level**: Hard

Explanation:

Set the first node as a pivot element. You may decide any other element as pivot. places the pivot element at its correct position in sorted list, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot.

Solution:

```
Node* partition(Node *l, Node *h){
    //Your code goes here
    if(l==h) return l;
    // set pivot element
    int pivot = l->data;
    struct Node* ll=l->next;

    // sort the list with respect to pivot
    while( ll!=h->next){
        while(ll!=h->next and ll->data < pivot)
            ll=ll->next;
        while(ll!=h->next and h->data>= pivot)
            h=h-> prev;
        if(ll!=h->next)
            swap(&(ll->data) , &(h->data));
    }
    // place the pivot at its right position
    swap((&l->data) , (&h->data));
    return h;
}
```

Complexity:

- Time: O(nlog(n))
- Space: O(1)