Delete Nodes At Indices (Medium)

Given a linked list and an sorted array of integers as the indices in the list. Delete all the nodes at the indices in the original list.

Examples
1 -> 2 -> 3 -> 4 -> NULL, indices = {0, 3, 5}, after deletion the list is 2 -> 3 -> NULL.

Assumptions

- the given indices array is not null and it is guaranteed to contain non-negative integers sorted in ascending order.

```
// Delete an element at pos in the linked list
// if we delete the head node, count = 1, otherwise count = 0;
int deleteNode(ListNode** head, int pos){
    int count = 0;
    if(*head == NULL)
        return count;
    ListNode* temp = *head;
    if(pos == 0){
        *head = temp->next;
        free(temp);
        return count+1;
    }
    for(int i = 0; temp!= NULL && i < pos - 1; i ++)
        temp = temp->next;
    if(temp == NULL || temp->next == NULL)
        return count;
    ListNode* next = temp->next->next;
    free(temp->next);
    temp->next = next;
    return count;
}

// Delete nodes at indices using function deleteNode
void deleteNodes(ListNode** head, vector<int> indices){
    int count = 0;
    for(int i = 0; i < indices.size(); i++){
        count += deleteNode(head, indices[i] - count);
    }
}
```

Time Complexity: O(kn), k is the size of indices.

//Ok, good!

Reverse Linked List In Pairs (Medium)

Reverse pairs of elements in a singly-linked list.

Examples

- L = null, after reverse is null
- L = 1 -> null, after reverse is 1 -> null
- L = 1 -> 2 -> null, after reverse is 2 -> 1 -> null
- L = 1 -> 2 -> 3 -> null, after reverse is 2 -> 1 -> 3 -> null

```
void swap(int*a, int*b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Reverse pairs of elements in linked list
void reverseInPairs(ListNode* head){
    ListNode* temp = head;
    if(temp == NULL || temp->next == NULL)
        return;
    while(temp != NULL && temp->next != NULL){
        swap(temp->value, temp->next->value);//value or address swap(&temp->value,
&temp->next->value)
        temp = temp->next->next;
    }
}
```

Time Complexity: O(n), we only traverse the linked list once.

//I don’t know your ListNode struct, but value is usually a value not a pointer. So maybe you should use & in the swap function instead of handing value directly.

Insertion Sort Linked List (Medium)

Given a singly-linked list, where each node contains an integer value, sort it in ascending order. The insertion sort algorithm should be used to solve this problem.

Examples

- null, is sorted to null
- 1 -> null, is sorted to 1 -> null
- 1 -> 2 -> 3 -> null, is sorted to 1 -> 2 -> 3 -> null
- 4 -> 2 -> 6 -> -3 -> 5 -> null, is sorted to -3 -> 2 -> 4 -> 5 -> 6

```
// insert a node to the sorted linked list
void sortedInsert(ListNode** head, ListNode* newNode){
    if(*head == NULL || (*head)->value >= newNode->value){ // head node need to be 
changed
        newNode->next = *head;
        *head = newNode;
    }
    else {
        ListNode *pCurr = *head;
        while (pCurr->next != NULL && pCurr->next->value < newNode->value)
            pCurr = pCurr->next;
        newNode->next = pCurr->next;
        pCurr->next = newNode;
    }
}

// Sort a Linked List using Insertion Sort
void insertionSortLL(ListNode** head){
    ListNode* sorted = NULL;
    ListNode* pCurr = *head;
    while(pCurr != NULL){
        ListNode* pNext = pCurr->next;
        sortedInsert(&sorted, pCurr);
        pCurr = pNext;
    }
    *head = sorted;
}
```

Time Complexity: O(n^2), same as insertion sort in array.

//Ok, no problem!

Rotate List by K places (Medium)

Given a list, rotate the list to the right by k places, where k is non-negative.

Input: 1->2->3->4->5->NULL, k = 2
Output: 4->5->1->2->3->NULL
Input: 1->2->3->4->5->NULL, k = 12
Output: 4->5->1->2->3->NULL

```
// rotate the list to the right by k places
void rotateLL(ListNode** head, int k){
    int length = listLength(*head);
    if(k % length == 0)
        return;
    int trainsitionIdx = length - k % length - 1;
    ListNode* curr = *head;
    int count = 0;
    while(count < trainsitionIdx && curr != NULL){
        curr = curr->next;
        count++;
    }
    // store the node before new head node
    ListNode* transition = curr;
    // traverse to the end of list
    while(curr->next != NULL)
        curr = curr->next;
    curr->next = *head;
    *head = transition->next;
    transition->next = NULL;
}
```

Time Complexity: O(n), traverse the linked list less than two times.
//Ok, good!

Merge Sort Linked List (Medium)

Given a singly-linked list, where each node contains an integer value, sort it in ascending order. The merge sort algorithm should be used to solve this problem.

Examples

- null, is sorted to null
- 1 -> null, is sorted to 1 -> null
- 1 -> 2 -> 3 -> null, is sorted to 1 -> 2 -> 3 -> null
- 4 -> 2 -> 6 -> -3 -> 5 -> null, is sorted to -3 -> 2 -> 4 -> 5 -> 6

```
// Split the linked list into two
void splitLL(ListNode* head, ListNode** front, ListNode** back){
    if(head == NULL)
        return;
    ListNode* slow = head;
    ListNode* fast = head;
    while(fast->next != NULL && fast->next->next != NULL){
        slow = slow->next;
        fast = fast->next->next;
    }
    *front = head;
    *back = slow->next;
    slow->next = NULL;
}

// Merge two sorted linked list
ListNode* sortedMerge(ListNode* a, ListNode* b){
    ListNode* result = NULL;
    if(a == NULL)
        return b;
    else if(b == NULL)
        return a;
    if(a->value < b->value){
        result = a;
        result->next = sortedMerge(a->next, b);
    }
    else{
        result = b;
        result->next = sortedMerge(a, b->next);
    }
    return result;
}

// Sort the linked list using Merge Sort
void mergeSortLL(ListNode** head){
    ListNode* a;
    ListNode* b;
    if(*head == NULL || (*head)->next == NULL)
        return;
    splitLL(*head, &a, &b);
    mergeSortLL(&a);
    mergeSortLL(&b);
    *head = sortedMerge(a, b);
}
```

Time Complexity: O(nlogn), same as merge sort in array.
//Ok, excellent!

In-order Traversal Of Binary Tree (Medium)

Implement an iterative, in-order traversal of a given binary tree, return the list of keys of each node in the tree as it is in-order traversed.

Examples

```
    5
   / \
  3   8
 / \   \
1   4   11
```

In-order traversal is [1, 3, 4, 5, 8, 11]

Corner Cases

- What if the given binary tree is null? Return an empty list in this case.

How is the binary tree represented?
We use the level order traversal sequence with a special symbol "#" denoting the null node.

For Example:
The sequence [1, 2, 3, #, #, 4] represents the following binary tree:

```
  1
 / \
2   3
   /
  4
```

```
// iterative, in-order traversal of binary tree.
vector<char> inOrder(vector<char> binaryTree){
    vector<int> resultIdx;
    vector<char> result;
    stack<char> s;
    int n = binaryTree.size();
    char curr = 0;//int curr;
    while(binaryTree[curr] != '#' || !s.empty()){
        while(binaryTree[curr] != '#' && curr <= n - 1){
            s.push(curr);
            curr = 2 * curr + 1;
        }
        curr = s.top();
        s.pop();
        resultIdx.push_back(curr);
        curr = 2 * curr + 2;
    }
    for(int i = 0; i < resultIdx.size(); i++)
        result.push_back(binaryTree[resultIdx[i]]);
    return result;
}
```

Time Complexity: O(n), since we traverse the binary tree two times.
//The meothod is right, but I think the binary tree vector should not be vector<char>. Consider the input is [11, 12, 3, #, #, 14]. So it should be vector<string>, and you should convert the string to a integer.And the index should be int instead of char.
//Please modify it and submit in homework 7, thanks!

Minimum Depth of Binary Tree (Medium)

Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Example:
Given the below binary tree

```
        5
      /    \
    3       8
      \
        4
```

minimum depth is 2, path is 5→8.

```
// return minimum depth of binary tree
int minDepth(BTNode* root){
    if(root == NULL)
        return 0;
    if(root->left == NULL && root->right == NULL)
        return 1;
    if(root->left == NULL)
        return 1 + minDepth(root->right);
    if(root->right == NULL)
        return 1 + minDepth(root->left);
    return min(minDepth(root->left), minDepth(root->right)) + 1;
}
```

Time Complexity: O(n), since we traverse the binary tree once.
//Ok, great!