Q2

#include <stdio.h>

#include <stdlib.h>

struct \_listNode{

float item;

struct \_listNode\* next;

};

typedef struct \_listNode ListNode;

void printList(ListNode \*head);

void deleteList(ListNode \*\*ptrHead);

ListNode \*reverseSegment(ListNode\* head, int start, int end);

int main()

{

ListNode \*head = NULL, \*temp;

    float f = 0.0;

    int S, E;

    scanf("%d %d",&S, &E);

    while (scanf("%f", &f)){

        if (head == NULL){

            head = (ListNode\*) malloc(sizeof(ListNode));

            temp = head;

        }

        else{

            temp->next = (ListNode\*) malloc(sizeof(ListNode));

            temp = temp->next;

        }

        temp->item = f;

    }

    temp->next = NULL;

    head = reverseSegment(head,S,E);

    printList(head);

    deleteList(&head);

    return 0;

}

void printList(ListNode \*head){

while(head !=NULL){

printf("%.2f ",head->item);

head = head->next;

}

printf("\n");

}

void deleteList(ListNode \*\*ptrHead){

ListNode \*cur = \*ptrHead;

ListNode \*temp;

while (cur!= NULL) {

        temp=cur->next;

        free(cur);

        cur=temp;

    }

    \*ptrHead=NULL;

}

ListNode \*reverseSegment(ListNode\* head, int start, int end)

{

// Write your code here

// if starting and ending index is the same, no changes made

if(start == end)

return head;

    // Check size of linked list

int size\_check = 0;

ListNode \*temp = head;

while(temp != NULL)

{

size\_check++;

temp = temp->next;

}

    // If indices are out of range or invalid, linked list unchanged and returned

    // Start index should also not be larger than end index

    if(start > size\_check || end > size\_check || start>end)

    {

        return head;

    }

// rev\_head and rev\_tail is the head and tail of the part of linked list to be reversed

// rev\_end\_next is the next node after end of reverse list

//

ListNode \*rev\_head, \*rev\_tail, \*rev\_end\_next, \*rev\_prev, \*current;

// Initialize all to NULL first

rev\_head = rev\_prev = rev\_tail = rev\_end\_next = NULL;

current = head;

// count is use to keep track of position in the linked list

int count = 0;

// Extract out the partial linked list to be reversed

while(current && count <= end)

{

// If node not reached yet, initialize rev\_prev to current

if(count<start)

{

rev\_prev = current;

}

// When starting index node reached, initialize rev\_head to current

if(count==start)

{

rev\_head = current;

}

// When ending index node reached, intialize rev\_tail to current

// rev\_end\_next will be initialized to next node after end of reverse list

if(count==end)

{

rev\_tail = current;

rev\_end\_next = current->next;

}

// If start index not reached yet, proceed to next node and add to counter

current = current->next;

count++;

}

// Indicate end of partial linked list with NULL

rev\_tail->next = NULL;

// Reinitialize current to use it for reversing the partial linked list

current = rev\_head;

ListNode \*prev, \*next;

prev = next = NULL;

// Reverse the linked list using the iterative method

while(current != NULL)

{

// Save next node

next = current->next;

// Change next of current

current->next=prev;

// Shift prev one step ahead

prev=current;

// Shift current one step ahead

current=next;

}

// Head is tail of reverse now

rev\_tail = prev;

//head = rev\_head;

// Check if starting position was at head or not

if(rev\_prev)

{

rev\_prev->next = rev\_tail;

}

else

{

head=rev\_tail;

}

rev\_head->next = rev\_end\_next;

return head;

}

Q3

#include <stdio.h>

#include <stdlib.h>

struct \_listNode{

int item;

struct \_listNode\* next;

};

typedef struct \_listNode ListNode;

void printList(ListNode \*head);

void deleteList(ListNode \*\*ptrHead);

void triPartition(ListNode\*\* head, int pivot);

int main()

{

ListNode \*head = NULL, \*temp;

    int i = 0;

    int pivot = 0;

    scanf("%d",&pivot);

    while (scanf("%d", &i)){

        if (head == NULL){

            head = (ListNode\*) malloc(sizeof(ListNode));

            temp = head;

        }

        else{

            temp->next = (ListNode\*) malloc(sizeof(ListNode));

            temp = temp->next;

        }

        temp->item = i;

    }

    temp->next = NULL;

    triPartition(&head, pivot);

    printList(head);

    deleteList(&head);

    return 0;

}

void printList(ListNode \*head){

while(head !=NULL){

printf("%d ",head->item);

head = head->next;

}

printf("\n");

}

void deleteList(ListNode \*\*ptrHead){

ListNode \*cur = \*ptrHead;

ListNode \*temp;

while (cur!= NULL) {

        temp=cur->next;

        free(cur);

        cur=temp;

    }

    \*ptrHead=NULL;

}

void triPartition(ListNode\*\* head, int pivot)

{

// Create the head and tail of 3 linked list nodes for easy categorization

ListNode \*less\_head, \*equal\_head, \*more\_head, \*less\_tail, \*equal\_tail, \*more\_tail;

less\_head = equal\_head = more\_head = NULL;

// Iterate through the original linked list

while((\*head) != NULL)

{

// If current node == value of pivot, append the node's item into the equal node

if((\*head)->item == pivot)

{

if(equal\_head == NULL)

{

equal\_head = equal\_tail = (\*head);

}

else

{

equal\_tail->next = (\*head);

equal\_tail= equal\_tail->next;

}

}

// If current node is > value of pivot, append the node's item into the more node

else if((\*head)->item > pivot)

{

if(more\_head == NULL)

{

more\_head = more\_tail = (\*head);

}

else

{

more\_tail->next = (\*head);

more\_tail = (\*head);

}

}

// If current node < value of pivot, append the node's item into the less node

else if((\*head)->item < pivot)

{

if(less\_head == NULL)

{

less\_head = less\_tail = (\*head);

}

else

{

less\_tail->next = (\*head);

less\_tail = (\*head);

}

}

//next node of the original linked list

\*head = (\*head)->next;

}

// If the more list contains nodes, change tail of more linked list to NULL

if (more\_tail != NULL)

{

more\_tail->next = NULL;

}

// Check if less list is empty

if(less\_head == NULL)

{

// Check if equal list is empty

if(equal\_head == NULL)

{

// If less and equal list are empty, point original list to the more list

\*head = more\_head;

return;

}

// If equal list is not empty, point tail of equal list to head of more list

// Point original list to this new combined list

equal\_tail->next = more\_head;

\*head = equal\_tail;

return;

}

// Check if equal list is empty

if(equal\_head == NULL)

{

// If equal list is empty, point tail of less list to head of more list

// Point original list to this new combined list

less\_tail->next = more\_head;

\*head = less\_head;

return;

}

// If all 3 lists are not empty, point tail of less list to head of equal list

// Point tail of equal list to head of more list

// Point original list to this new combined list

less\_tail->next= equal\_head;

equal\_tail->next=more\_head;

\*head = less\_head;

return;

// Write your code here

}

Q4

#include <stdio.h>

#include <stdlib.h>

struct \_listNode{

int item;

struct \_listNode\* next;

};

typedef struct \_listNode ListNode;

void printList(ListNode \*head);

void deleteList(ListNode \*\*ptrHead);

void reverseKNodes(ListNode\*\* head, int K);

int main()

{

ListNode \*head = NULL, \*temp;

    int i = 0;

    int K = 0;

    scanf("%d",&K);

    while (scanf("%d", &i)){

        if (head == NULL){

            head = (ListNode\*) malloc(sizeof(ListNode));

            temp = head;

        }

        else{

            temp->next = (ListNode\*) malloc(sizeof(ListNode));

            temp = temp->next;

        }

        temp->item = i;

    }

    temp->next = NULL;

    reverseKNodes(&head, K);

    printList(head);

deleteList(&head);

    return 0;

}

void printList(ListNode \*head){

while(head !=NULL){

printf("%d ",head->item);

head = head->next;

}

printf("\n");

}

void deleteList(ListNode \*\*ptrHead){

ListNode \*cur = \*ptrHead;

ListNode \*temp;

while (cur!= NULL) {

        temp=cur->next;

        free(cur);

        cur=temp;

    }

    \*ptrHead=NULL;

}

void reverseKNodes(ListNode\*\* head, int K){

//Write your code here

// If reverse K node is 1 or 0 or linked list is empty, no changes

if(K ==1 || K == 0 || head== NULL)

{

return;

}

// Create dummy node, point dummy node to head of given linked list

ListNode \*dummy;

dummy = (ListNode\*) malloc(sizeof(ListNode));

dummy->next = (\*head);

(\*head) = dummy;

// next\_n refers to next node, to avoid confusion with next

ListNode \*current, \*next\_n, \*previous;

current = next\_n = previous = dummy;

// Counts length of linked list with count

int count = 0;

while(current->next !=NULL)

{

current = current->next;

count++;

}

while(count >= K)

{

current = previous->next;

next\_n = current->next;

// Perform reversing of k nodes

for(int i=1; i< K; i++)

{

current->next = next\_n->next;

next\_n->next = previous->next;

previous->next = next\_n;

next\_n = current->next;

}

previous = current;

count = count-K;

}

// Add remaining nodes<k that was previously ignored

// Add to end

while(current->next != NULL)

{

current = current->next;

}

// Remove dummy character

ListNode \*temp = (\*head);

(\*head) = (\*head)->next;

}