

In-Lab – 1(page No: 71)

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int value;
    struct node* next;
};
struct node* head;
struct node* node_new(int val)
{
    struct node* newnode = (struct node*)malloc(sizeof(struct node));
    newnode->value = val;
    newnode->next = NULL;
    return newnode;
}
void insertAtEnd(int val)
{
    struct node* newnode = node_new(val);
    if(head==NULL)
    {
        head=newnode;
    }
    else{
        struct node* current = head;
        while(current->next!=NULL)
        {
            current = current->next;
        }
        current->next = newnode;
    }
}
void print()
{
    struct node* temp = head;
    while(temp!=NULL){
        printf("%d ",temp->value);
        temp=temp->next;
    }
    printf("\n");
}
```

```

int main()
{
    head=NULL;
    int n;
    scanf("%d",&n);
    for(int i=0 ; i<n ; i++){
        int value;
        scanf("%d",&value);
        insertAtEnd(value);
    }
    print();
}

```

In-Lab – 2(page No: 73)

```

typedef struct Node NODE;
NODE *getMiddleElement(NODE *head)
{
    if (head == NULL)
    {
        return NULL; // Return NULL if the list is empty
    }
    struct Node *temp=head,*pretemp=head;
    while(temp !=NULL && temp->next != NULL)
    {
        pretemp=pretemp->next;
        temp=temp->next->next;
    }
    return pretemp;
}

```

In-Lab – 3(page No: 75)

```

void insertAfterK(LinkedList* list, int value, int k)
{
    Node* newNode = createNode(value);
    Node* current = list->head;
    if (current == NULL)
    {
        list->head = newNode;
        return;
    }
    for (int i = 1; i < k; i++) {
        current = current->next;
    }
    newNode->next=current->next;
    current->next=newNode;
}

```

Post-Lab – 1(page No: 77)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeNthFromEnd(struct ListNode* head, int n)
{
    struct ListNode *temp=head,*pre;
    int count=0,i=1;
    while(temp!=NULL)
    {
        count++;
        temp=temp->next;
    }
    if(count<=1) return NULL;
    if(count==n) return head->next;
    temp=head;
    for(i=1;i<(count-n);i++)
    {
        temp=temp->next;
    }
    temp->next=temp->next->next;
    return head;
}
```

Post-Lab – 2(page No:79)

```
// Definition for singly-linked list
/*struct ListNode {
    int val;
    struct ListNode* next;
};*/
struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2)
{
    struct ListNode temp; // Dummy node to simplify the result list management
    struct ListNode* current = &temp; // Pointer to build the result list
    temp.val = 0;
    temp.next = NULL;
    int carry = 0;
```

```

while (l1 != NULL || l2 != NULL || carry > 0)
{
    int sum = carry;
    if (l1 != NULL)
    {
        sum += l1->val;
        l1 = l1->next;
    }
    if (l2 != NULL)
    {
        sum += l2->val;
        l2 = l2->next;
    }
    carry = sum / 10;
    int digit = sum % 10;
    current->next = malloc(sizeof(struct ListNode));
    current = current->next;
    current->val = digit;
}
current->next = NULL;
return temp->next;
}

```

Skill Lab-1(page No:81)

```

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* deleteDuplicates(struct ListNode* head)
{
    if (head == NULL) return NULL;
    struct ListNode* current = head;
    while (current->next != NULL)
    {
        if (current->val == current->next->val)
        {
            struct ListNode* temp = current->next;
            current->next = current->next->next;
            free(temp);
        }
        else

```

```

        {
            current = current->next;
        }
    }
    return head;
}

```

Skill Lab-2(page No:83)

```

typedef struct LinkedList
{
    Node* head;
    Node *tail;
} LinkedList;

```

// Function to insert a node at the end of the linked list

```

void insertAtEnd(LinkedList* list, int value) {
    Node* newNode = createNode(value);

    // If there are no nodes in the linked list
    // Set the new node as head and tail
    if (list->head == NULL)
    {
        list->head = newNode;
        list->tail = newNode;
        return;
    }

    // Set next of tail to the new Node
    list->tail->next = newNode;
    // Set new Node as the new tail
    list->tail = newNode;
}

```

Skill Lab-3(page No:85)

/*struct node

```

{
    int data;
    struct node* next;
};*/

```

struct node* rearrange(struct node* head)

```

{
    struct node *evenDummyHead,*oddDummyHead;
    evenDummyHead=(struct node*)malloc(sizeof(struct node));

```

```

evenDummyHead->data=0;
evenDummyHead->next=NULL;
oddDummyHead=(struct node*)malloc(sizeof(struct node));
oddDummyHead->data=0;
oddDummyHead->next=NULL;
struct node* evenPtr = evenDummyHead; // Pointer to the even sublist
struct node* oddPtr = oddDummyHead;  // Pointer to the odd sublist
struct node* cur = head;
while (cur != NULL)
{
    if (cur->data % 2 == 0)
    {
        evenPtr->next = cur;
        evenPtr = evenPtr->next;
    }
    else
    {
        oddPtr->next = cur;
        oddPtr = oddPtr->next;
    }
    cur = cur->next;
}
evenPtr->next = oddDummyHead->next;
oddPtr->next = NULL;

struct node* rearrangedHead = evenDummyHead->next;

free(evenDummyHead);
free(oddDummyHead);

return rearrangedHead;
}

```

Skill Lab-4(page No:87)

```

// struct node{
//     int d;
//     struct node *link;
// };
// int i;
// typedef struct node NODE;
// NODE *start=NULL,*nn,*curr,*temp;
// NODE *insert(NODE *start,int m){
//     nn=(NODE*)malloc(sizeof(NODE));
//     nn->link=NULL;

```

```

// nn->d=m;
// if(start==NULL){
//     start=nn;
//     temp=nn;
// }
// else{
//     temp->link=nn;
//     temp=temp->link;
// }
// return start;
// }
NODE *reverseSegment(NODE *start,int l,int r)
{
    if (start == NULL || l == r) return start;
    NODE temp;
    temp.link = start;
    NODE *pretemp = &temp;

    for (int i = 1; i < l; i++)
    {
        pretemp = pretemp->link;
    }

    NODE *current = pretemp->link;
    NODE *next = NULL;

    for (int i = 0; i < r - l; i++)
    {
        next = current->link;
        current->link = next->link;
        next->link = pretemp->link;
        pretemp->link = next;
    }
    return temp.link;
}

```

Skill Lab-5(page No:89)

```

struct Node* merge(struct Node* left,struct Node* right)
{
    if (left == NULL) return right;
    if (right == NULL) return left;

    struct Node* result = NULL;

```

```

    if (left->val <= right->val)
    {
        result = left;
        result->next = merge(left->next, right);
    }
    else
    {
        result = right;
        result->next = merge(left, right->next);
    }
    return result;
}

struct Node* mergeSort(struct Node* head)
{
    if (head == NULL || head->next == NULL) return head;

    struct Node *slow = head;
    struct Node *fast = head->next;

    while (fast != NULL && fast->next != NULL)
    {
        slow = slow->next;
        fast = fast->next->next;
    }

    struct Node *mid = slow->next;
    slow->next = NULL;

    struct Node* left = mergeSort(head);
    struct Node* right = mergeSort(mid);

    return merge(left, right);
}

struct Node* rearrange(struct Node* head)
{
    return mergeSort(head);
}

```

Skill Lab-6(page No:91)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2)
{
    struct ListNode* temp = (struct ListNode*)malloc(sizeof(struct ListNode));
    temp->val = 0;
    temp->next = NULL;
    struct ListNode* current = temp;
    while (list1 != NULL && list2 != NULL)
    {
        if (list1->val <= list2->val)
        {
            current->next = list1;
            list1 = list1->next;
        }
        else
        {
            current->next = list2;
            list2 = list2->next;
        }
        current = current->next;
    }
    if (list1 != NULL)
    {
        current->next = list1;
    }
    else
    {
        current->next = list2;
    }
    return temp->next;
}
```