In-Lab – 1(page No: 93)

```
// struct Node {
//     int data;
//     struct Node *next;
//     struct Node *prev;
// };
// struct Node *start;
void insertAtAnyPosition(int position)
{
  struct Node *newnode=malloc(sizeof(struct Node));
  if(newnode == NULL) return ;
  scanf("%d", &newnode->data);
  if(position < 1) return ;
   if(position == 1)
   {
       newnode->next=start;
       newnode->prev=NULL;
       if(start != NULL)
       {
           start->prev=newnode;
       }
       start=newnode;
   }
   else
   {
       struct Node *temp=start;
       for(int i=1;i<position-1 && temp != NULL;i++)
       {
           temp=temp->next;
       }
       if(temp == NULL)
       {
           free(newnode);
           return ;
       }
       else
       {
           newnode->next=temp->next;
           newnode->prev=temp;
           if(temp->next != NULL)
           {
               temp->next->prev=newnode;
           }
           temp->next=newnode;
```

```
        }
    }
}
void deleteAtAnyPosition(int position)
{
    if(start == NULL) return ;
    struct Node *temp=start;
    struct Node *pretemp=NULL;
    if(position == 1)
    {
        start=start->next;
        if(start != NULL)
        {
            start->prev=NULL;
        }
        free(temp);
    }
    else
    {
        for(int i=1;i<position && temp != NULL;i++)
        {
            pretemp=temp;
            temp=temp->next;
        }
        if(temp == NULL) return ;
        pretemp->next=temp->next;
        if(temp->next != NULL)
        {
            temp->next->prev=pretemp;
        }
        free(temp);
    }
}
```

In-Lab – 2(page No: 95)

```
/*
 * For your reference:
 *
 * DoublyLinkedListNode {
 *     int data;
 *     DoublyLinkedListNode* next;
 *     DoublyLinkedListNode* prev;
 * };
  */
```

```
DoublyLinkedListNode* reverse(DoublyLinkedListNode* llist)
{
   struct DoublyLinkedListNode *head=llist;
   if(head==NULL || head->next==NULL)
      return head;
   else
   {
      DoublyLinkedListNode *q=NULL,*p=NULL,*tmpe=head;
      while(tmpe->next!=NULL)
      {
         p=tmpe->next;
         tmpe->next=q;
         tmpe->prev=p;
         q=tmpe;
         tmpe=p;
      }
      tmpe->prev=NULL;
      tmpe->next=q;
      return tmpe;
   }
}
```

In-Lab – 3(page No: 97)

```
// struct Node{
//     int data;
//     struct Node * next;
// };

typedef struct Node NODE;
NODE *detectCycle(NODE *head)
{
   if(head == NULL || head->next ==NULL)
   return NULL;
   NODE *pretemp=head,*temp=head;
   while(temp != NULL && temp->next != NULL)
   {
      pretemp=pretemp->next;
      temp=temp->next->next;
      if(pretemp == temp)
      {
         break;
      }
   }
   if(temp == NULL || temp->next == NULL)
```

```
        return NULL;
    pretemp=head;
    while(pretemp != temp)
    {
        pretemp=pretemp->next;
        temp=temp->next;
    }
    return pretemp;
}
```

Post-Lab – 1(page No: 99)

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* doubleIt(struct ListNode* head)
{
    struct ListNode* temp1 = head;
    struct ListNode* temp2 = head->next;
    head->val = head->val*2;
    if(head->val >= 10)
    {
        struct ListNode* new_node = (struct ListNode*)(malloc(sizeof(struct ListNode)));
        new_node->next = head;
        head->val = head->val % 10;
        new_node->val = 1;
        head = new_node;
    }
    while(temp2 != NULL)
    {
        temp2->val = temp2->val*2;
        temp1->val = temp1->val + (temp2->val/10);
        temp2->val = temp2->val % 10;
        temp1=temp1->next;
        temp2=temp2->next;
    }
    return(head);
}
```

Post-Lab – 2(page No:101)

```
typedef struct Node
{
   int val;
   struct Node* next;
} Node;
typedef struct
{
   Node* head;
   int size;
} MyLinkedList;
MyLinkedList* myLinkedListCreate()
{
   MyLinkedList* newnode = (MyLinkedList*)malloc(sizeof(MyLinkedList));
   newnode->head=NULL;
   newnode->size=0;
   return newnode;
}
int myLinkedListGet(MyLinkedList *t, int index)
{
   if (index < 0 || index >= t->size)
   {
      return -1;
   }
   Node* current = t->head;
   for (int i = 0; i < index; i++)
   {
      current = current->next;
   }
   return current->val;
}
void myLinkedListAddAtHead(MyLinkedList *t, int val)
{
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->val = val;
   newNode->next = t->head;
   t->head = newNode;
   t->size++;
}
void myLinkedListAddAtTail(MyLinkedList *t , int val)
{
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->val = val;
   newNode->next = NULL;
```

```
    if (t->head == NULL)
    {
        t->head = newNode;
    }
    else
    {
        Node* current = t->head;
        while (current->next != NULL)
        {
            current = current->next;
        }
        current->next = newNode;
    }
    t->size++;
}
void myLinkedListAddAtIndex(MyLinkedList *t, int index, int val)
{
    if (index < 0 || index > t->size)
    {
        return;
    }
    if (index == 0)
    {
        myLinkedListAddAtHead(t, val);
    }
    else
    {
        Node* newNode = (Node*)malloc(sizeof(Node));
        newNode->val = val;
        Node *temp = t->head;
        for (int i = 0; i < index - 1; i++)
        {
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
        t->size++;
    }
}
void myLinkedListDeleteAtIndex(MyLinkedList *t, int index)
{
    if (index < 0 || index >= t->size) {
        return; // Invalid index
    }
    Node* toDelete;
    if (index == 0)
    {
        toDelete = t->head;
        t->head = t->head->next;
    }
    else
    {
        Node* temp = t->head;
        for (int i = 0; i < index - 1; i++)
        {
            temp = temp->next;
        }
        toDelete = temp->next;
        temp->next = toDelete->next;
    }
    free(toDelete);
    t->size--;
}
void myLinkedListFree(MyLinkedList *t)
{
    Node* temp = t->head;
    while (temp != NULL)
    {
        Node *next = temp->next;
        free(temp);
        temp = next;
    }
    free(t);
}
```

Skill Lab-1(page No:103)

```
// struct ListNode {
//     int val;
//     struct ListNode *next;
// };

int solve(struct ListNode* head)
{
   struct ListNode *slow = head;
    struct ListNode *fast = head;
    int count=0;
    while (fast != NULL && fast->next != NULL)
```

```
 {
  slow = slow->next;
  fast = fast->next->next;
  if (slow == fast)
  {
   count=1;
   struct ListNode *temp=slow;
   while(temp->next != slow)
   {
    count++;
    temp=temp->next;
   }
   return count;
  }
 }
 return -1;
}
```

Skill Lab-2(page No:105)

```
// Definition for singly-linked list.
// struct ListNode {
//     int val;
//     struct ListNode *next;
// };

int countCriticalPoints(struct ListNode *head)
{
 struct ListNode *pre, *current, *post;
 pre=head;
 current=pre->next;
 post=current->next;
 int count=0;
 if(pre->next==NULL)
  return 0;
 else
 {
  while (pre! =NULL && post! =NULL)
  {
   if((current->val < pre->val && current->val < post->val) ||
   (current->val > pre->val && current->val > post->val))
   {
    count++;
    pre=current;
    current=pre->next;
```

```
                post=current->next;
            }
            else
            {
                pre=current;
                current=pre->next;
                post=current->next;
            }
        }
    return count;
    }
}
```

Skill Lab-3(page No:107)

```
#include<stdio.h>
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

void Max(struct Node* head)
{
    if (head == NULL)
    {
        return;
    }

    struct Node* maxNode = head;
    struct Node* temp = head->next;
    while (temp != NULL)
    {
        if (temp->data > maxNode->data)
        {
            maxNode = temp;
        }
        temp = temp->next;
    }
    if (maxNode->prev != NULL)
    {
        printf("%d\n", maxNode->prev->data);
    }
    else
    {
```

```
    printf("%d", maxNode->data);
    }
}
int main()
{
    int n;
    scanf("%d",&n);
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    head->data = n;
    head->next = NULL;
    head->prev = NULL;
    Max(head);
    return 0;
}
```

Skill Lab-4(page No:109)

```
DoublyLinkedListNode* sortedInsert(DoublyLinkedListNode *head , int data)
{
    DoublyLinkedListNode *temp=head,*temp1;
 DoublyLinkedListNode *newnode=(DoublyLinkedListNode*)malloc(sizeof
(DoublyLinkedListNode));
    newnode->data=data;
    newnode->prev=NULL;
    newnode->next=NULL;
    if(head->data > data)
    {
        head->prev=newnode;
        newnode->next=head;
        head=newnode;
        return head;
    }
    while(temp !=NULL)
    {
        if(temp->data > data)
        {
            temp1->next=newnode;
            newnode->prev=temp1;
            temp->prev=newnode;
            newnode->next=temp;
            return head;
        }
        temp1=temp;
        temp=temp->next;
    }
    temp1->next=newnode;
    newnode->prev=temp1;
    return head;
}
```

Skill Lab-5(page No:111)

```
int findmax(int x,int y)
 {
    return (x>y)?x:y;
 }
int pairSum(struct ListNode *head)
{
    int sum=0;
    struct ListNode *pre = NULL;
    struct ListNode *current = head;
    struct ListNode *currHalf = head;
    while (currHalf != NULL && currHalf->next != NULL)
    {
        currHalf = currHalf->next->next;
        struct ListNode* temp = current->next;
        current->next = pre;
        pre = current;
        current = temp;
    }
    while(current != NULL)
    {
        sum = findmax(sum,(current->val + pre->val));
        current = current->next;
        pre = pre->next;
    }
    return sum;
}
```

Skill Lab-6(page No:113)

```
void swap(int *x, int *y)
 {
    int t=*x;
    *x=*y;
    *y=t;
 }
struct ListNode* swapNodes(struct ListNode* head, int k)
{
    struct ListNode *pre=NULL,*post=NULL,*temp=head;
    int length=0,index=0;
    while(temp != NULL)
    {
        temp=temp->next;
        length++;
    }
    temp=head;
    while(temp != NULL)
    {
        if(index == k-1)
            pre=temp;
        if(index == length-k)
            post=temp;
        if(pre != NULL && post != NULL)
            break;
        temp=temp->next;
        index++;
    }
    swap(&pre->val,&post->val);
    return head;
}
```