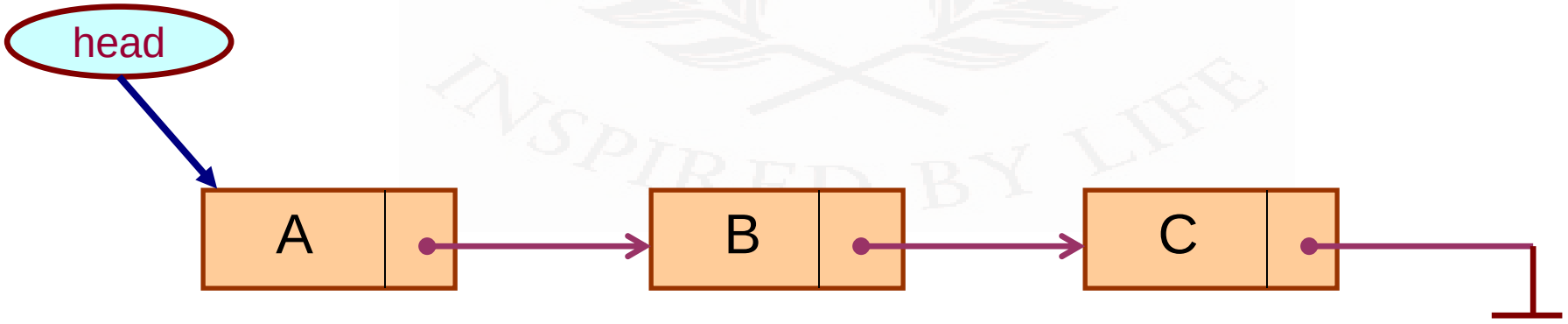
Linked list

Why Linked Lists?

- Advantages of Arrays:
  - ❖ Data access is faster
  - ❖ Simple
- Disadvantages:
  - ❖Size of the array is fixed.
  - ❖Array items are stored contiguously.
  - ❖Insertion and deletion operations involve tedious job of shifting the elements with respect to the index of the array.

Introduction

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



Introduction

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Illustration: Insertion

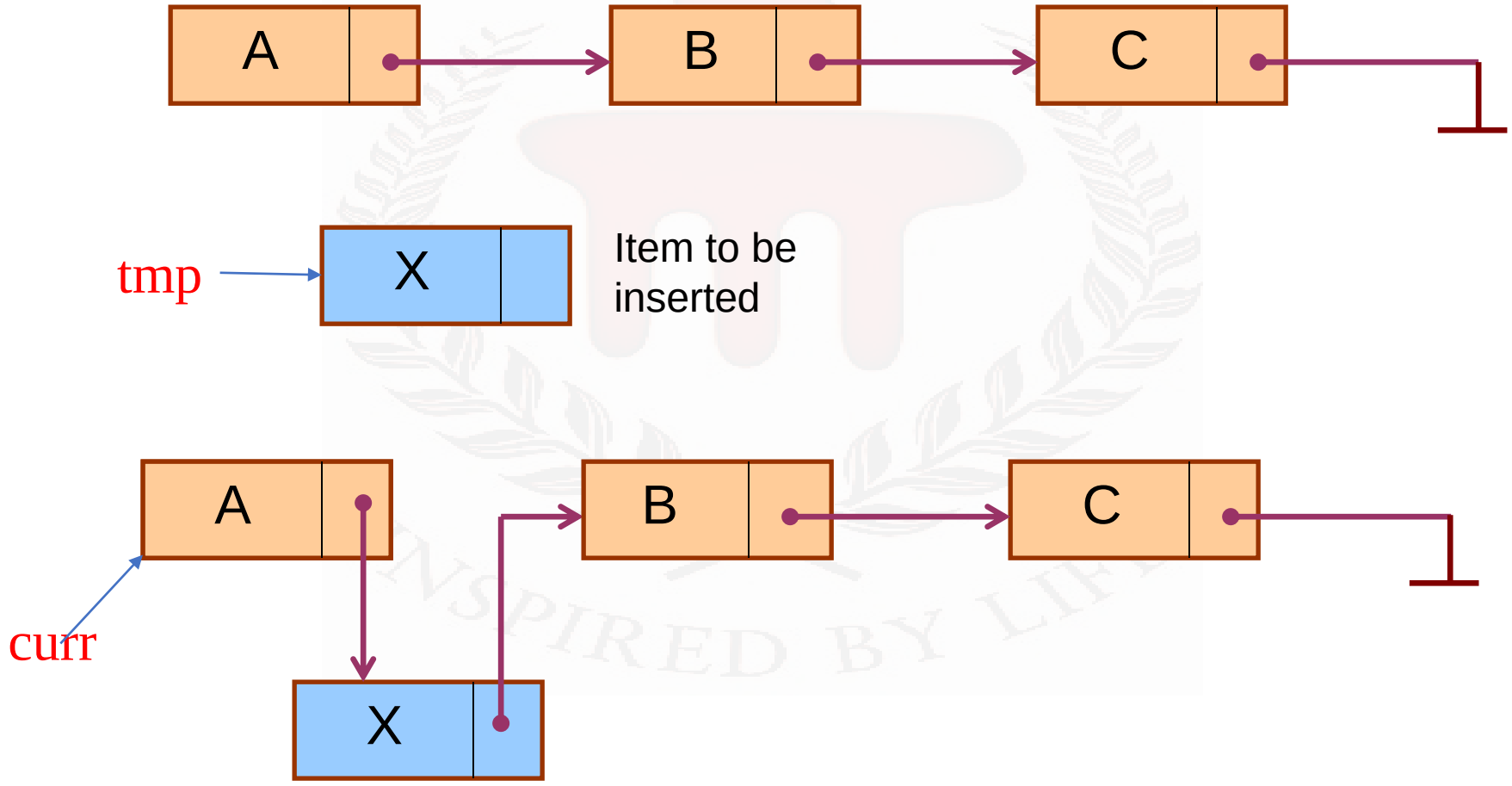
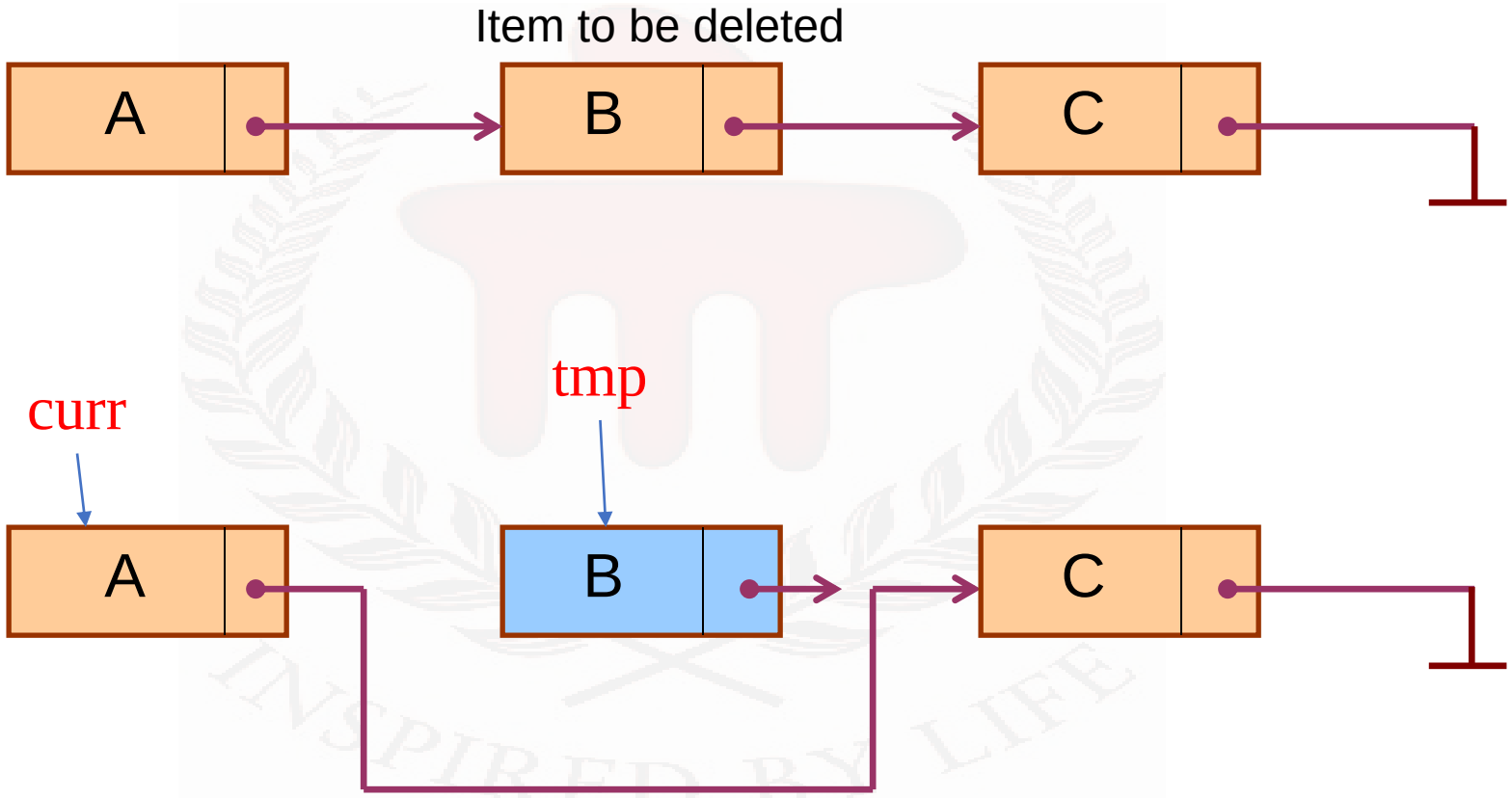


Illustration: Deletion



Summary

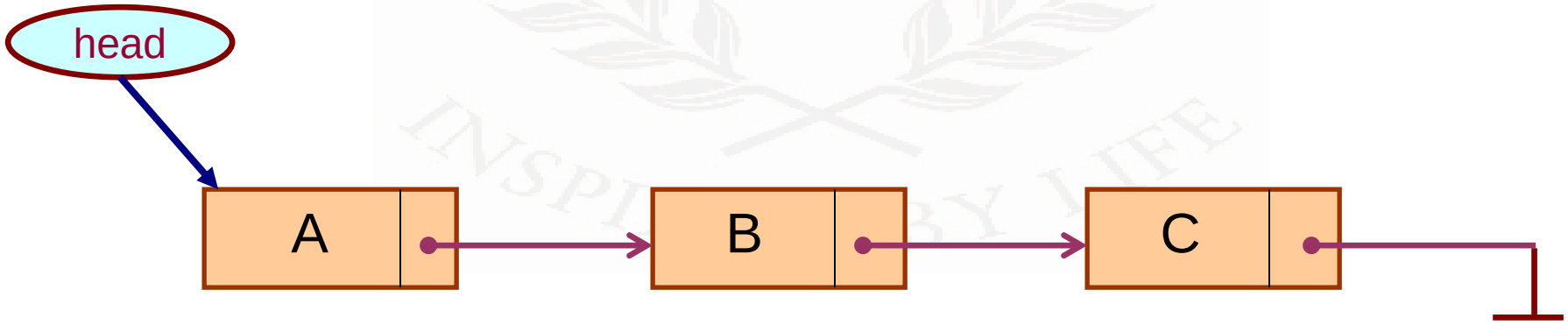
- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Array versus Linked Lists

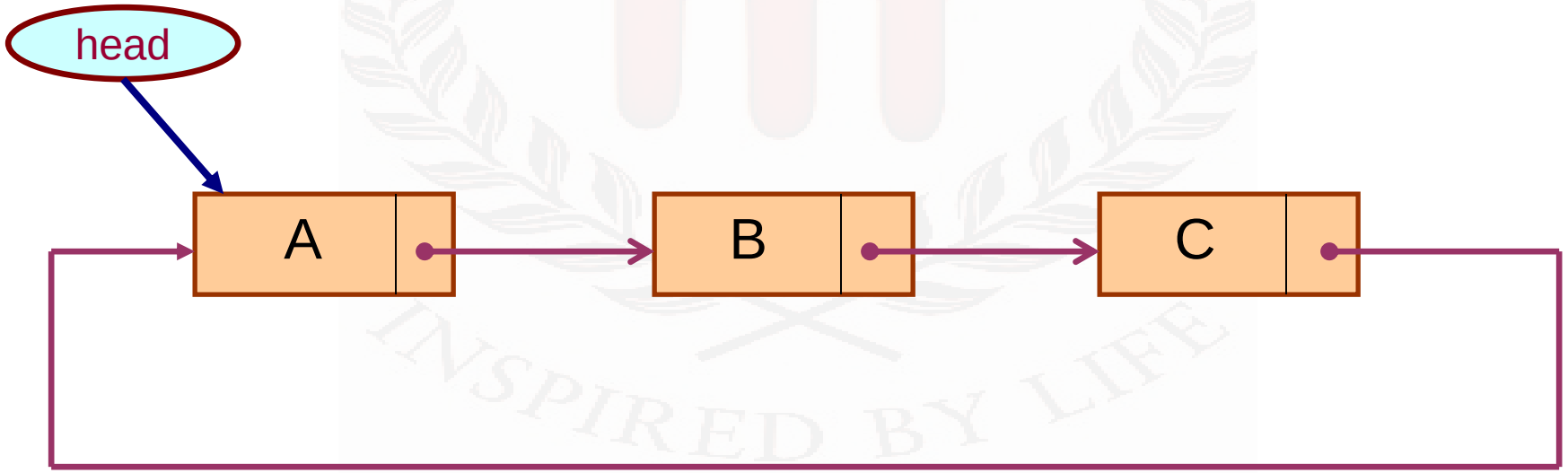
- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.

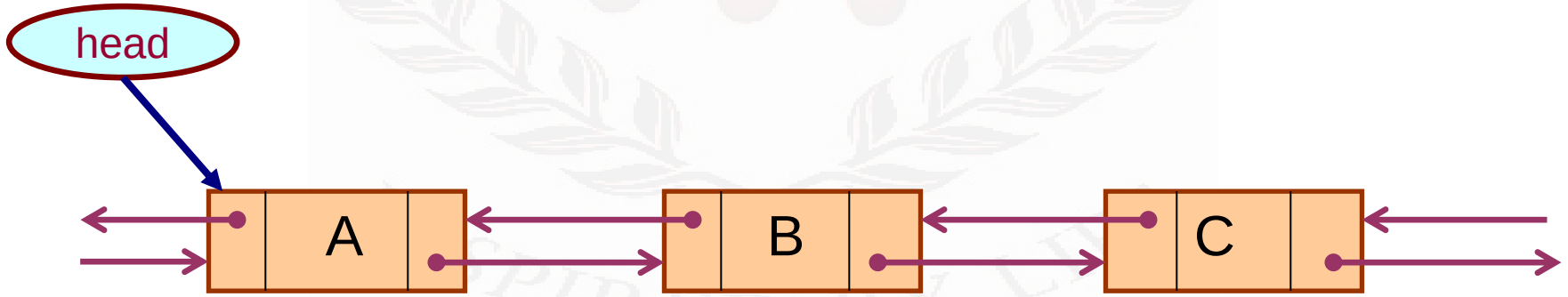


- Circular linked list
  - The pointer from the last element in the list points back to the first element.



- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.

.



Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

```
#include <stdio.h>
#include <stdlib.h>

// Linked List Node
struct node {
        int info;
        struct node* link;
};
struct node* start = NULL;
```

```
// Function to create list with n nodes initially
void createList()
{
        if (start == NULL) {
                int n;
                printf("\nEnter the number of nodes: "); scanf("%d", &n);
                if (n != 0) {   int data;
                        struct node* newnode; struct node* temp;
                        newnode = malloc(sizeof(struct node));
                        start = newnode; temp = start;
                        printf("\nEnter number to be inserted : ");
                        scanf("%d", &data);
                        start->info = data;
                        for (int i = 2; i <= n; i++) {
                                newnode = malloc(sizeof(struct node));
                                temp->link = newnode;
                                printf("\nEnter number to be inserted : ");     scanf("%d", &data);
                                newnode->info = data;           newnode->link=NULL;
                                temp = temp->link;
                        }
                }
                printf("\nThe list is created\n");
        }
        else    printf("\nThe list is already created\n");
}
```

```
// Function to traverse the linked list
void traverse()
{
        struct node* temp;
        if (start == NULL) printf("\nList is empty\n");
        // Else print the LL
        else {
                temp = start;
                while (temp != NULL) {
                        printf("Data = %d\n", temp->info);
                        temp = temp->link;
                }
        }
}
```

```
// Function to insert at the front of the linked list
void insertAtFront()
{
        int data;-
        struct node* temp;
        temp = malloc(sizeof(struct node));
        printf("\nEnter number to"
                " be inserted : ");
        scanf("%d", &data);
        temp->info = data;

        // Pointer of temp will be
        // assigned to start
        temp->link = start;
        start = temp;
}
```

```
// Function to insert at the end of the linked list
void insertAtEnd()
{
        int data;
        struct node *temp, *head;
        temp = malloc(sizeof(struct node));
        printf("\nEnter number to be inserted : ");
        scanf("%d", &data);
        temp->link = 0;
        temp->info = data;
        head = start;
        if(head==NULL)  start=temp;
        else{
        while (head->link != NULL) {
                head = head->link;
        }
        head->link = temp;
        }
}
```

```
// Function to insert at any specified position in the linked list
void insertAtPosition()
{
        struct node *temp, *newnode;
        int pos, data, i = 1;
        newnode = malloc(sizeof(struct node));
        // Enter the position and data
        printf("\nEnter position and data :");
        scanf("%d %d", &pos, &data);
        temp = start;   newnode->info = data;   newnode->link = 0;
        while (i < pos - 1) {
                temp = temp->link;
                i++;
        }
        newnode->link = temp->link;
        temp->link = newnode;
}
```

```
// Function to delete from the front of the linked list
void deleteFirst()
{
        struct node* temp;
        if (start == NULL)
                printf("\nList is empty\n");
        else {
                temp = start;
                start = start->link;
                free(temp);
        }
}
```

```
// Function to delete from the end
// of the linked list
void deleteEnd()
{       struct node *temp, *prevnode;
        if (start == NULL)      printf("\nList is Empty\n");
        else if(start->link==NULL) {free(start); start=NULL;}
        else {
                temp = start;
                while (temp->link != 0) {
                        prevnode = temp;
                        temp = temp->link;
                }
                free(temp);
                prevnode->link = 0;
        }
}
```

```
// Function to delete from any specified position from the linked list
void deletePosition()
{       struct node *temp, *position;   int i = 1, pos;
        if (start == NULL) printf("\nList is empty\n"); // If LL is empty
        else {
                printf("\nEnter index : ");     // Position to be deleted
                scanf("%d", &pos);
                temp = start;
                while (i < pos - 1) {
                        temp = temp->link;
                        i++;
                }
                position = temp->link;
                temp->link = position->link;
                free(position);
        }
}
```

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct Node *head)
{
    if (head == NULL) return;

    /* Initialize prev and node to be deleted */
    struct Node *prev = head;
    struct Node *node = head->link;

    while (prev != NULL && node != NULL)
    {
        /* Change next link of previous node */
        prev->link = node->link;

        /* Free memory */
        free(node);
        /* Update prev and node */
        prev = prev->link;
        if (prev != NULL)
            node = prev->link;
    }
}
```

```
struct node *reverse(struct node *start)
{
        struct node *prev, *ptr, *next;
        prev=NULL;
        ptr=start;
        while(ptr!=NULL)
        {
                next=ptr->link;
                ptr->link=prev;
                prev=ptr;
                ptr=next;
        }
        start=prev;
        return start;
}/*End of reverse()*/
```

Working on Multiple Lists

List Concatenation (1/6)

```
struct node
{
        int info;
        struct node *link;
};
struct node *create_list(struct node *);
struct node *concat( struct node *start1,struct node *start2);
struct node *addatbeg(struct node *start, int data);
struct node *insert(struct node *start,int data); //addtoend
void display(struct node *start);
```

List Concatenation (2/6)

```
int main()
{
        struct node *start1=NULL,*start2=NULL;
        start1=create_list(start1);
        start2=create_list(start2);
        printf("\nFirst list is  : ");        display(start1);
        printf("\nSecond list is  : ");        display(start2);
        start1=concat(start1, start2);
        printf("\nConcatenated list is  : ");
        display(start1);
        return 0;

}/*End of main()*/
```

List Concatenation (3/6)

```
struct node *concat( struct node *start1,struct node *start2)
{
        struct node *ptr;
        if(start1==NULL)
        {
                start1=start2;
                return start1;
        }
        if(start2==NULL)
                return start1;
        ptr=start1;
        while(ptr->link!=NULL)
                ptr=ptr->link;
        ptr->link=start2;
        return start1;
}
```

List Concatenation (4/6)

```
struct node *create_list(struct node *start)
{
        int i,n,data;
        printf("\nEnter the number of nodes : ");
        scanf("%d",&n);
        start=NULL;
        if(n==0)
                return start;

        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addatbeg(start,data);

        for(i=2;i<=n;i++)
        {
                printf("Enter the element to be inserted : ");
                scanf("%d",&data);
                start=insert(start,data);
        }
        return start;
}/*End of create_list()*/
```

List Concatenation (5/6)

```
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=start;
        start=tmp;
        return start;
}/*End of addatbeg()*/
```

List Concatenation (6/6)

```
struct node *insert(struct node *start, int info)
{
        struct node *p,*tmp;
        tmp= (struct node *)malloc(sizeof(struct node));
        tmp->info=info;
        tmp->link=NULL;
        if(start==NULL)
        {
            start=temp;
            return(start);
        }
        p=start;
        while(p->link!=NULL)
                p=p->link;
        p->link=tmp;
        tmp->link=NULL;
        return start;
}/*End of insert()*/
```

Merge two lists

- Both input lists are sorted

Case 1

- By creating new third sorted list
- New third list is sorted

Case 2

- Without creating any new node/list.
- Input list1 or list2 is resulting list
- list1 is result first1->data<=first2->data
- list2 is result first2->data<first1->data

Merge into new third list (1/2)

```
void merge(struct node *p1,struct node *p2)
{
        struct node *start3;
        start3=NULL;

        while(p1!=NULL && p2!=NULL)
        {
                if(p1->info < p2->info)
                {
                        start3=insert(start3,p1->info);
                        p1=p1->link;
                }
                else if(p2->info < p1->info)
                {
                        start3=insert(start3,p2->info);
                        p2=p2->link;
                }
                else if(p1->info==p2->info)
                {
                        start3=insert(start3,p1->info);
                        p1=p1->link;
                        p2=p2->link;
                }
        }
```

Merge into new third list (2/2)

```
        /*If second list has finished and elements left in first list*/
        while(p1!=NULL)
        {
                start3=insert(start3,p1->info);
                p1=p1->link;
        }
        /*If first list has finished and elements left in second list*/
        while(p2!=NULL)
        {
                start3=insert(start3,p2->info);
                p2=p2->link;
        }
        printf("Merged list is : ");
        display(start3);
}
```

Merge without creating any new node/list (1/3)

```
struct node* mwithout_new(struct node *firsta,struct node *firstb)
{
        struct node *curra=firsta;
        struct node *preva=NULL,*prevb=NULL;
        struct node *currb=firstb;
        struct node *res=NULL;
        int pa=0,pb=0;
    if(firsta->info<=firstb->info)
    {
        pa=1;
        res=firsta;
    }
    else
    {
         pb=1;
        res=firstb;
    }
```

Merge without creating any new node/list (2/3)

```
while(curra!=NULL&&currb!=NULL)
{
        if(curra->info<=currb->info)
        {
                if(pb==1)
                {
                        prevb->link=curra;
                        pa=1;
                        pb=0;
                 }
                preva=curra;
                curra=curra->link;
        }
        else //curra->info>currb->info
        {
                if(pa==1)
                 {
                         preva->link=currb;
                        pa=0;
                        pb=1;
                }
                prevb=currb;
                currb=currb->link;
        }
}//while
```

Merge without creating any new node/list (3/3)

```
        if(curra)       prevb->link=curra;
        if(currb)               preva->link=currb;
        display(res);
        return(res);
}

int main()
{
        struct node *s1,*s2,*start1=NULL,*start2=NULL,*start3=NULL;
        start1=create_list(start1);
        start2=create_list(start2);
             s1=start1; s2=start2;
        printf("\nFirst list is  : ");        display(start1);
        printf("\nSecond list is  : ");        display(start2);

        printf("\nMerged into thrid list: ");
        merge(start1, start2);

        printf("\nMerged without new node: ");
        //start3=mwithout_new(s1, s2);
        display(start3);
        return 0;

}/*End of main()*/
```