# UCS1302:
# DATA STRUCTURES

## Linked list ADT

# Session Meta Data

| | |
|---|---|
| **Author** | **Dr. B. Bharathi** |
| Reviewer | |
| Version Number | 1.2 |
| Release Date | 01  July 2019 |

# Revision History

| Revision Date | Details | Version no. |
|---|---|---|
| 22 September 2017 | 1. New SSN template applied | 1.2 |

# Session Objectives

- To learn about Linked list ADT
- Implementation of Linked list

# Session Outcomes

- At the end of this session, participants will be able to
  - Understand the concepts of Linked list ADT
  - Implementation of Linked list ADT

# Agenda

- Linked list ADT
- Implementation of linked list operations

# Linked List ADT

Dr. B. Bharathi
SSNCE

July 01, 2019

# Linked list

- Alternate approach to maintaining an array of elements

- Rather than allocating one large group of elements, allocate elements as needed

Q: how do we know what is part of the array?

A: have the elements keep track of each other

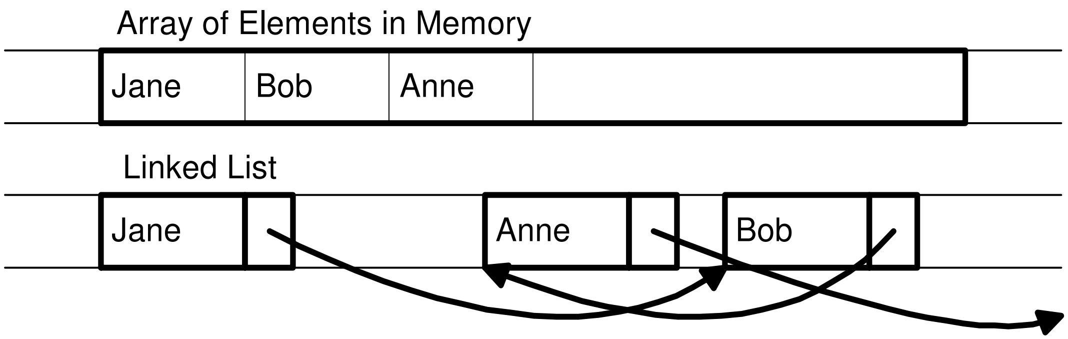
use pointers to connect the elements together as a *LIST* of things

# Limitation of arrays

- An array has a limited number of elements
  - routines inserting a new value have to check that there is room
- Can partially solve this problem by reallocating the array as needed (how much memory to add?)
  - adding one element at a time could be costly
  - one approach - double the current size of the array
- A better approach: use a *Linked List*

# Dynamically Allocating Elements

- Allocate elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer



# Linked List

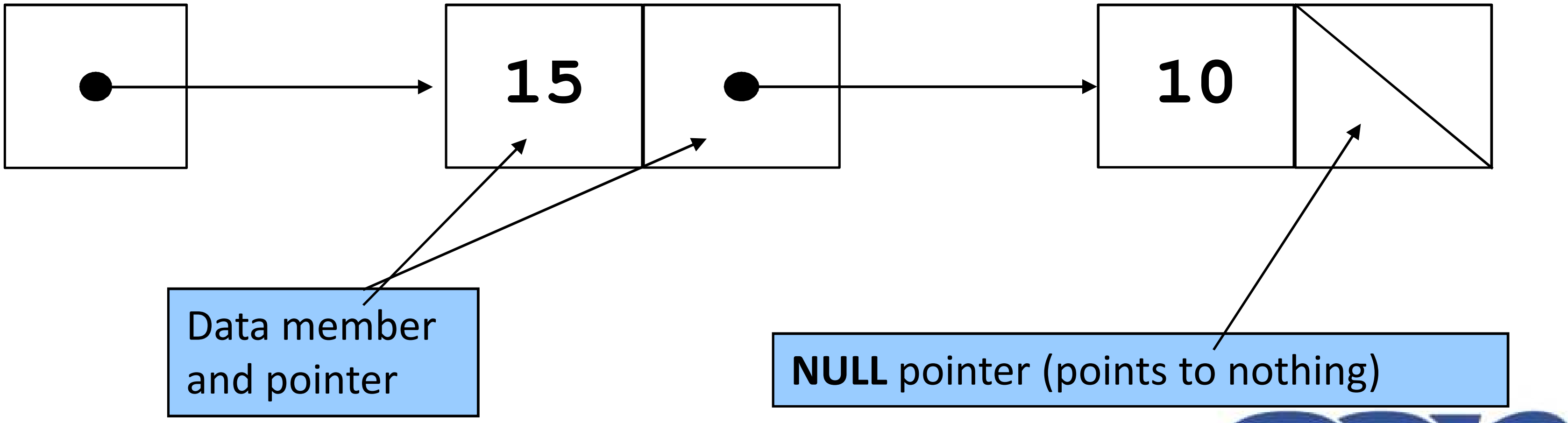
- Need way to indicate end of list (NULL pointer)
- Need to know where list starts (first element)
- Each element needs pointer to next element (its link)
- Need way to allocate new element (use malloc)
- Need way to return element not needed any more (use free)
- Divide element into data and pointer

# Types of linked list

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked
  - Pointer in the last node points back to the first node
- Doubly linked list
  - Two "start pointers" – first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- Circular, doubly linked list
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

# Self referential structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a **NULL** pointer (**0**)
- Diagram of two self-referential structure members linked together



15

10

Data member and pointer

**NULL** pointer (points to nothing)

# Self referential structures

```
struct node {
        int data;
        struct node *nextPtr;
     }
```

- **nextPtr**
  - Points to a structure of type node
  - Referred to as a link
    - Ties one **node** to another **node**

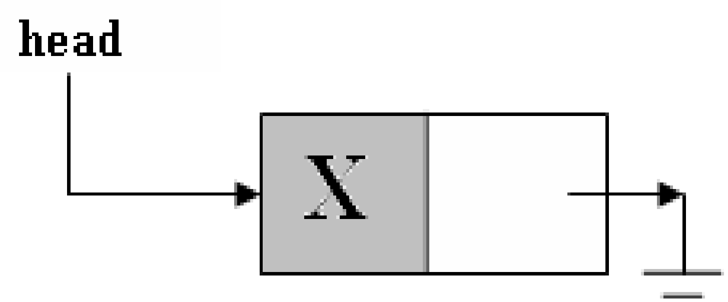
# Basic operations on the list

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list

# Creating a empty list

```
typedef struct mynode
{
        int data;
        struct mynode *next;
}node;


  main()
  {
      // Declarations
        node *head;
        head = CreateEmptyList();
  }
```

```
node* CreateEmptyList()
{
     node *h;
     h = (node*)malloc(sizeof(node));
     h->next = NULL;
      return h;
}
```



# Inserting a node in Singly Linked List(SLL)

- Insertion at the beginning
- Insertion at the end
- Insertion after a particular node

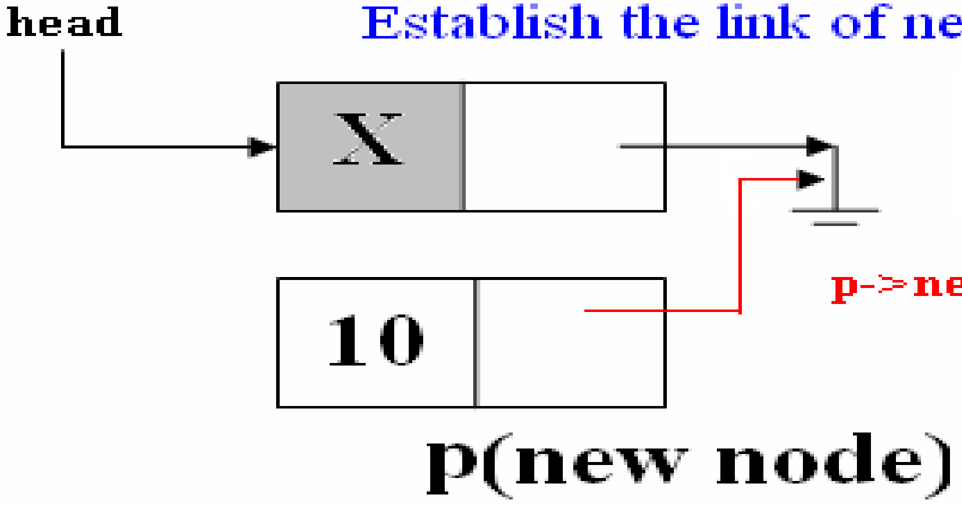
# Insertion at the beginning

```
void InsertFirst(node *hd,int element)
{
        node *p;
        p =(node*)malloc(sizeof(node));
        p->data = element;
        p->next = hd->next;
        hd->next = p;
}
```

# Insert First Explanation

(1)

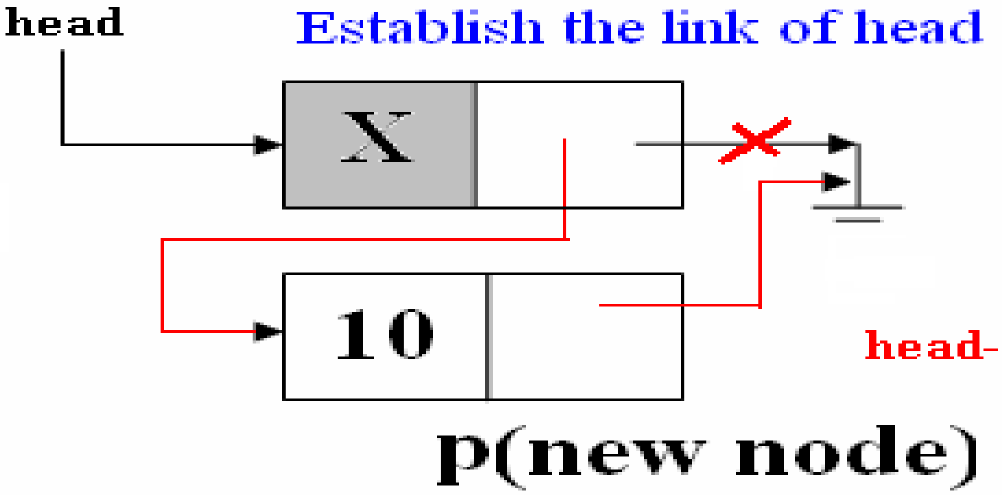
Establish the link of new node first



p->next = head->next

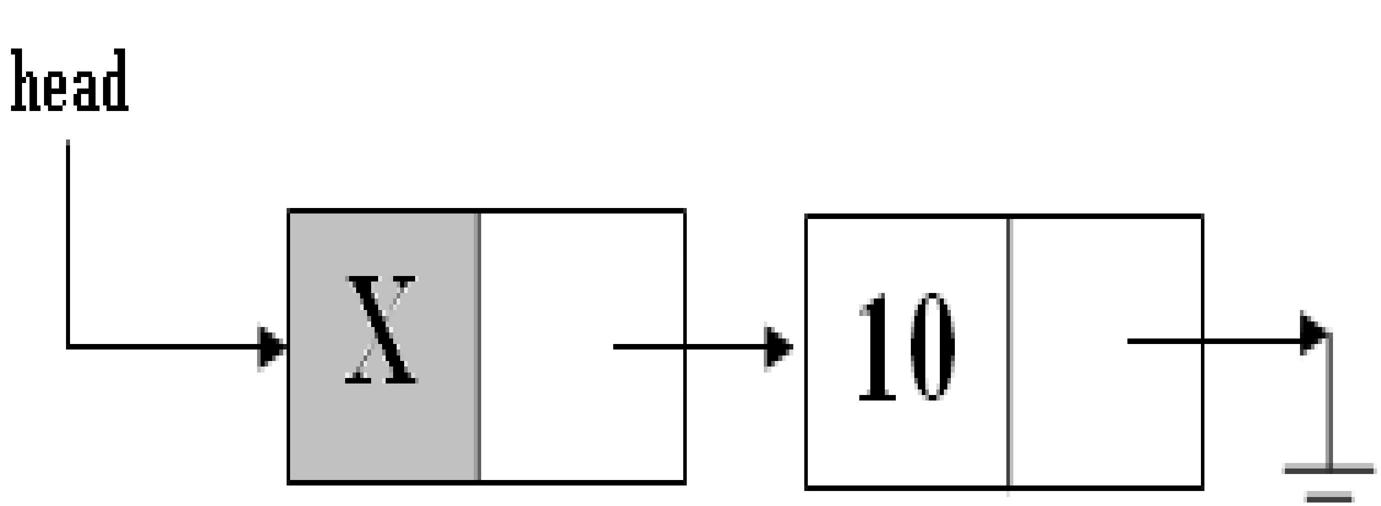
(2)

Establish the link of head node then

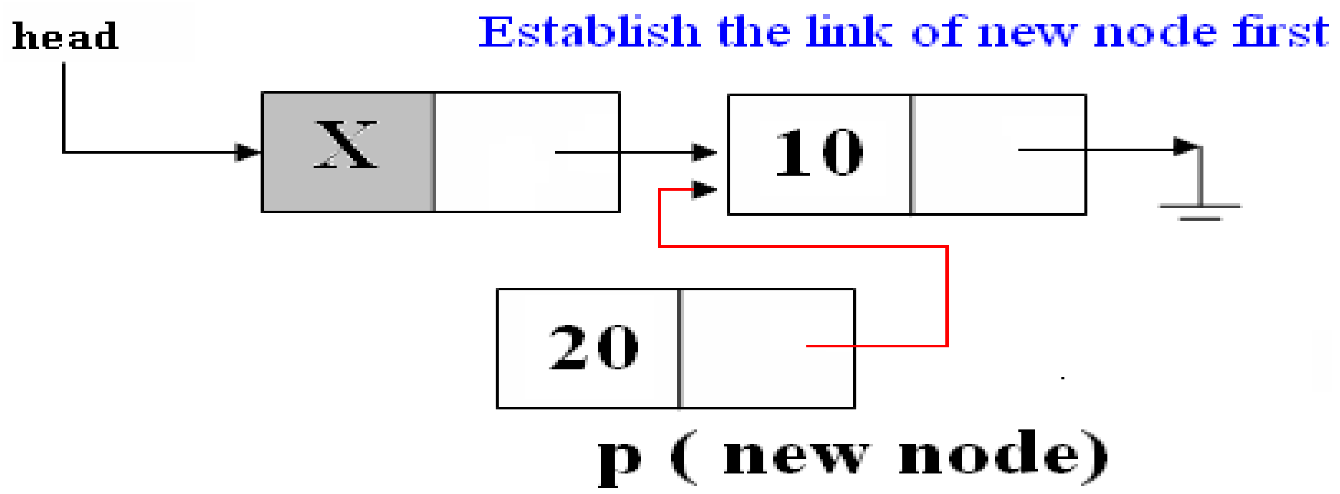


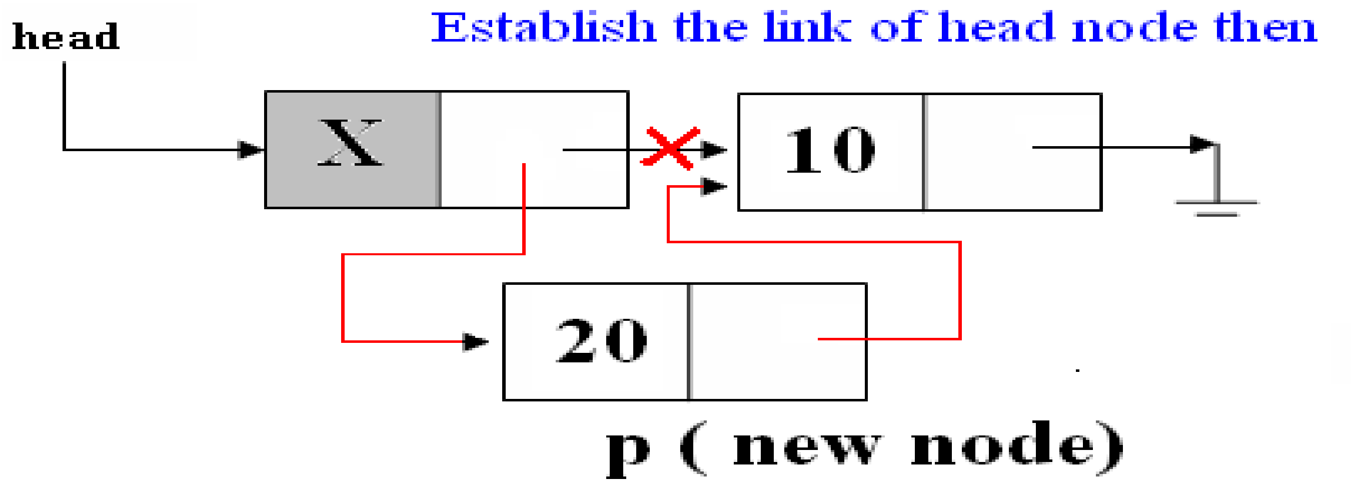
head->next = p

# Insert First ( Cont .. )



# Insert First ( Cont .. )


Establish the link of new node first


Establish the link of head node then

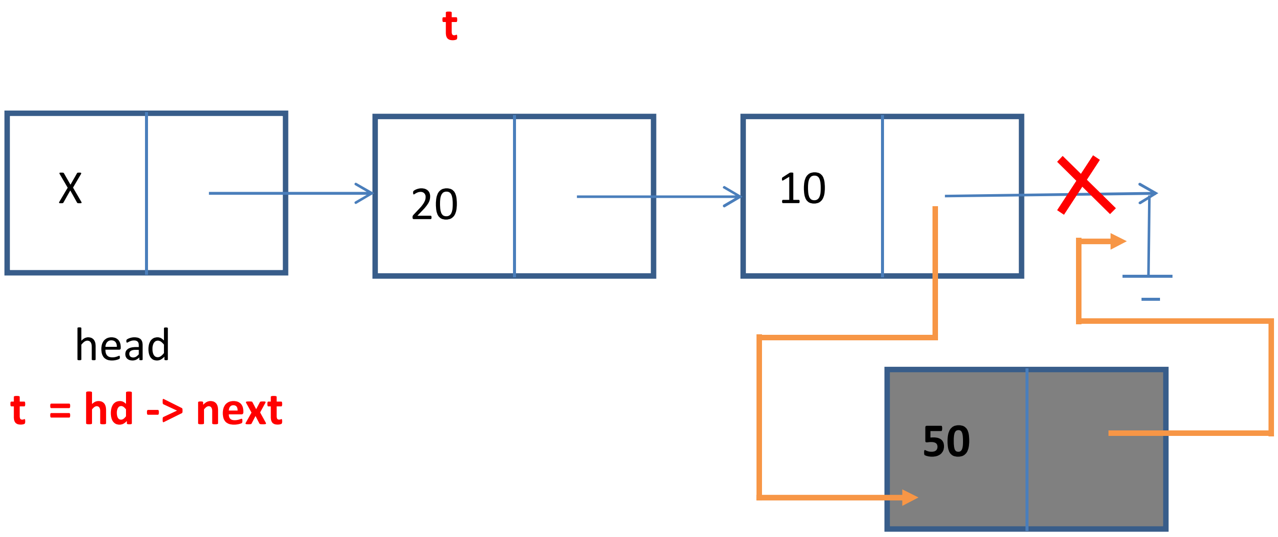
# Insert last

```
void InsertLast(node *hd,int element)
{
        node *p,*t;
        p =(node*)malloc(sizeof(node));
        p->data = element;
        t = hd->next;
        while(t->next!=NULL)
                t = t->next;
        p->next =t->next;
        t->next = p;
}
```

# Insert Last

## 1. Traversing

## 2. Insert the new node @ last pos



t = hd -> next

P->data = element

p->next = t->next

t->next = p

t->next != NULL , t = t -> next

t->next = NULL , Fails

# Insert at Position

```
main()
{

printf("\n Enter Element to insert : ");

scanf("%d",&n);

printf("\n Enter Position to insert : ");

scanf("%d",&pos);

p = FindPrevious(head,pos);

InsertMiddle(n,p);
            .
            .
            .


}
```

```
node* FindPrevious(node *hd,int pos)
{
    int i =1;
    node *t;
    t=hd;
    while(i<pos)
    {
        i++;
        t = t->next;
    }
    if(t!=NULL)
            return t;
    else
            return NULL;
}
```

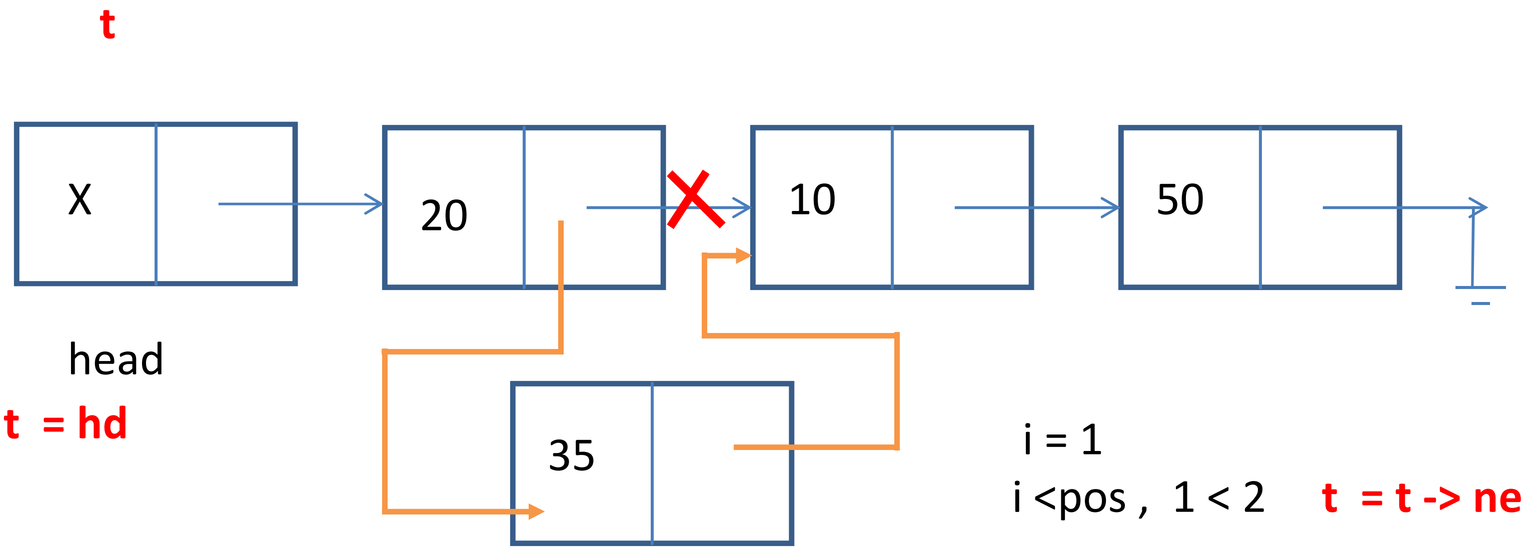
# Insert at Position ( Cont…)

```
void InsertMiddle(int element, node *t)
{
        node *p;
        p = (node*)malloc(sizeof(node));
        p->data = element;
        p->next = t->next;
        t->next = p;
}
```

# Insert at Position 2

## 1. Find Previous

## 2. Insert the new node @ pos 2



**t = hd**

i = 1
i <pos , 1 < 2 **t = t -> next**

i = 2
i <pos , 2 < 2 **False**

**P->data = element**

**p->next = t->next**

**t->next = p**

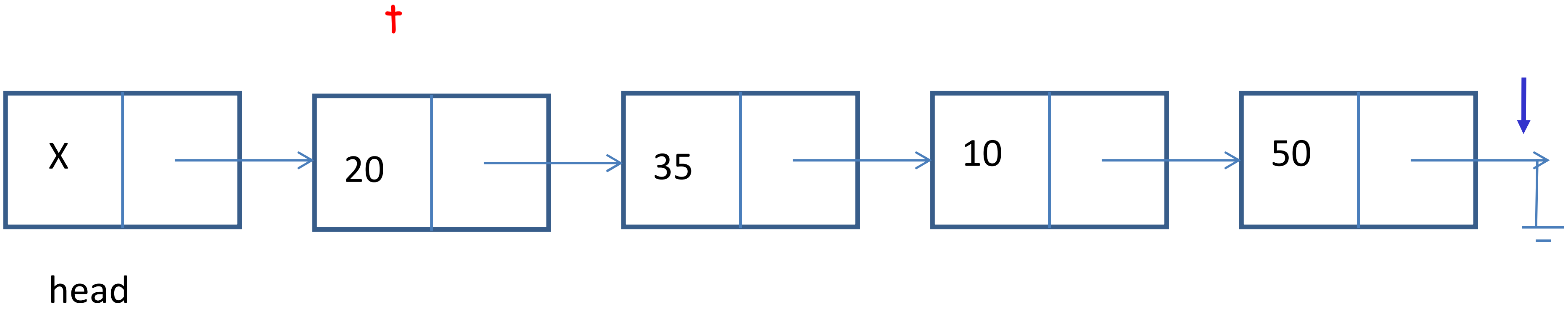
# Display List

```
main()
{

    DisplayList(head);
}

short isEmpty(node *hd)
{
  return( hd->next == NULL );
}
```

```
void DisplayList(node *hd)
{
   node *t;
  printf("\n Displaying Elements : ");
  if( isEmpty(hd) )
          printf(" List is Empty ");
   for(t=hd->next;t!=NULL;t=t->next)
           printf("%d\t",t->data);
}
```

# Displaying the list of Elements

Increment t t = t-> next



t = head -> next

Check t ! = NULL, TRUE FALSE

Printing 20 35 10 50

# Delete First

```
main()
{

 n=DeleteFirst(head);
  if(n!=-1)
     printf("\n Deleted Element :

        %d",n);
   else
       printf("\n List Empty");
}
```

```
int DeleteFirst(node *hd)
{
        node *tmp;
        int r_item;
        if(!isEmpty(hd))
        {
               tmp = hd->next;
               hd->next = hd->next->next;
                r_item = tmp->data;
                free(tmp);
                return r_item;
        }
        else
                return -1;
}
```

# Delete Last

```
main()
{

 n=DeleteLast(head);
  if(n!=-1)
             printf("\n Deleted
     Element : %d",n);
   else
           printf("\n List Empty");

}
```

```
int DeleteLast(node *hd)
{
        node *t,*tmp;
        int r_item=-1;
        t = hd;
        while(t->next->next!=NULL)
                  t = t->next;
        tmp = t->next;
        r_item = tmp->data;
        t->next = NULL;
        return r_item;
}
```

# Delete in the Middle

```
main()
{

printf(“ Enter Position to delete : ");
scanf("%d",&pos);
p =FindPrevious(head,pos);
n = DeleteMiddle(p);
if(n!=-1)
 printf("\n Deleted Element : %d",n);

 else
     printf("\n List Empty");              .
}
```

```
int DeleteMiddle(node *p)
{
      node *tmp;
      int r_item=-1;
      tmp = p->next;
      p->next = p->next->next;
      r_item = tmp->data;
      free(tmp);
      return r_item;
}
```

# Summary

- Linked list ADT
- Linked list operations