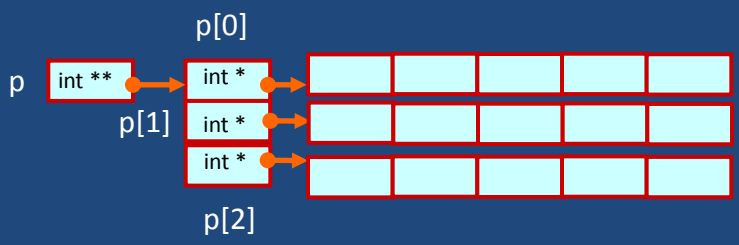
PDS Class Test 2

- Date: October 27, 2016
- Time: 7pm to 8pm
- Marks: 20 (Weightage 50%)

Room	Sections	No of students
V1	Section 8 (All) Section 9 (AE,AG,BT,CE, CH,CS,CY,EC,EE,EX)	101 49
V2	Section 9 (Rest, if not allotted in V1) Section 10 (All)	50 98
V3	Section 11 (All)	98
V4	Section 12 (All)	94
F116	Section 13 (All)	95
F142	Section 14 (All)	96

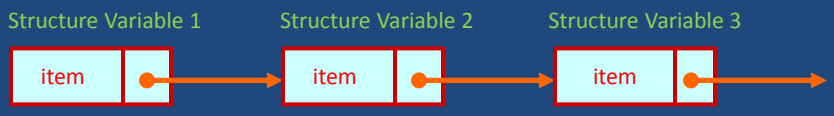
Pointer to Pointer

```
int **p;
p=(int **) malloc(3 * sizeof(int *));
p[0]=(int *) malloc(5 * sizeof(int));
p[1]=(int *) malloc(5 * sizeof(int));
p[2]=(int *) malloc(5 * sizeof(int));
```



Linked List

- A completely different way to represent a list:
  - Make each item in the list part of a structure.
  - The structure contains the item and a pointer or link to the structure containing the next item.
  - This type of list is called a linked list.



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.

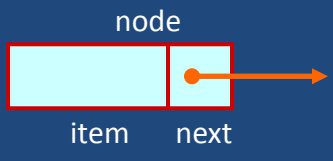
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Linked List Facts

- Each structure of the list is called a node, and consists of two fields:
  - Item(s).
  - Address of the next item in the list.
- The data items comprising a linked list need not be contiguous in memory.
  - They are ordered by logical links that are stored as part of the data in the structure itself.
  - The link is a pointer to another structure of the same type.

Declaration of a linked list

```
struct node
       {
         int   item;
         struct node  *next;
       };
```



- Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Illustration

- Consider the structure:

```
struct stud
 {
    int roll;
    char name[30];
    int age;
    struct stud *next;
 };
```

- Also assume that the list consists of three nodes n1, n2 and n3.

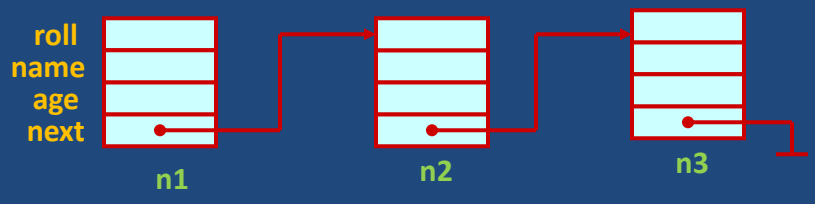
```
struct stud n1, n2, n3;
```

Illustration

- To create the links between nodes, we can write:

```
n1.next = &n2 ;
n2.next = &n3 ;
n3.next = NULL ;  /* No more nodes follow */
```

- Now the list looks like:



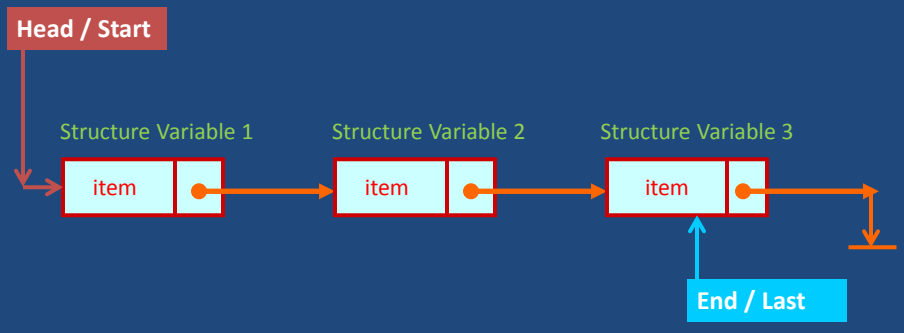
Example

```
#include  <stdio.h>
struct  stud
 {
    int  roll;
    char  name[30];
    int  age;
    struct  stud  *next;
 };

main()
{
   struct  stud  n1, n2, n3;
   struct  stud  *p;

   scanf ("%d %s %d", &n1.roll,
                      n1.name, &n1.age);
   scanf ("%d %s %d", &n2.roll,
                      n2.name, &n2.age);
   scanf ("%d %s %d", &n3.roll,
                      n3.name, &n3.age);
```

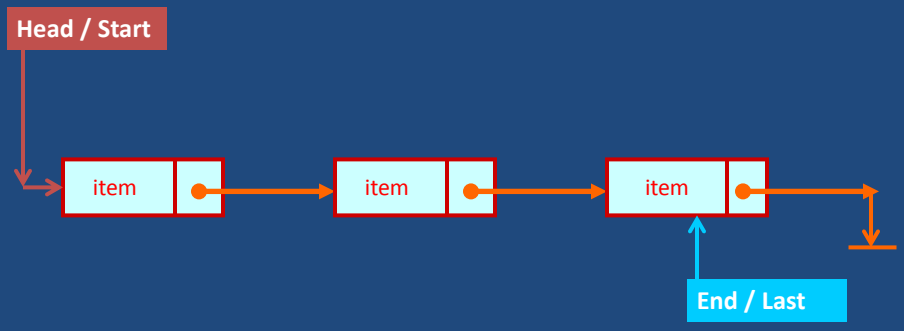
```
   n1.next  =  &n2 ;
   n2.next  =  &n3 ;
   n3.next  =  NULL ;

/* Now traverse the list and print
  the elements */

   p  =  &n1 ;  /* point to 1st element */
   while  (p != NULL)
   {
      printf ("\n %d %s %d",
        p->roll, p->name, p->age);
        p  =  p->next;
   }
}
```

Linked List

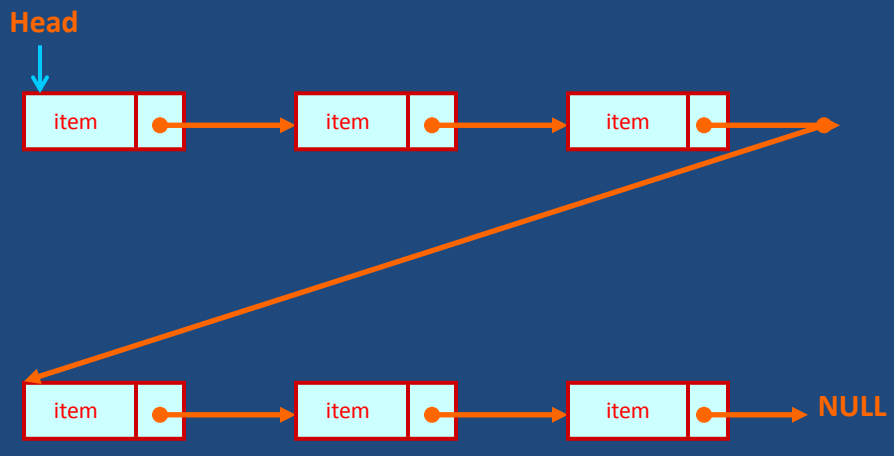
- Where to start and where to stop?



Linked List



Traverse a linked list



Example

```
#include  <stdio.h>
struct  stud
 {
    int  roll;
    char  name[30];
    int  age;
    struct  stud  *next;
 };

main()
{
   struct  stud  n1, n2, n3, *p;
……..
 p  =  &n1 ;  /* point to 1st element */
   while  (p != NULL)
   {
      printf ("\n %d %s %d", p->roll, p->name, p->age);
        p  =  p->next;
   }
……..
}
```

Insert into a linked list

Step 1: Create a new node.

Step 2: Copy the item.

Step 3: Make the link/next as NULL (point nowhere)

Step 4:

Case 1: If there does not exists any linked list.

Step 4a: Make the new node as head node.

Step 4b: Go to End.

Case 2: Else

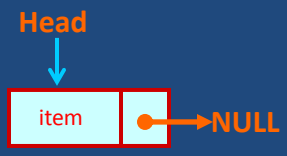
Step 4c: Locate the insertion point.

Step 4d: Insert the new node.

Step 4e: Adjust the link.

Step 4f: Go to End.

Insert into a linked list



Step 1: Create a new node.

Step 2: Copy the item.

Step 3: Make the link/next as NULL (point nowhere)

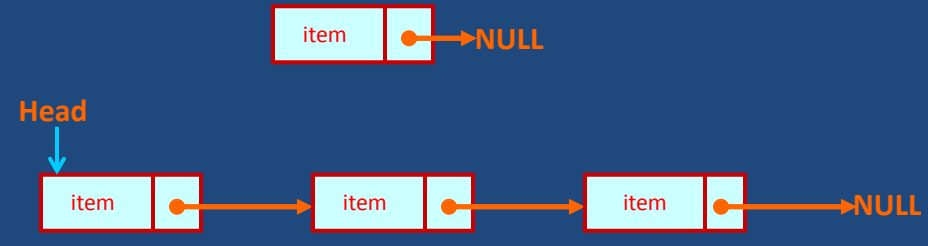
Step 4:

Case 1: If there does not exists any linked list.

Step 4a: Make the new node as head node.

Step 4b: Go to End.

Insert into a linked list



Step 1: Create a new node.
Step 2: Copy the item.
Step 3: Make the link/next as NULL (point nowhere)
Step 4:

Case 2: Else

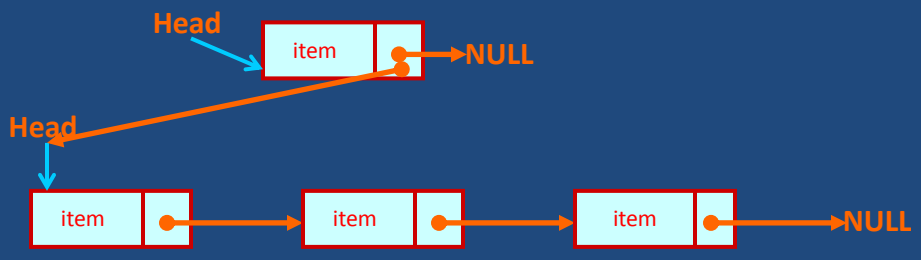
Step 4c: Locate the insertion point.

Step 4d: Insert the new node.

Step 4e: Adjust the link.

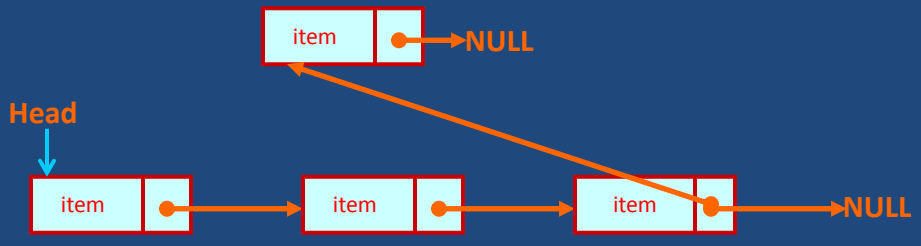
Step 4f: Go to End.

Insert into a linked list: Head Node



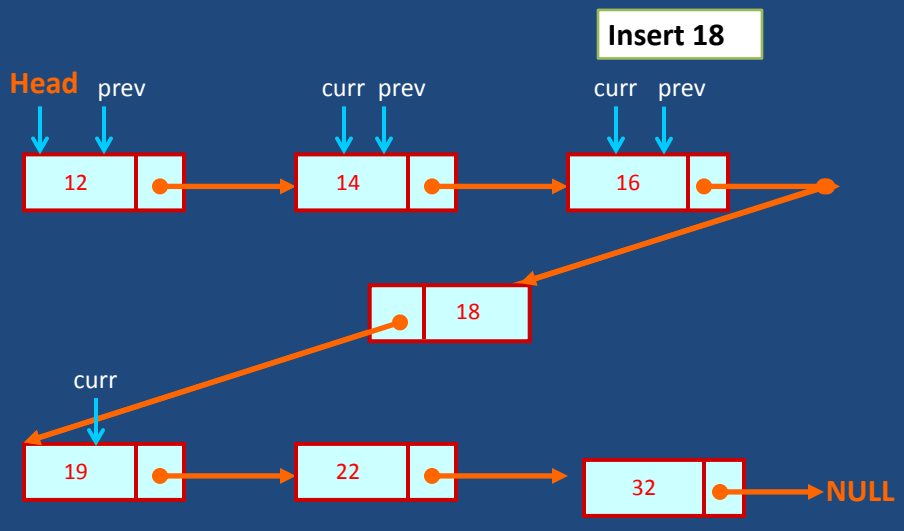
Step 4ci: Make the next of new node as the address of existing head node.
Step 4cii: Copy the address of the new node as the head node.

Insert into a linked list: End Node

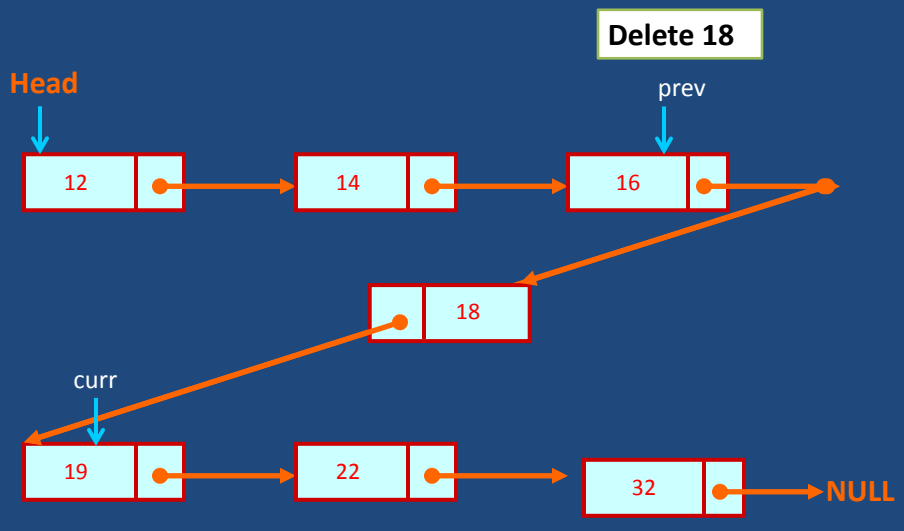


Step 4ci: Traverse till last/end node.
Step 4cii: Make the next of last node as the address of new node.

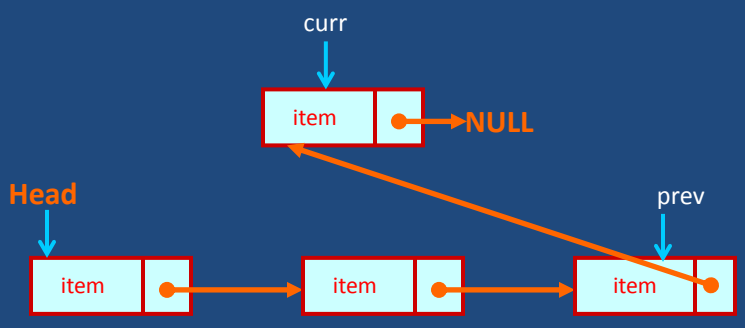
Insert into a sorted linked list



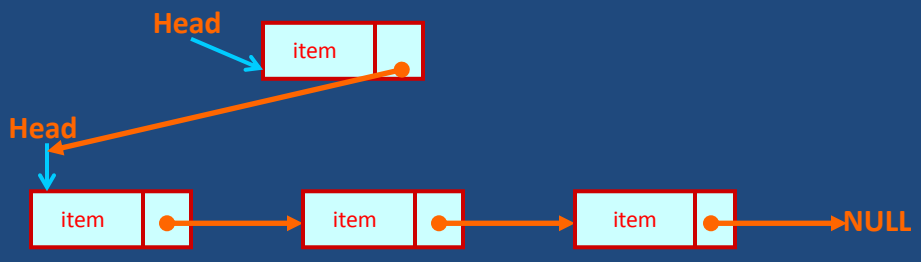
Delete a specific node from linked list



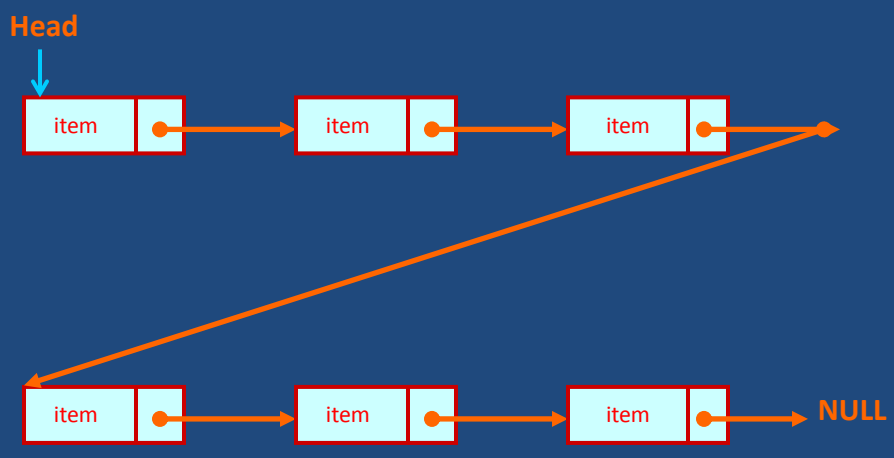
Delete End Node from linked list



Delete Head Node from a linked list



Linked list and Dynamic Memory Allocation



Linked list and Dynamic Memory Allocation

1. We need not have to know how many nodes are there.

2. Dynamic memory allocation provides a flexibility on the length of a linked list.

3. Example,

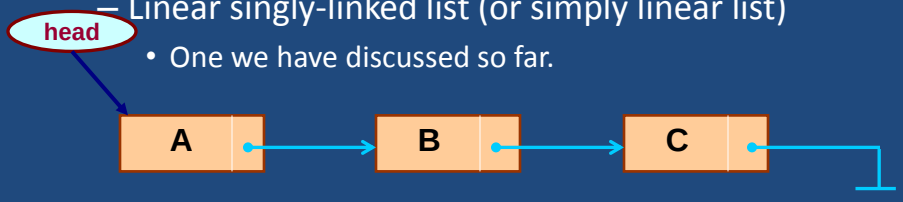
```
struct node {
        int item;
        struct node *next;
};
struct node *head, *temp;

temp=(struct node *)malloc(sizeof(struct node)*1);
temp->next=NULL;
temp->item=10;
head=temp;

free(temp);
```

Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

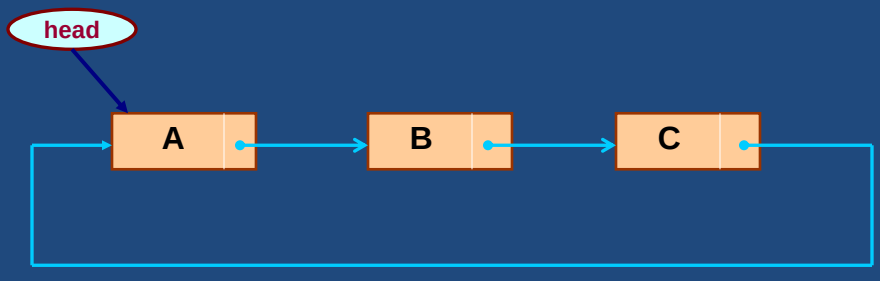
– Linear singly-linked list (or simply linear list)

- One we have discussed so far.



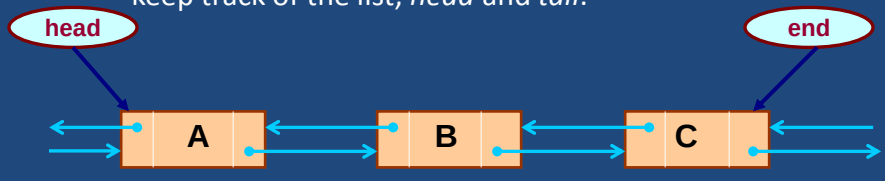
– **Circular linked list**

- The pointer from the last element in the list points back to the first element.
- No need for NULL link.
- How do you keep track of traversal?



– **Doubly linked list**

- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers (prev and next) are maintained to keep track of the list, *head* and *tail*.

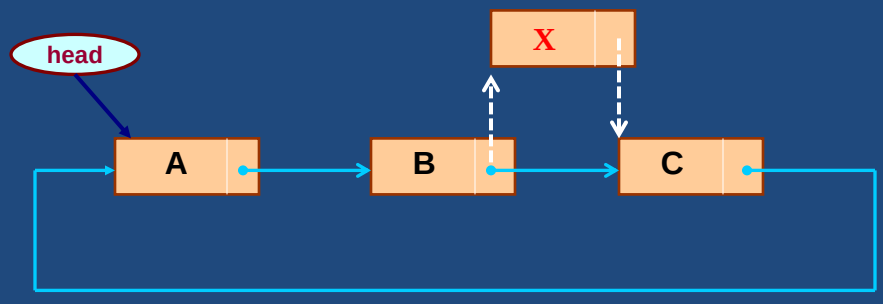



```
struct node {
                int item;
                struct node *prev, *next;
        };
struct node *head, *temp;
```

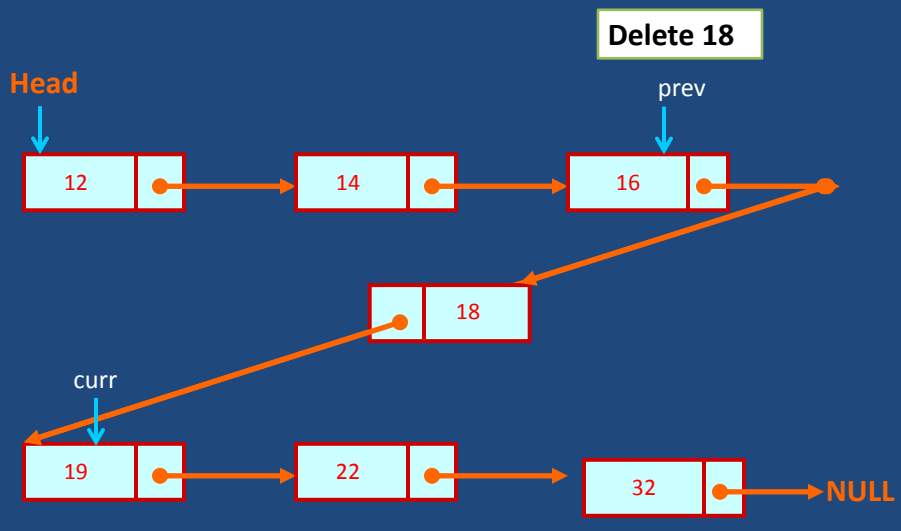
Basic Operations on a List

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

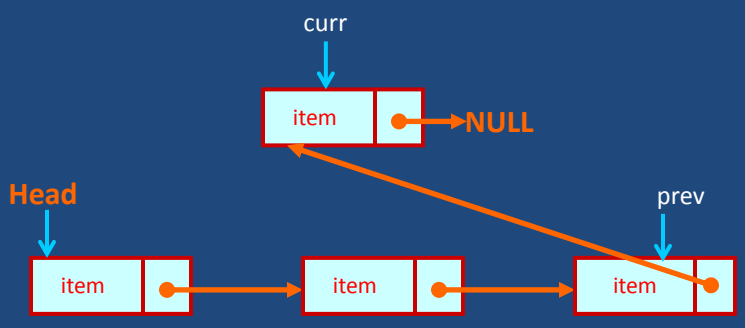
Insert into circular linked list



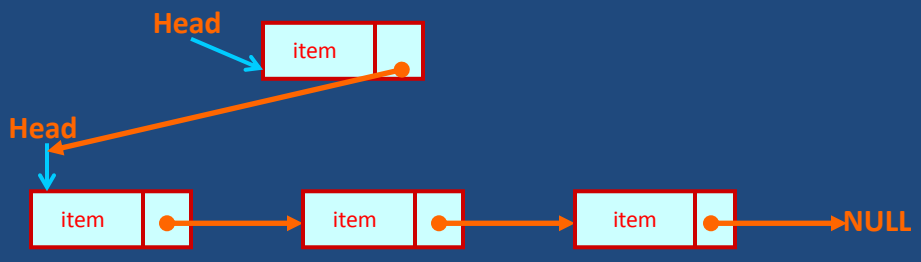
Delete a specific node from linked list



Delete End Node from linked list



Delete Head Node from a linked list



Delete a node and free memory

- Do not forget to free() memory location dynamically allocated for a node after deletion of that node.
- It is programmer's responsibility to free that memory location.
- Failure to do so may create a dangling pointer – a memory location that is not used either by the programmer or by the system.
- The content of a free memory location is not erased.

Concatenating two linked lists