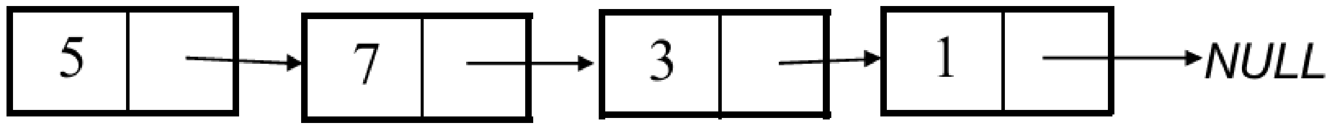
Data Structure and Algorithms-I

Linked Lists

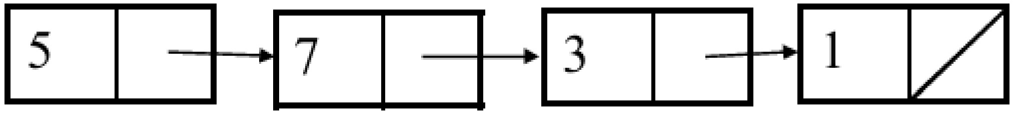


Lecturer Saifur Rahman, Dept. of CSE, United International University

Linked List

- A linked list is a linear collection of data elements (called nodes), where the linear order is given by means of pointers.
- Each node is divided into 2 parts:
  - 1st part contains the information of the element.
  - 2nd part is called the link field or next pointer field which contains the address of the next node in the list.

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

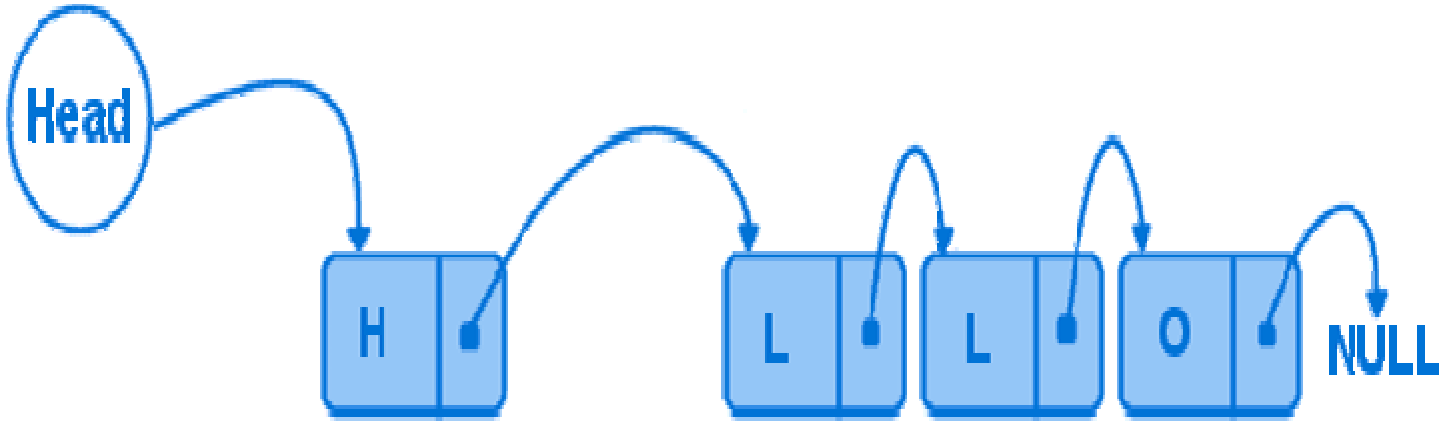


Basic Operations

- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

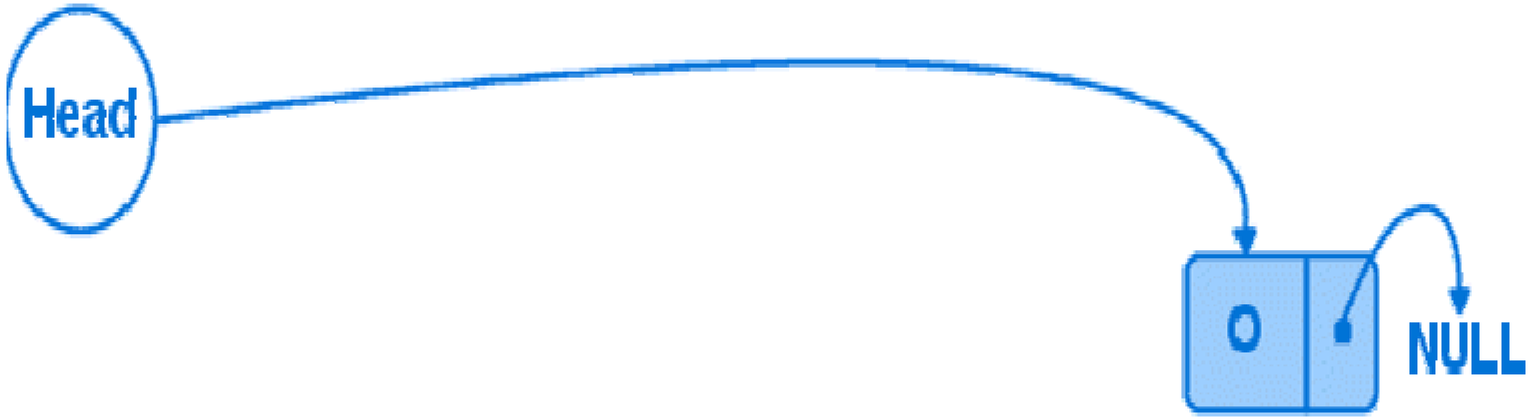
Insertion to a Linear Linked list

- Add a new node at the first, last or interior of a linked list.



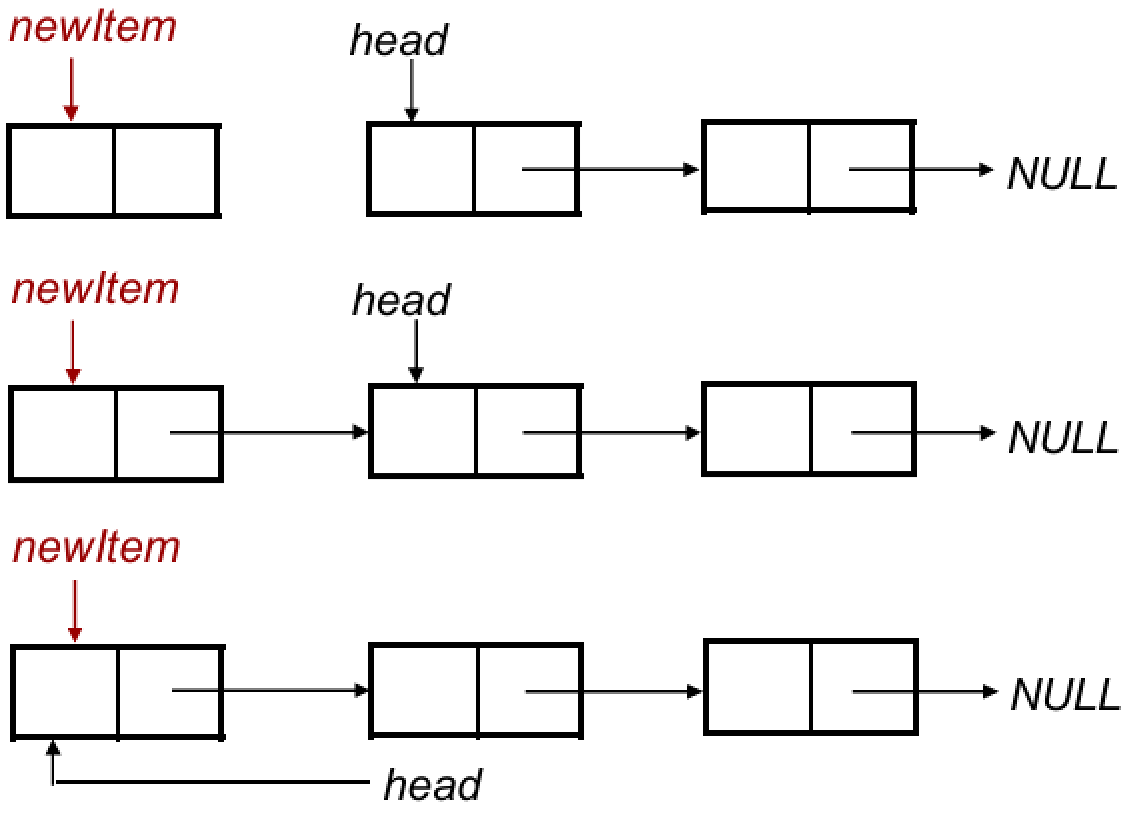
Insert First

- To add a new node to the head of the linear linked list, we need to construct a new node that is pointed by pointer *newitem*.
- Assume there is a global variable *head* which points to the first node in the list.
- The new node points to the first node in the list. The *head* is then set to point to the new node.



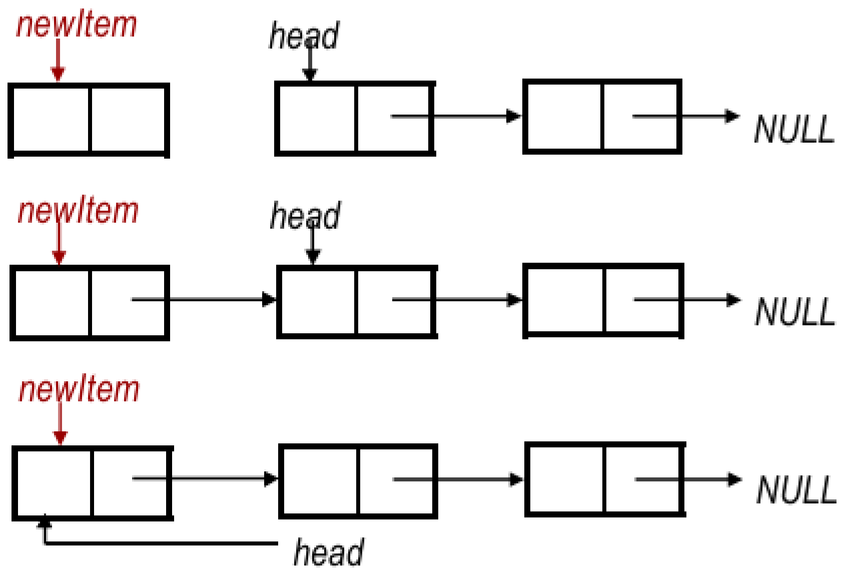
Insert First (Cont.)

- Step 1. Create a new node that is pointed by pointer *newItem*.
- Step 2. Link the new node to the first node of the linked list.
- Step 3. Set the pointer *head* to the new node.



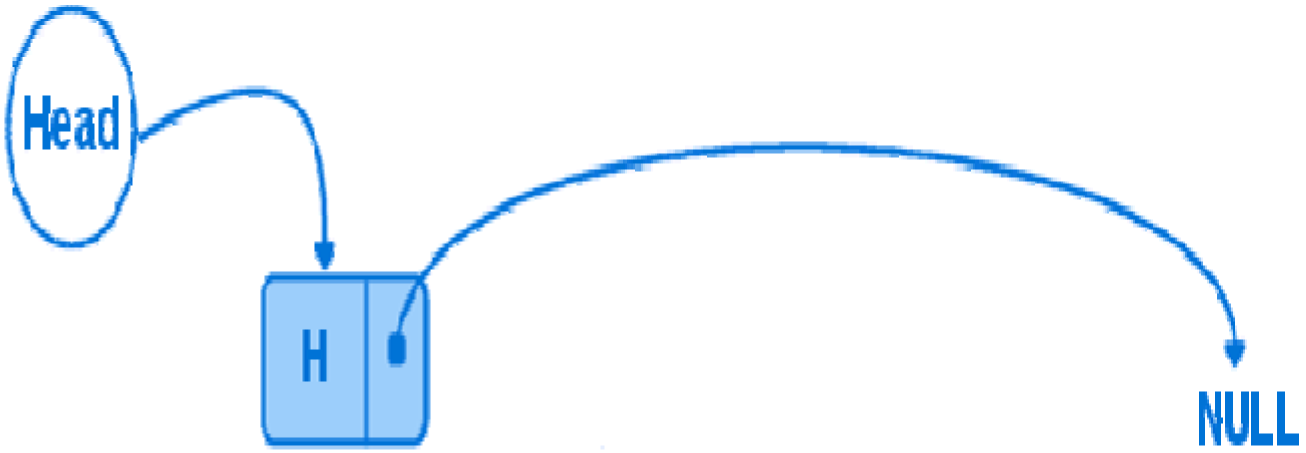
Insert First (Cont.)

```
struct node{
     int value;
     struct node *next;
};
struct node  *head;

void insertHead(){
   //create a new node
   struct node *newItem;
   newItem=(struct node  *)malloc(sizeof(struct node));
   newItem->value = 10;
   newItem->next = NULL;
   //insert the new node at the head
   newItem->next = head;
   head = newItem;
}
```



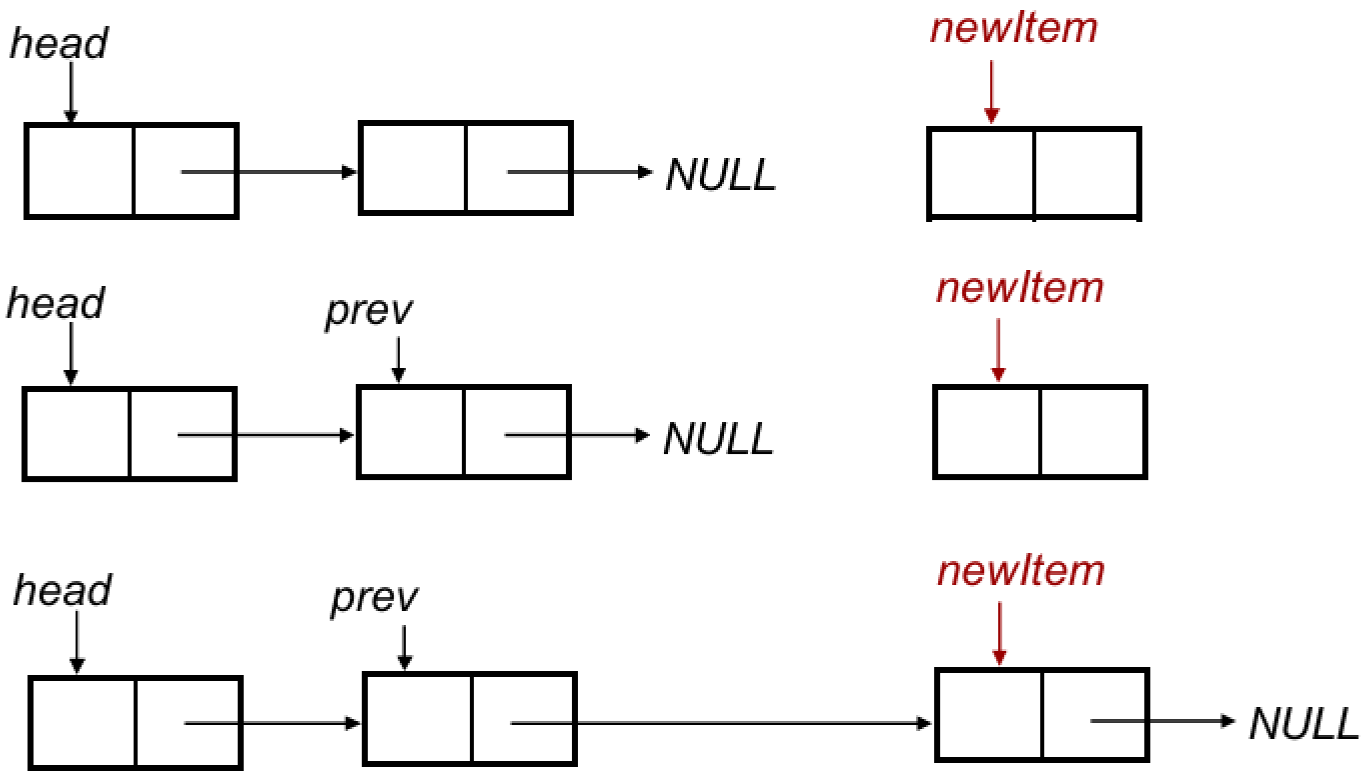
Insert Last

- To add a new node to the tail of the linear linked list, we need to construct a new node and set it's link field to “NULL".
- Assume the list is not empty, locate the last node and change it's link field to point to the new node.



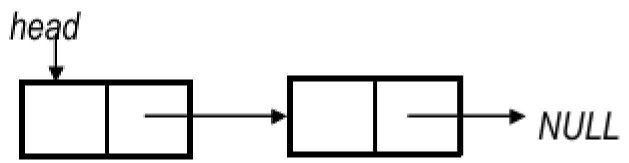
Insert Last (Cont.)

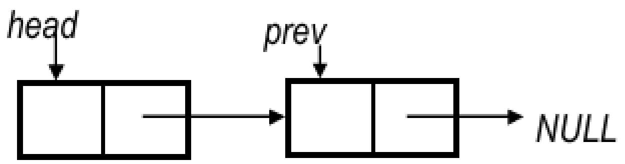
- Step1. Create the new node.
- Step2. Set a temporary pointer prev to point to the last node.
- Step3. Set prev to point to the new node and new node as last node.

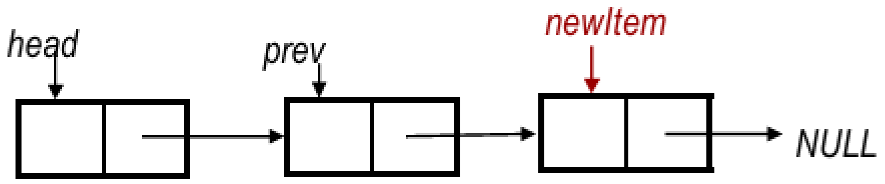


Insert Last (Cont.)

```
struct node{
      int value;
      struct node *next;
};
struct node *head;

void insertTail(){
   //create a new node to be inserted
   struct node *newItem;
   newItem=(struct node *)malloc(sizeof(struct node));
   newItem->value = 10;
   newItem->next = NULL;
   // set prev to point to the last node of the list
   struct node *prev = head;
   while (prev->next != NULL)
       prev = prev->next;
  newItem->next = NULL;
   prev->next = newItem;
}
```

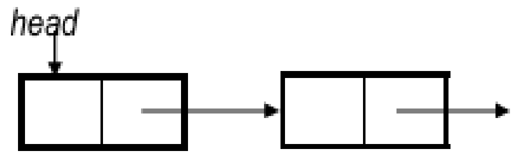


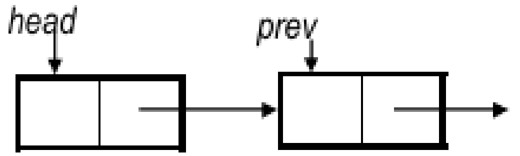


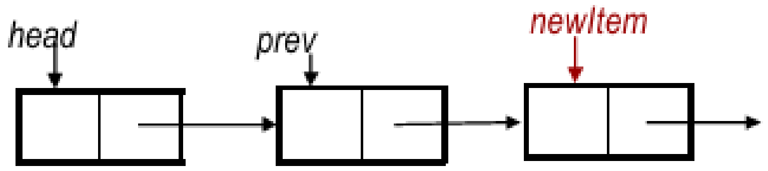


Insert Middle (after a desired node)

```
struct node{
     int value;
     struct node *next;
};
struct node *head;

void insertMiddle(int num){
   //create a new node to be inserted
   struct node *newItem;
   newItem=(struct node *)malloc(sizeof(struct node));
   newItem->value = 10;
   newItem->next = NULL;
   // set prev to point to the desired node of the list
   struct node *prev = head;
   while (prev->value != num){
       prev = prev->next;
   }
   newItem->next = prev->next;
   prev->next = newItem;
}
```





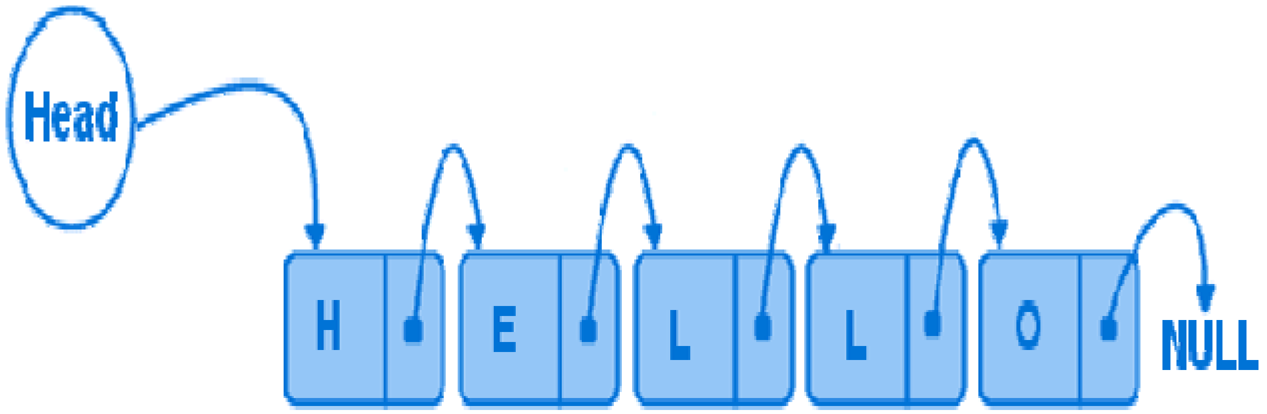


Printing List

```
void printList()
{
 if (head = = NULL)   // no list at all
      return;
 struct node  *cur =  head;
 while (cur != NULL)
  {
      printf("%d \t", cur->value);
    cur = cur->next;
  }
}
```

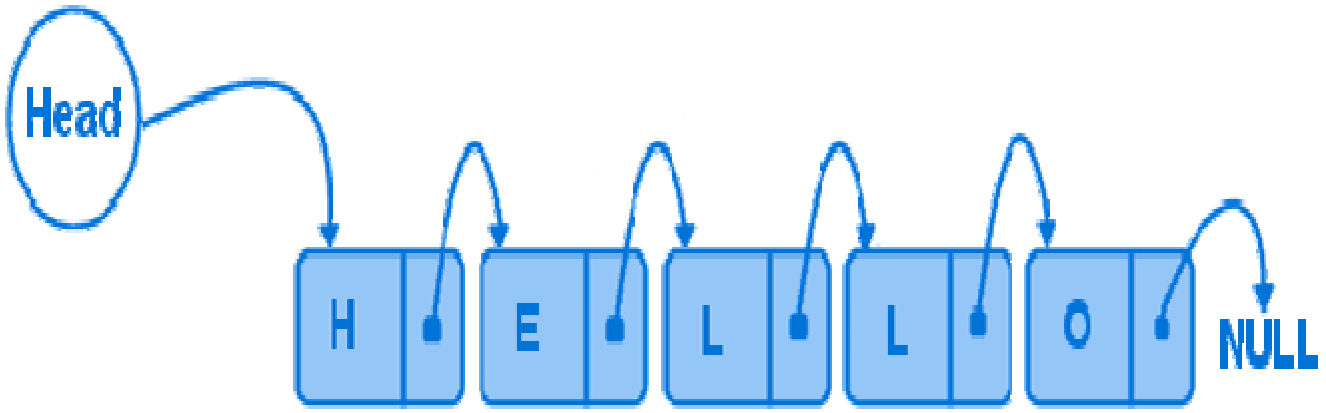
Deletion from a Linear Linked List

- Deletion can be done
  - At the first node of a linked list.
  - At the end of a linked list.
  - Within the linked list.



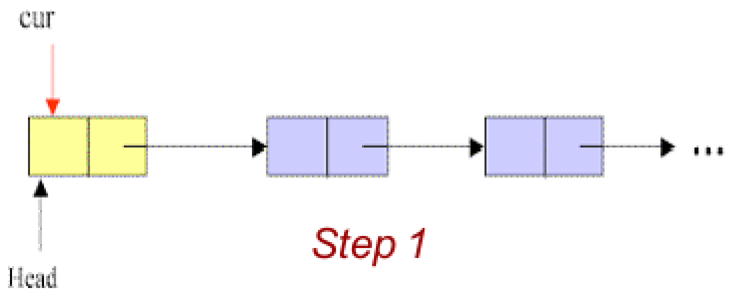
Delete First

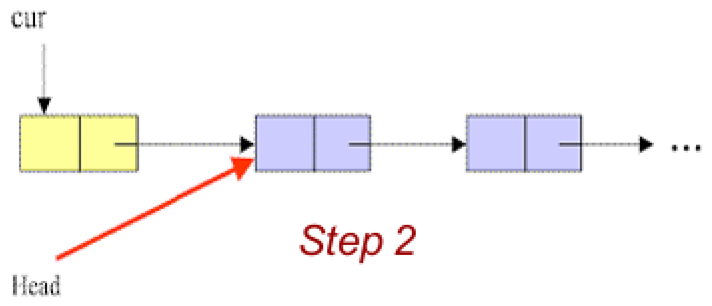
- To delete the first node of the linked list, we not only want to advance the pointer *head* to the second node, but we also want to release the memory occupied by the abandoned node.

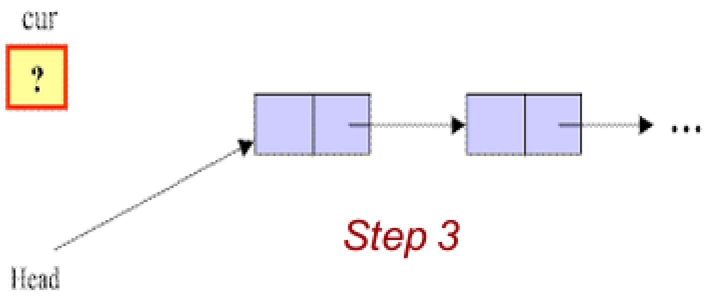


Delete First (Cont.)

- Step1. Initialize the pointer *cur* point to the first node of the list.
- Step2. Move the pointer *head* to the second node of the list.
- Step3. Remove the node that is pointed by the pointer *cur*.

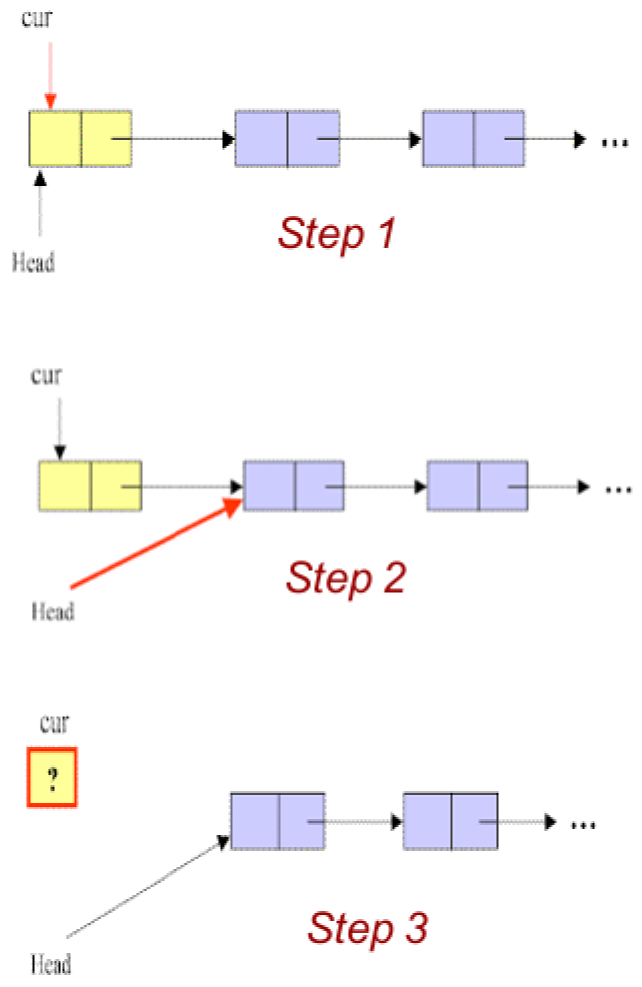






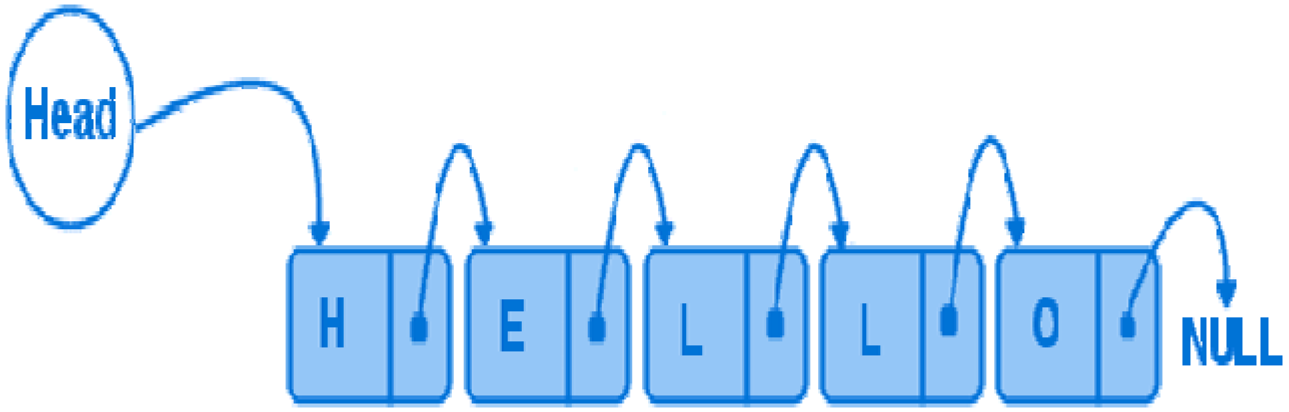
Delete First (Cont.)

```
void deleteHead()
{
  struct node *cur;
  if (head = = NULL)  //list empty
      return;
  cur = head;  // save head pointer
  head = head->next; //advance head
  free(cur);
}
```



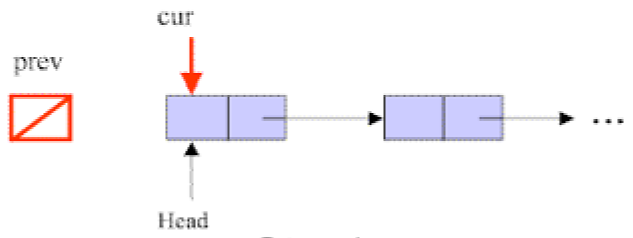
Delete Last

- To **delete** the last node in a linked list, we use a local variable, *cur*, to point to the last node. We also use another variable, *prev*, to point to the second last node in the linked list.

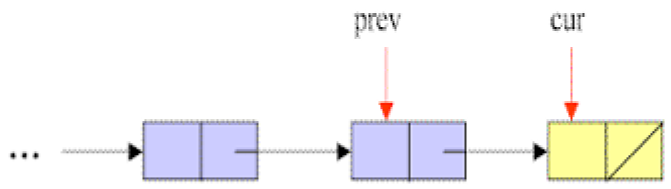


Delete Last (Cont.)

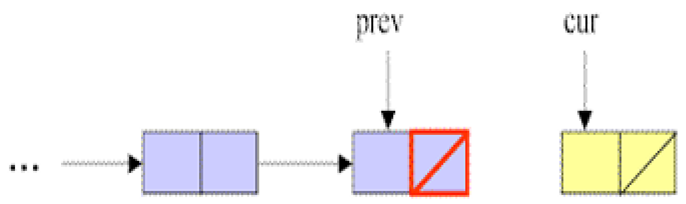
- Step1. Initialize pointer *cur* to point to the first node of the list, while the pointer *prev* has a value of NULL.
- Step2. Traverse the entire list until the pointer *cur* points to the last node of the list.
- Step3. Set NULL to *next* field of the node pointed by the pointer *prev*.
- Step4. Remove the last node that is pointed by the pointer *cur*.



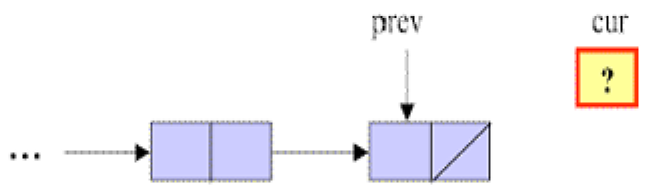
Step1



Step2



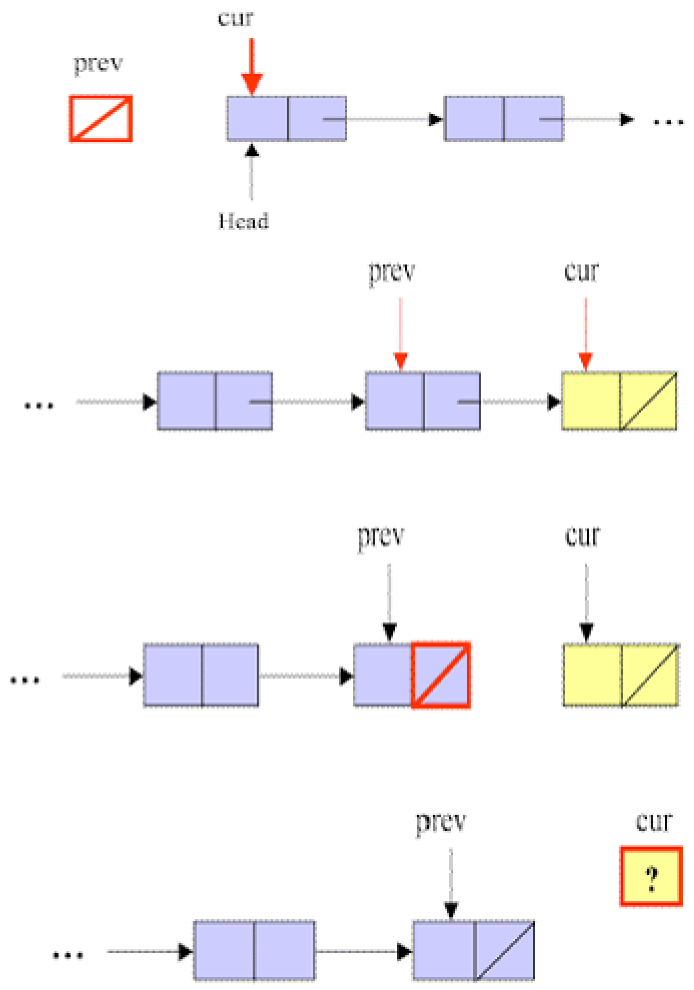
Step3



Step4

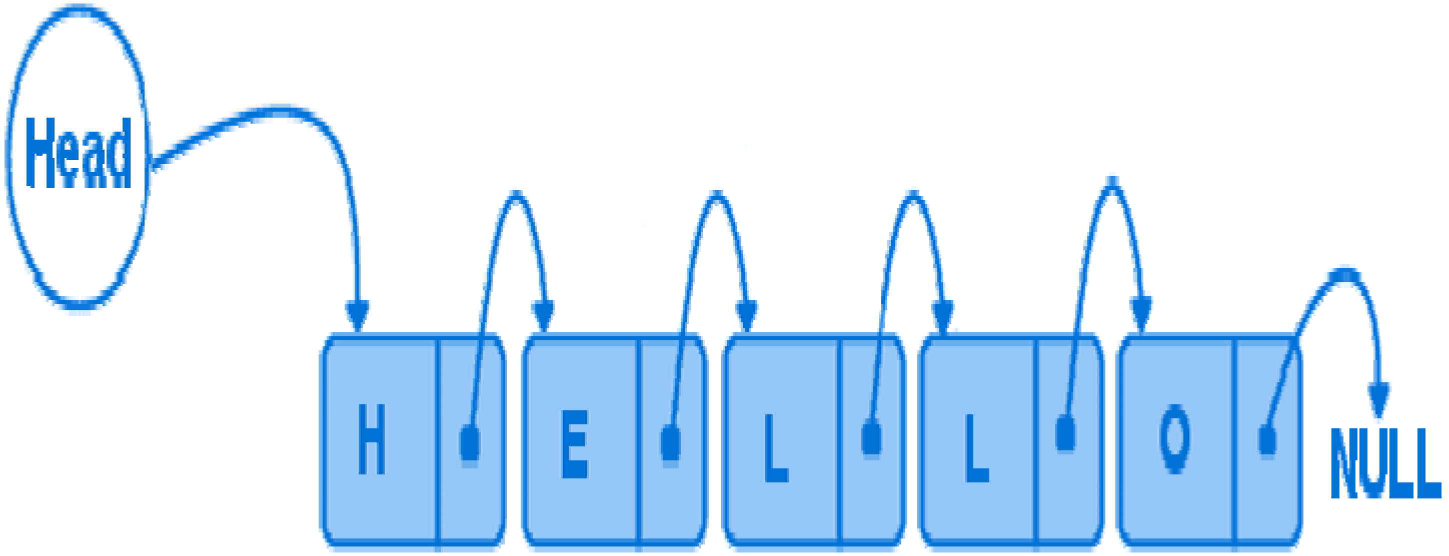
Delete Last (Cont.)

```
void deleteTail(){
  if (head = = NULL)  //list empty
      return;
  struct node *cur = head;
  struct node *prev = NULL;
  while (cur->next != NULL){
      prev = cur;
      cur=cur->next;
  }
  if (prev != NULL)
      prev->next = NULL;
  free(cur);
}
```



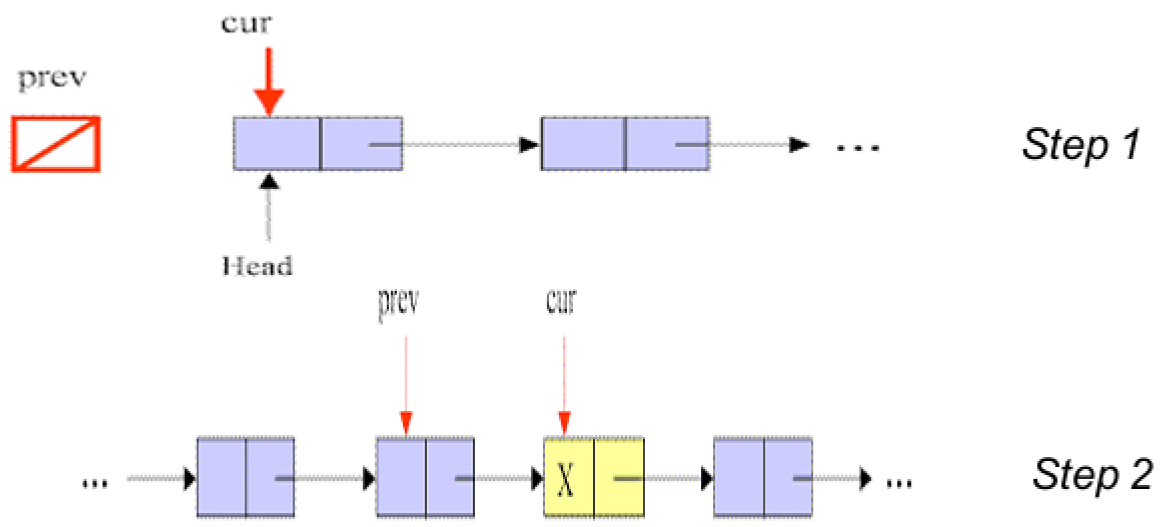
Delete Any

- *To* ***delete*** *a node that contains a particular value x in a linked list, we use a local variable, cur, to point to this node, and another variable, prev, to hold the previous node.*



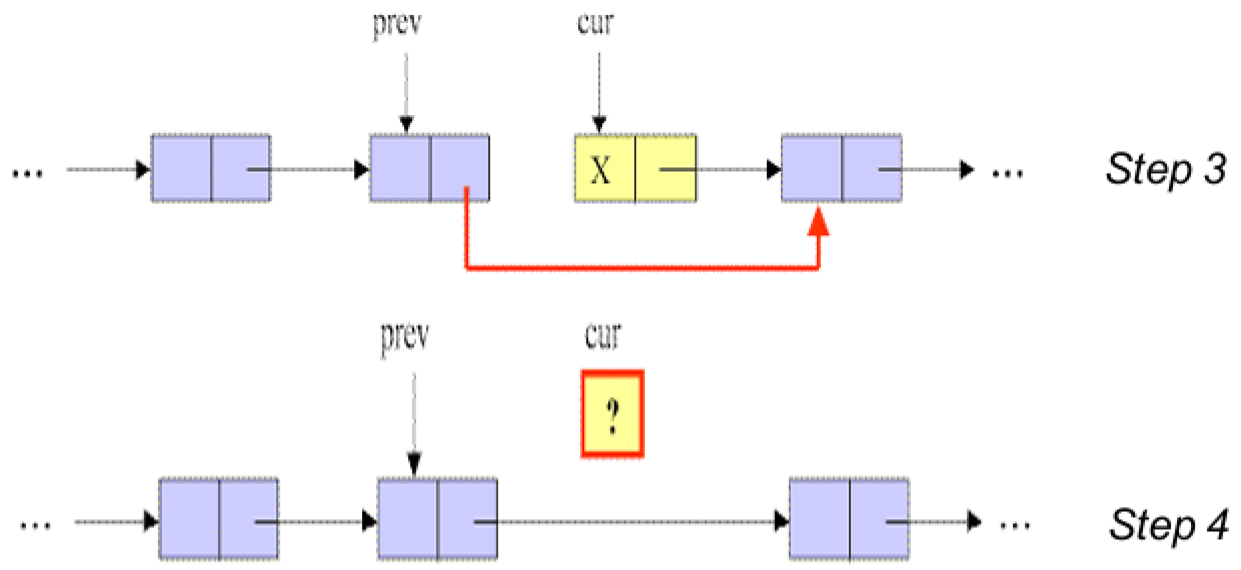
Delete Any (Cont.)

- Step1. Initialize pointer *cur* to point to the first node of the list, while the pointer *prev* has a value of null.
- Step2. Traverse the entire list until the pointer *cur* points to the node that contains value of *x*, and *prev* points to the previous node.
- ……..



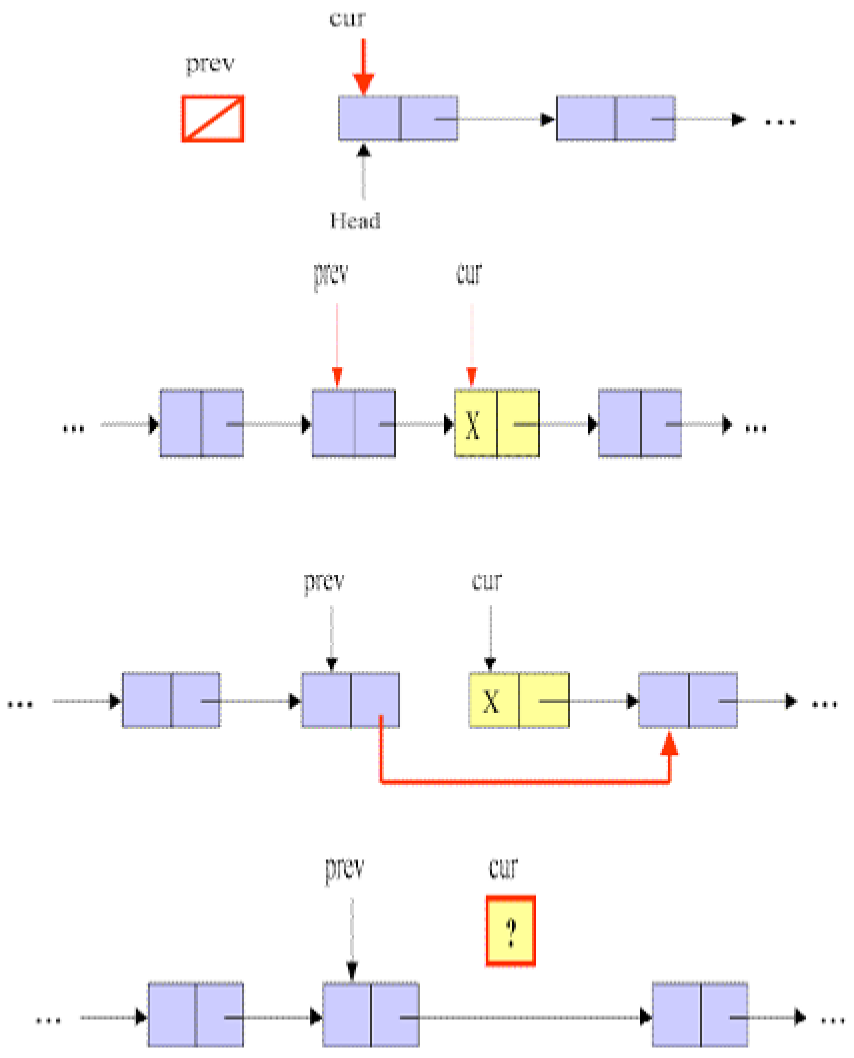
Delete Any (Cont.)

- …….
- Step3. Link the node pointed by pointer *prev* to the node after the *cur*'s node.
- Step4. Remove the node pointed by *cur*.



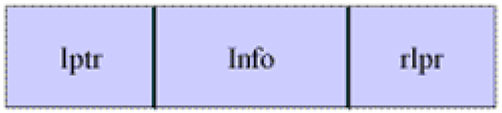
Delete Any (Cont.)

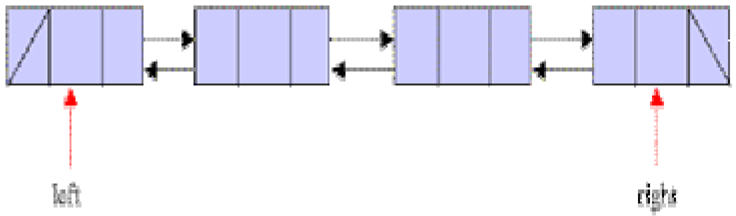
```
void deleteAny( int x ){
  if (head = = NULL)   //list empty
       return;
  struct node  *cur = head;
  struct node  *prev = NULL;
  while (cur->value != x){
       prev = cur;
       cur=cur->next;
  }
 if (prev != NULL)
       prev->next = cur->next;
 free(cur);
}
```



Introduction to Doubly Linked List

- We have discussed the details of linear linked list. In the linear linked list, we can only traverse the linked list in one direction.
- But sometimes, it is very desirable to traverse a linked list in either a forward or reverse manner.
- This property of a linked list implies that each node must contain two link fields instead of one. The links are used to denote the predecessor and successor of a node. The link denoting the predecessor of a node is called the left link, and that denoting its successor its right link





Basic Operation of Doubly Linked List

- Insert: Add a new node in the first, last or interior of the list.
- Delete: Delete a node from the first, last or interior of the list.
- Search: Search a node containing particular value in the linked list.

Insert First or Insert Last into a DLL

- **Insertion** is to add a new node into a linked list. It can take place anywhere -- the first, last, or interior of the linked list.

- To add a new node to the head and tail of a double linked list is similar to the linear linked list.
- First, we need to construct a new node that is pointed by pointer *newItem*.
- Then the newItem is linked to the left-most node (or right-most node) in the list. Finally, the *Left* (or *Right*) is set to point to the new node.

Insert Interior of Doubly Linked List

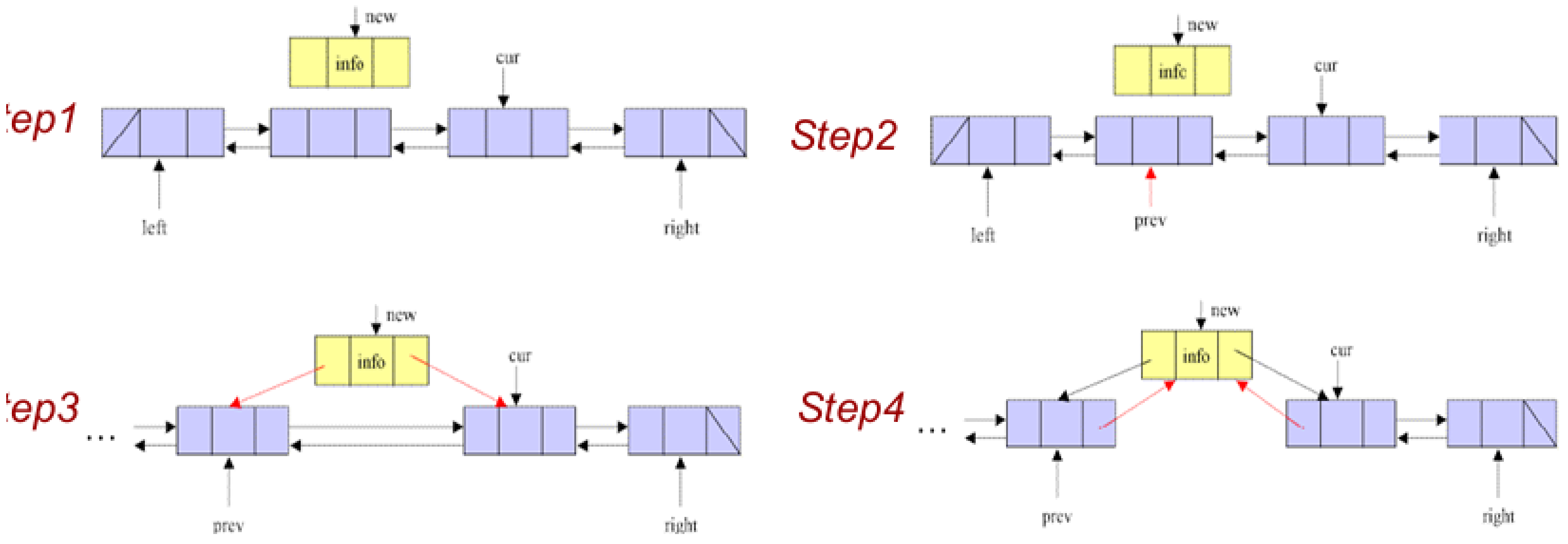
Insert a node before the node pointed by cur

Step1. *Create a new node that is pointed by new*

Step2. *Set the pointer prev to point to the left node of the node pointed by cur.*

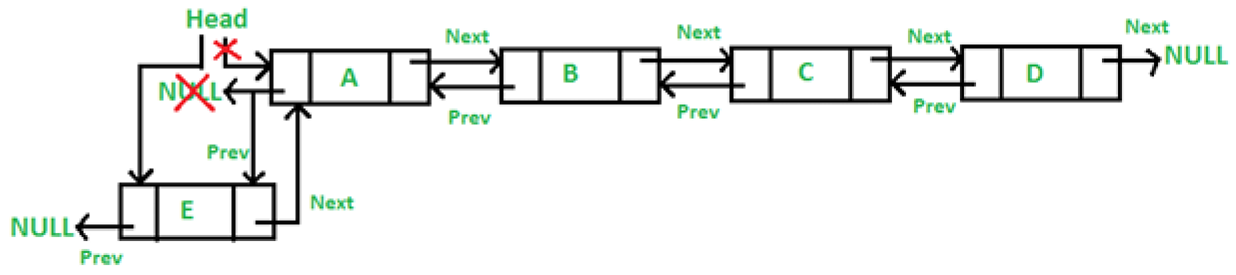
Step3. *Set the left link of the new node to point to the node pointed by prev, and the right link of the new node to point to the node pointed by cur.*

Step4. *Set the right link of the node pointed by prev and the left link of the node pointed by cur to point to the new node.*



Insert (First) into a Doubly Linked List

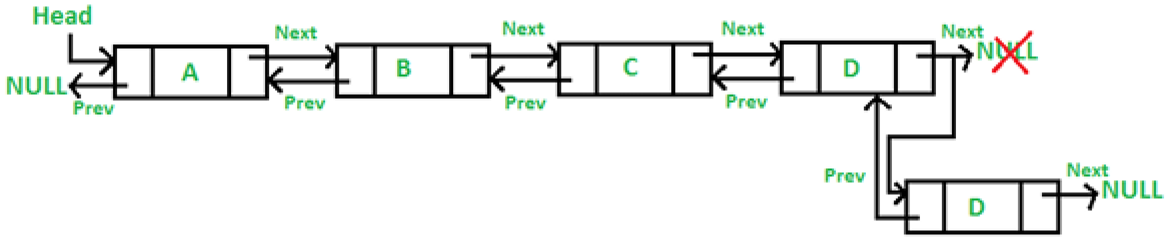
```
struct dnode{
    struct dnode *prev;
    int value;
    struct dnode *next;
}*head, *last;
```



```
void insert_begning(int data){
    struct dnode *newItem;
    newItem=(struct dnode *)malloc(sizeof(struct dnode));
    newItem->value=data;
    if(head==NULL){
        head=newItem;        head->prev=NULL;
        head->next=NULL;     last=head;
    }
    else{
        newItem->prev=NULL;          newItem->next=head;
        head->prev=newItem;          head=newItem;
    }
}
```

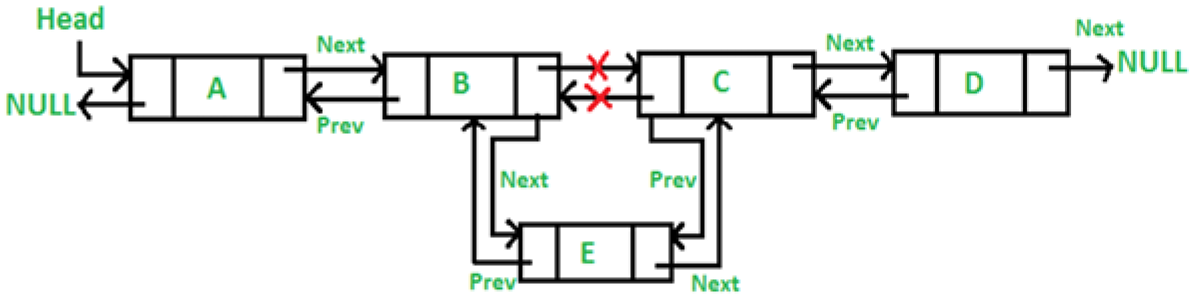
Insert (Last) into a Doubly Linked List

```
void insert_end(int data){
    struct dnode *newItem,*temp;
    newItem=(struct dnode *)malloc(sizeof(struct dnode));
    newItem->value=data;

    if(head==NULL){
        head=newItem;          head->prev=NULL;
        head->next=NULL;       last=head; }

    else{
        last=head;
        while(last != NULL){
            temp=last;
            last=last->next;
        }
    last=newItem;              temp->next=last;
    last->prev=temp;           last->next=NULL;
    }
}
```



Insert (Middle) into a Doubly Linked List

```
int insert_after(int data, int x){    \\ Insert after node x
    struct dnode *temp,*newItem,*temp1;
    newItem=(struct dnode *)malloc(sizeof(struct dnode));
    newItem->value=data;
      if(head==NULL){
        head=newItem;  head->prev=NULL;   head->next=NULL;  }
    else{
        temp=head;
        while(temp!=NULL && temp->value!=x)
             temp=temp->next;
        if (temp==NULL)
            printf("\n %d is not present in the list ", x);
        else{
        temp1=temp->next;  newItem->prev=temp; newItem->next=temp1;
        temp1->prev=newItem; temp->next=newItem; }
    }
    last=head;
    while(last->next!=NULL)
        last=last->next;
}
```

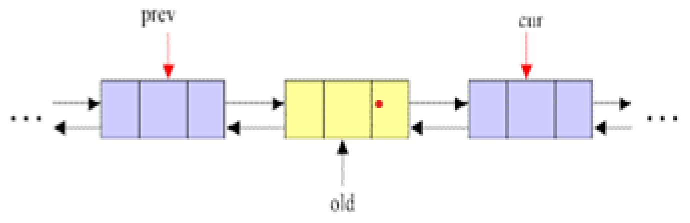


Deletion of Doubly Linked List

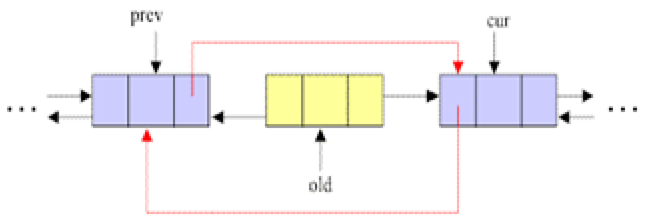
- **Deletion** is to remove a node from a list. It can also take place anywhere -- the first, last, or interior of a linked list.
- To delete a node from a double linked list is easier than to delete a node from a linear linked list.
- For deletion of a node in a single linked list, we have to search and find the predecessor of the discarded node. But in the double linked list, no such search is required.
- Given the address of the node that is to be deleted, the predecessor and successor nodes are immediately known.

Deletion of Doubly Linked List (Cont.)

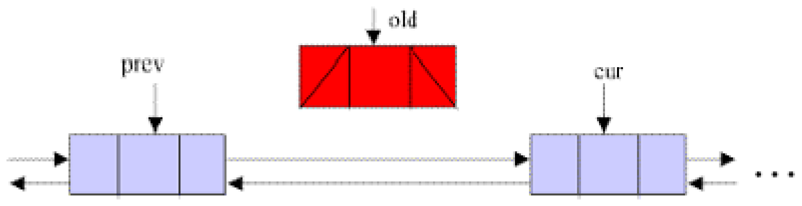
- Step1. Set pointer *prev* to point to the left node of *old* and pointer *cur* to point to the node on the right of *old*.
- Step2. Set the right link of *prev* to *cur*, and the left link of *cur* to *prev*.
- Step3. Discard the node pointed by *old*.



step1



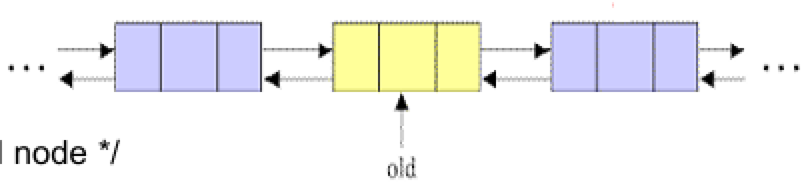
step2

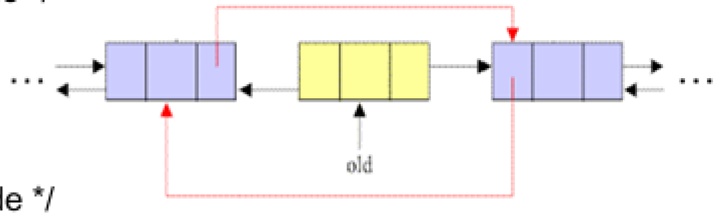


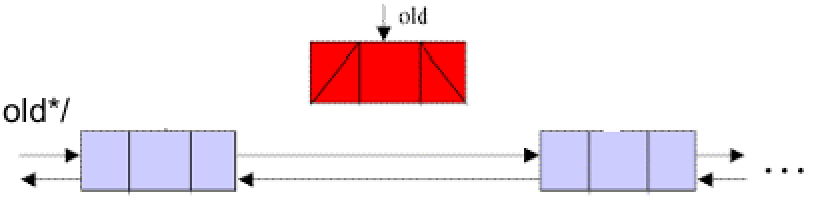
step3

Deletion of Doubly Linked List (Cont.)

```
struct dnode {
    struct dnode *prev;
    int value;
    struct dnode *next;
    };

void deleteNode (struct dnode *old) {
    if(head == old)      /* If node to be deleted is head node */
        head = old->next;

    /* Change next only if node to be deleted is not the last node */
    if(old->next != NULL)
        old->next->prev = old->prev;

    /* Change prev only if node to be deleted is not the first node */
    if(old->prev != NULL)
        old->prev->next = old->next;

    free(old);   /* Finally, free the memory occupied by old*/
    return;
}
```







Advantages/Disadvantages of DLL

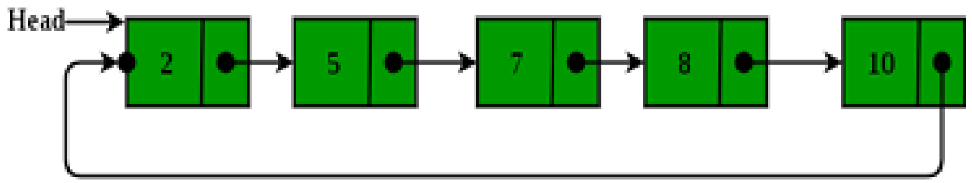
- **Advantages over singly linked list:**
  - A DLL can be traversed in both forward and backward direction.
  - We can quickly insert a new node before a given node.
  - The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
    - In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using 'prev' pointer.

- **Disadvantages over singly linked list:**
  - Every node of DLL require extra space for an previous pointer.
  - All operations require an extra pointer 'prev' to be maintained.

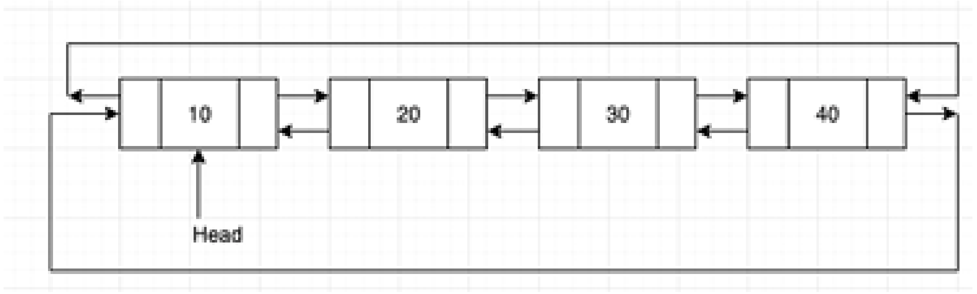
Circular Linked List

- In a circular linked list every element has a link to its next element and the last element has a link to the first element.
- That means circular linked list is similar to the single linked list except that the last node points to the first node in the list. There is no NULL at the end.
- A circular linked list can be a singly circular linked list or doubly circular linked list.

Singly circular linked list



Doubly circular linked list



Summary on Linked Lists

- **Advantages:**
  - Linked List is dynamic data structure, that is, the Linked List grows dynamically.
  - Insertion into Linked Lists and deletion from Linked Lists are very fast ($O(1)$ time).
- **Disadvantages:**
  - Linked List is a sequential access data structure.
  - Accessing an element by pointers is very slow ($O(n)$ time)
- A Linked List is a suitable structure when
  - a lot of insertions and deletions are required.
  - a small number of searching and retrieval are required.