Data Structure

Linked List

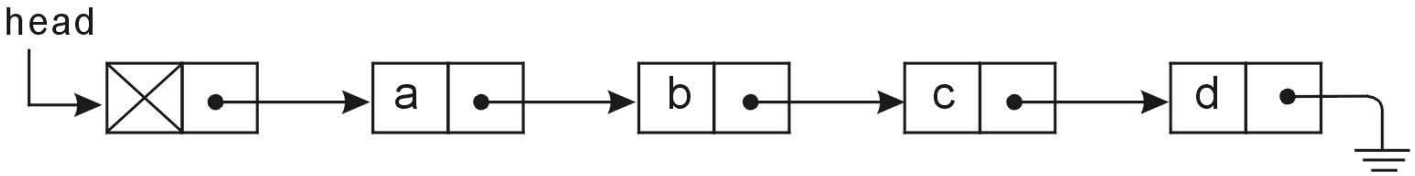
25-Apr-23

Linked list

- Linked list are composed of many nodes and are much easier to add and remove than arrays.
- Linked list can be classified as singly linked list, circular linked list and doubly linked list.

Singly Linked List

- If the list A = {a, b, c, d}, the data structure is as follows:

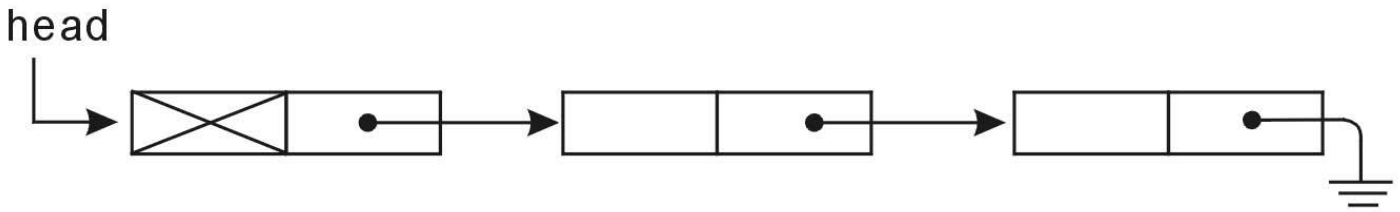



```
struct node{
    int data;
    struct node *next;
};
```

Singly Linked List

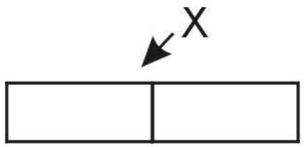
- Adding Action
- Add to the front of the list:

Suppose there is a list as follows:



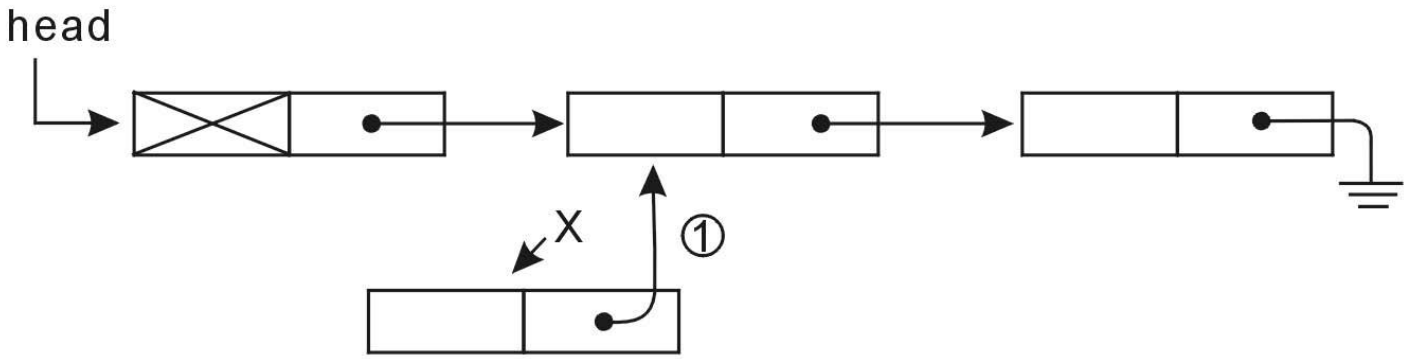
Singly Linked List

- One node x will be added to the front of the list column as follows:
  - (1) x=(struct node*) malloc(sizeof(struct node));



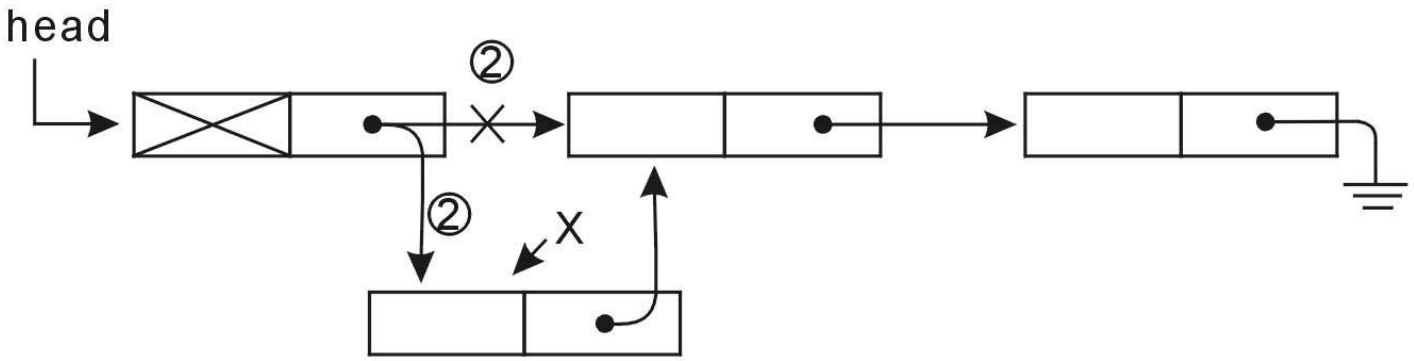
Singly Linked List

- (2)(x->next=head->next; /* ① */



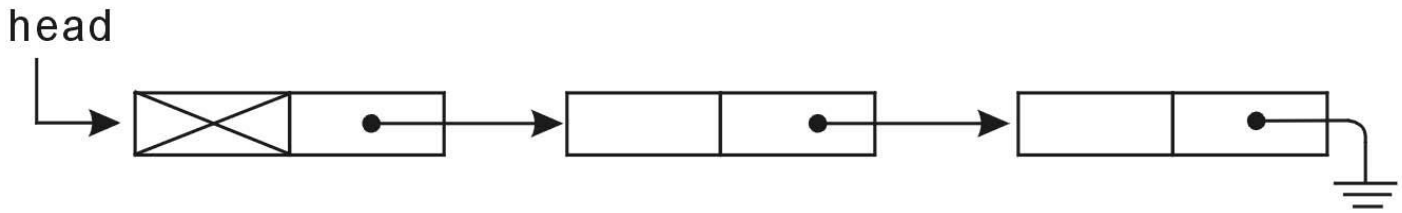
Singly Linked List

- (3)head->next=x;　　/* ② */



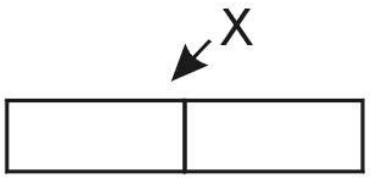
Singly Linked List

- Add to the end of the list :
- Suppose there is a list as follows:

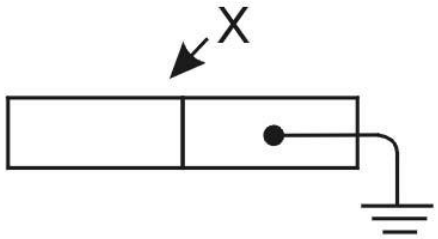


Singly Linked List

- Add a node x to the end of the list as follows:
  - (1) x=(struct node *)malloc(sizeof(struct node));



  - (2) x->next=NULL;

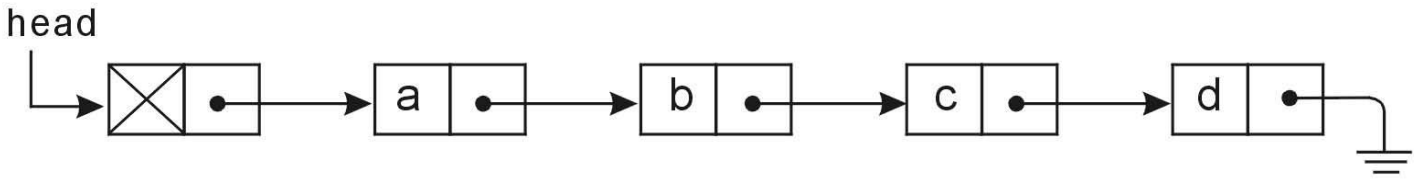


Singly Linked List

- In this case, you have to trace the end of the list, using the following fragment program

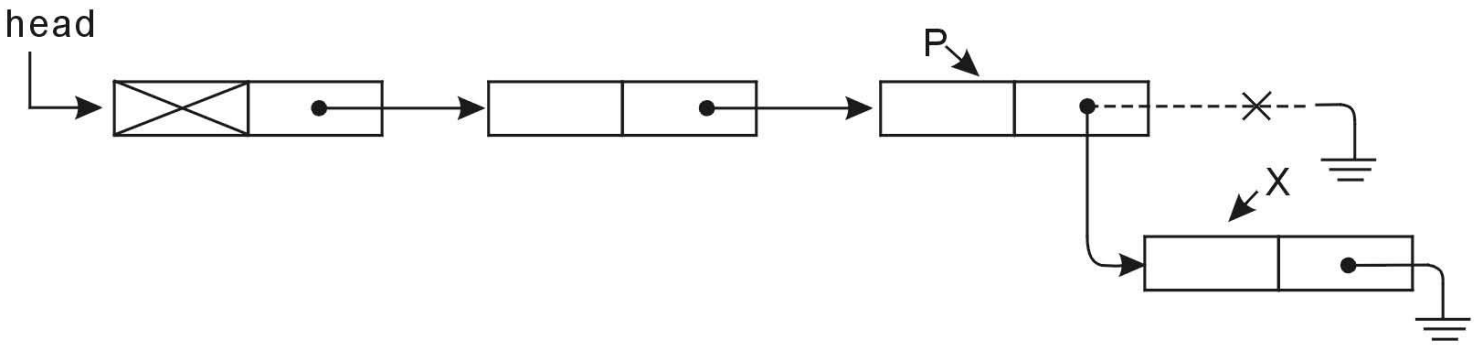
```
P=head->next;
while(p->next != NULL)
    p=p->next;
```

- Then the end of the series can be found.



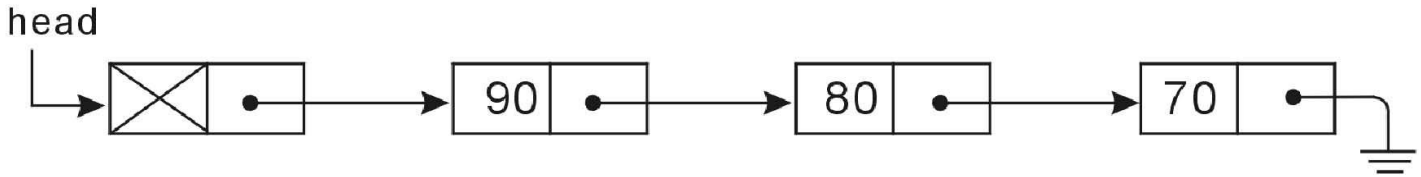
Singly Linked List

- (3)p->next=x;



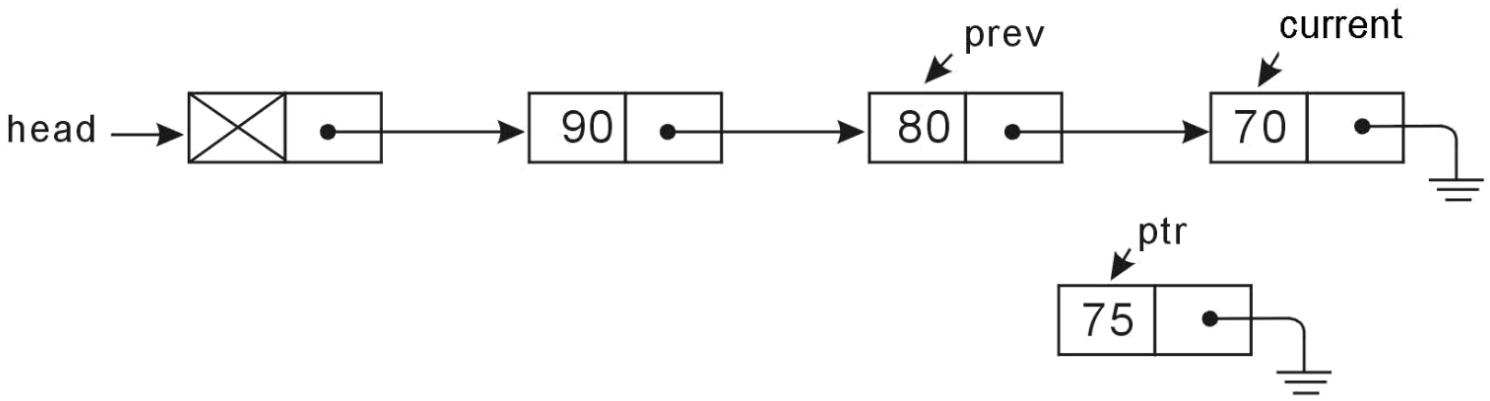
Singly Linked List

- Add a unidirectional linked list after a specific node in the list, and arrange them from largest to smallest by data column.



Singly Linked List

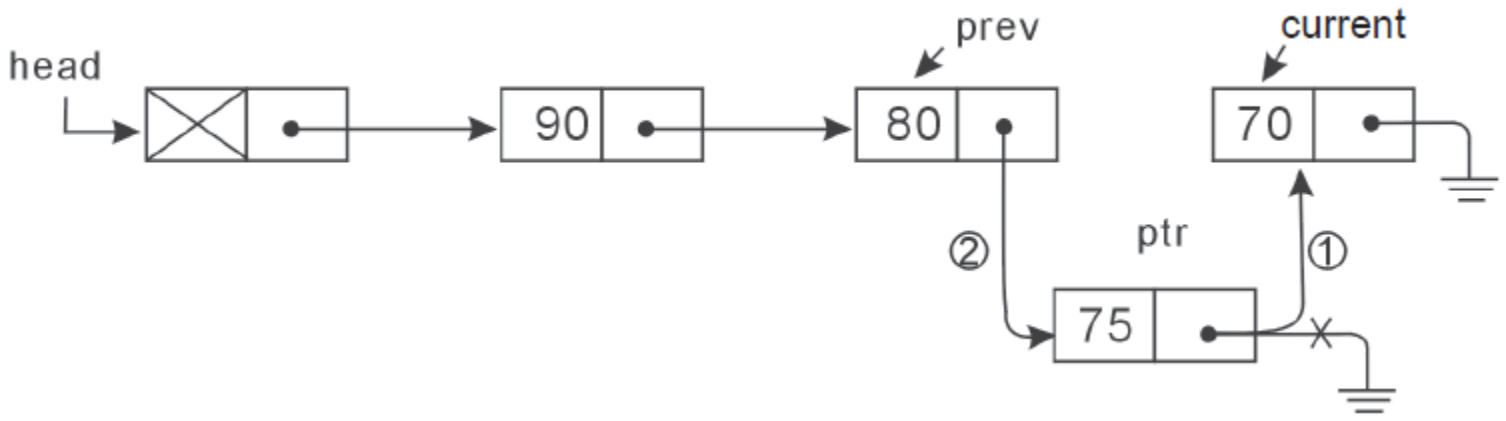
```
prev=head;
current=head->next;

while(current != NULL && current->data > ptr->data)
{
        prev=current;
        current=current->next;
}
```



Singly Linked List

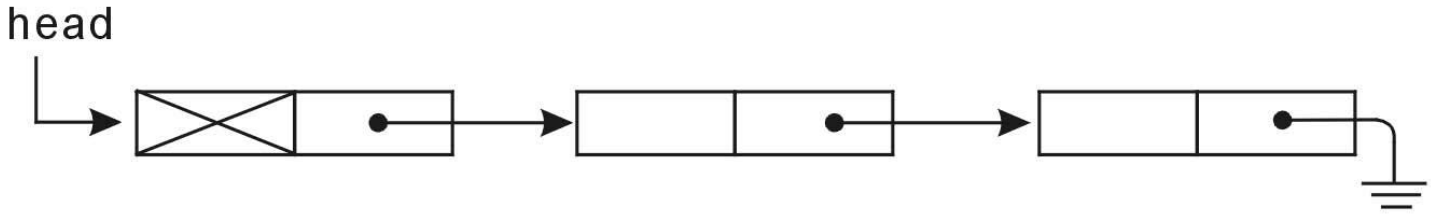
- The next action: is to insert the ptr pointing to the node after the prev.

```
ptr->next=current;          /*  ①  */
prev->next=ptr;             /*  ②  */
```



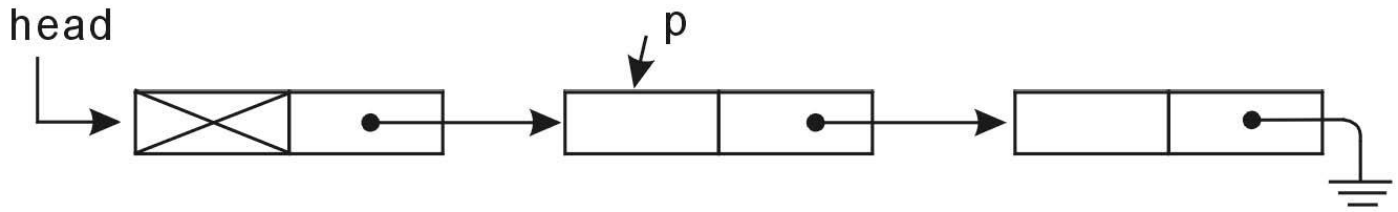
Singly Linked List

- Delete Action
- Delete the front node of the list assuming that there is a list as follows:

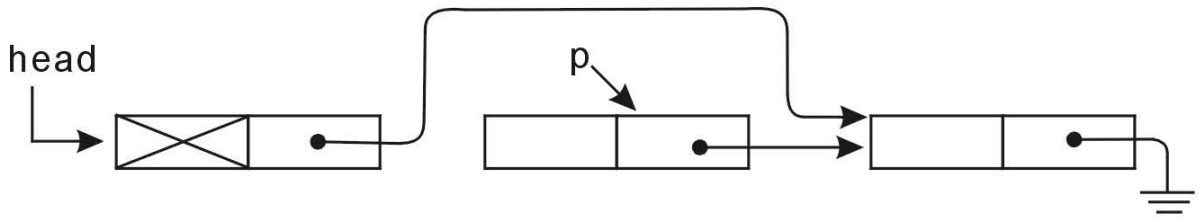


Singly Linked List

- This can be accomplished in a few steps:
  - (1)p=head->next;

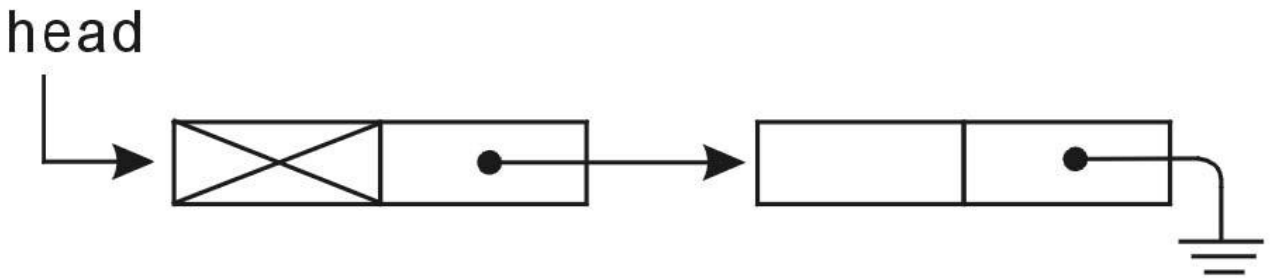


  - (2)head->next=p->next;



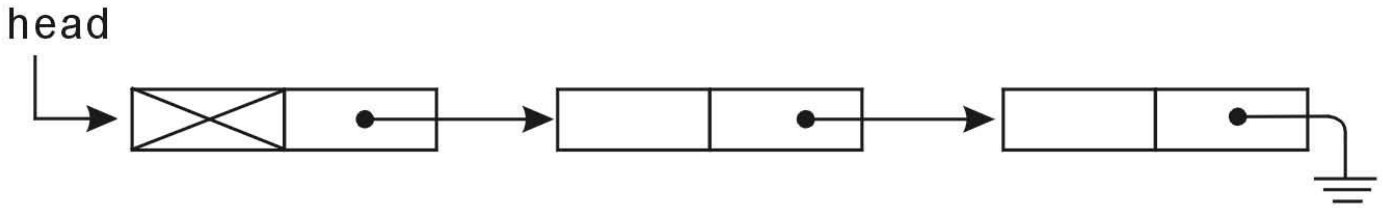
Singly Linked List

- (3)free(p);



Singly Linked List

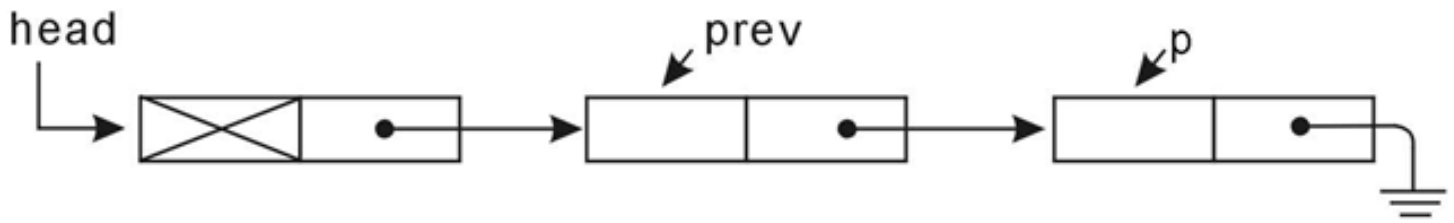
- Delete the last node of the list:



Singly Linked List

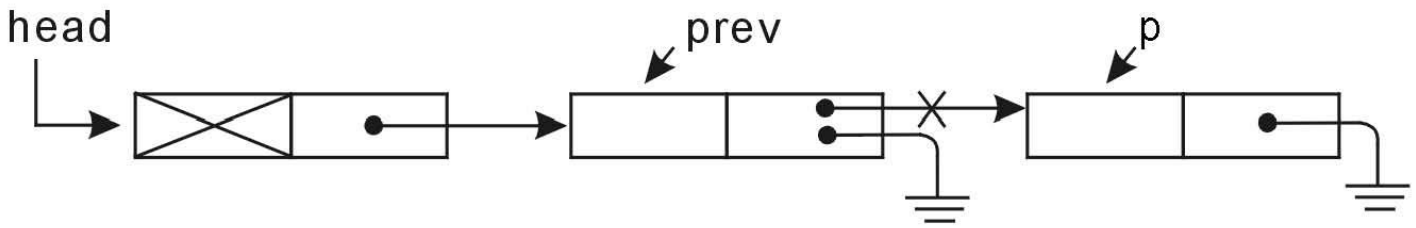
- (1) p=head->next;

```
while(p->next != NULL){      /*  Find the end of the previous node prev  */
   prev=p;
   p=p->next;
}
```

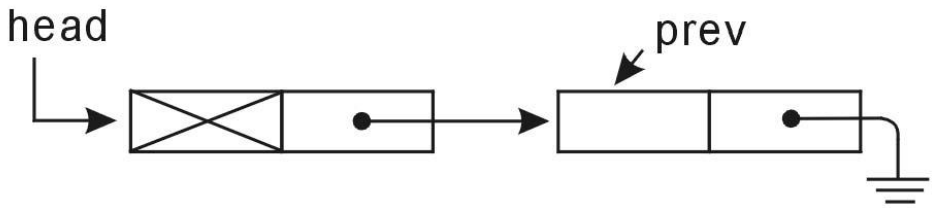


Singly Linked List

- (2) prev->next=NULL;

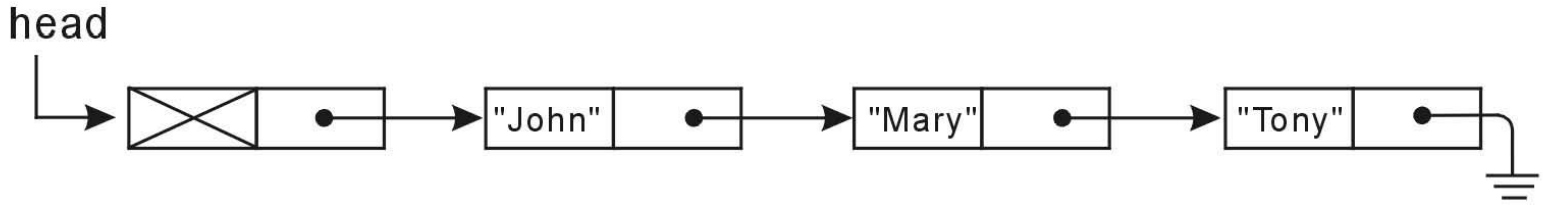


- (3) free(p);



Singly Linked List

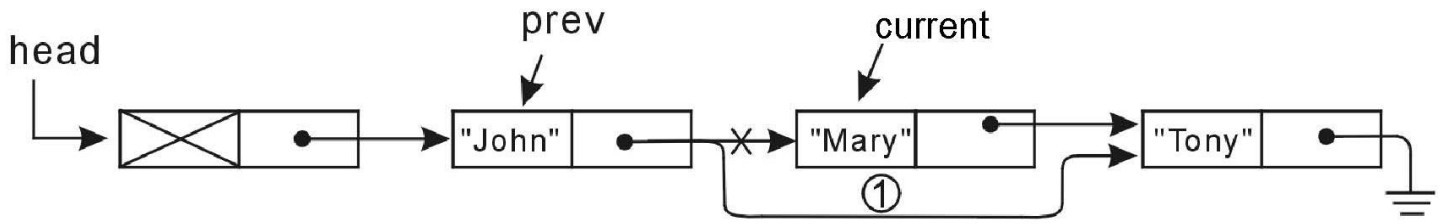
- Deleting a particular node:



Singly Linked List

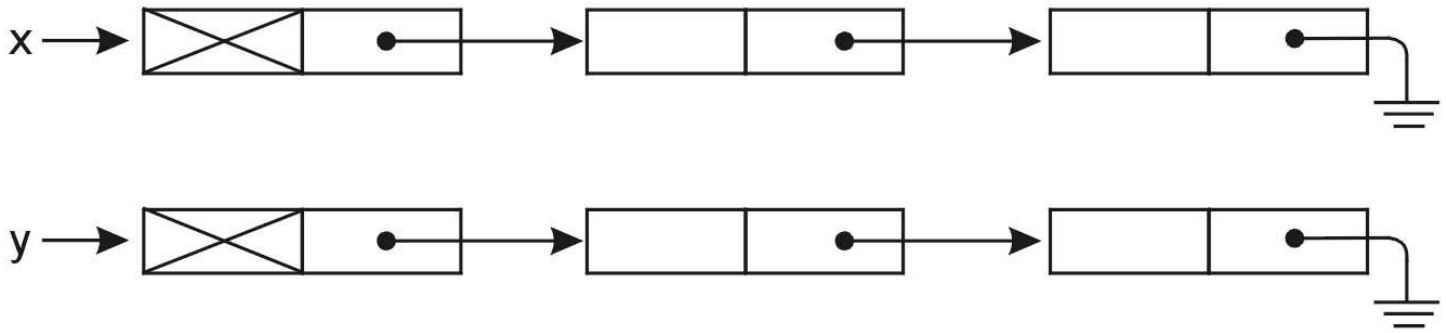
```
prev=head;
current=head->next;
while(current != NULL && strcmp(current->data, del_data)!= 0){
    prev=current;
    current=current->next;
}
if (current != NULL) {    /* Delete the current node */
    prev->next=current->next;    /*  ①   */
    free(current);
}
else
    printf("the data is not found");
```

Strcmp returns 0 for string equality



Singly Linked List

- Connecting two unidirectional links to each other
- Suppose there are two lists as follows:

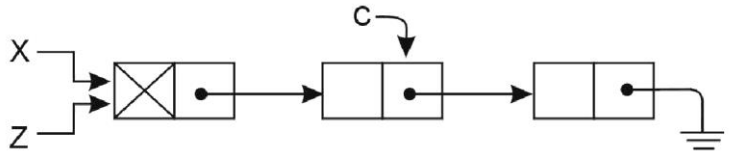


Singly Linked List

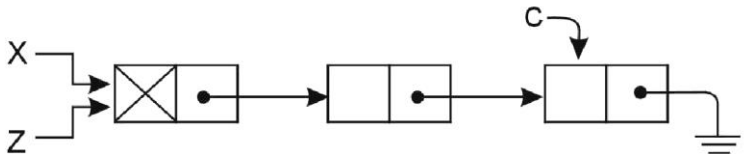
- To merge x and y list into z list, the steps are as follows:
  - if(x->next==NULL)
    z=y;
  - if(y->next ==NULL)
    z=x;

Singly Linked List

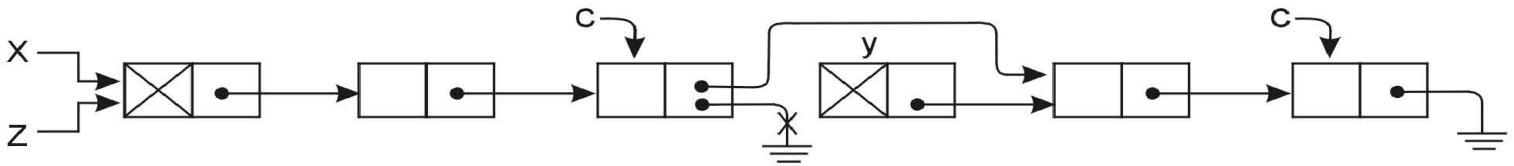
- z=x;
  c=x->next;




```
while(c->next != NULL)
        c=c->next;
```



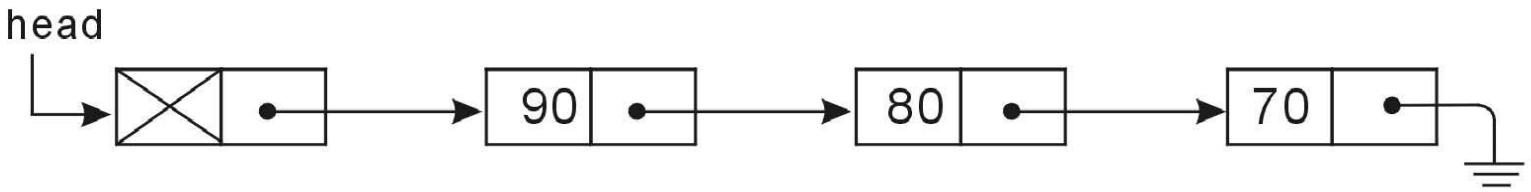

```
c->next=y->next;
```




```
free(y);
```

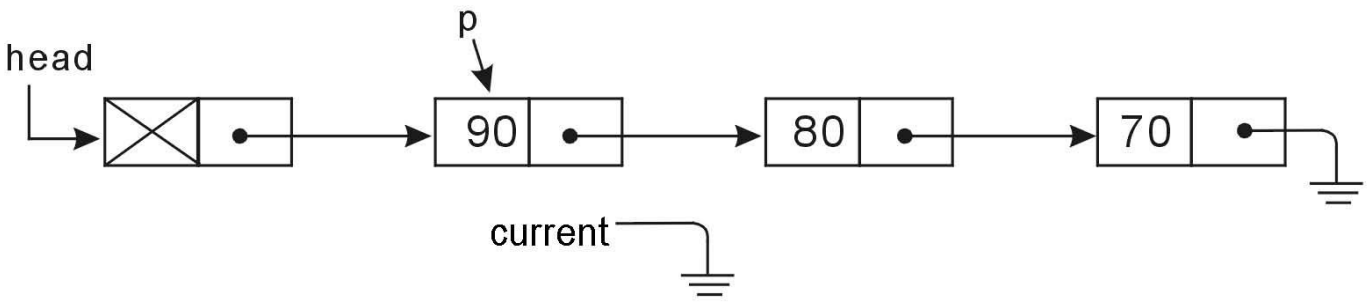
Singly Linked List

- Inverting a list



Singly Linked List

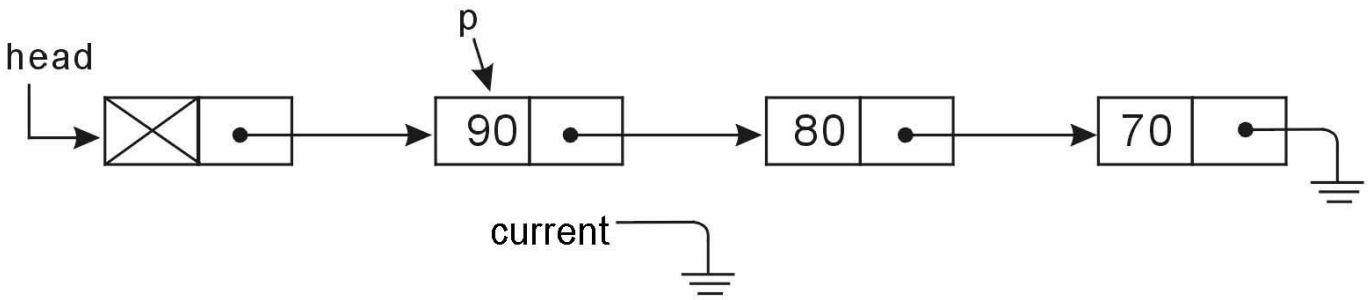
- The reversal can be accomplished in the following steps:
  - (1) p=head->next;
  - (2) current=NULL;



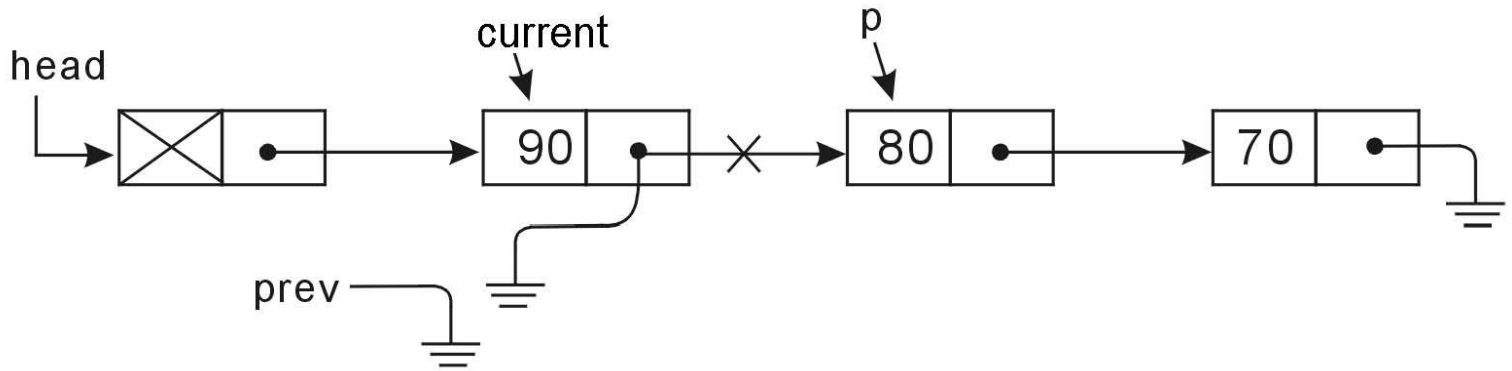
Singly Linked List

- (3)while(p != NULL) {
  prev=current;
  current=p;
  p=p->next;
  current->next=prev;
  }

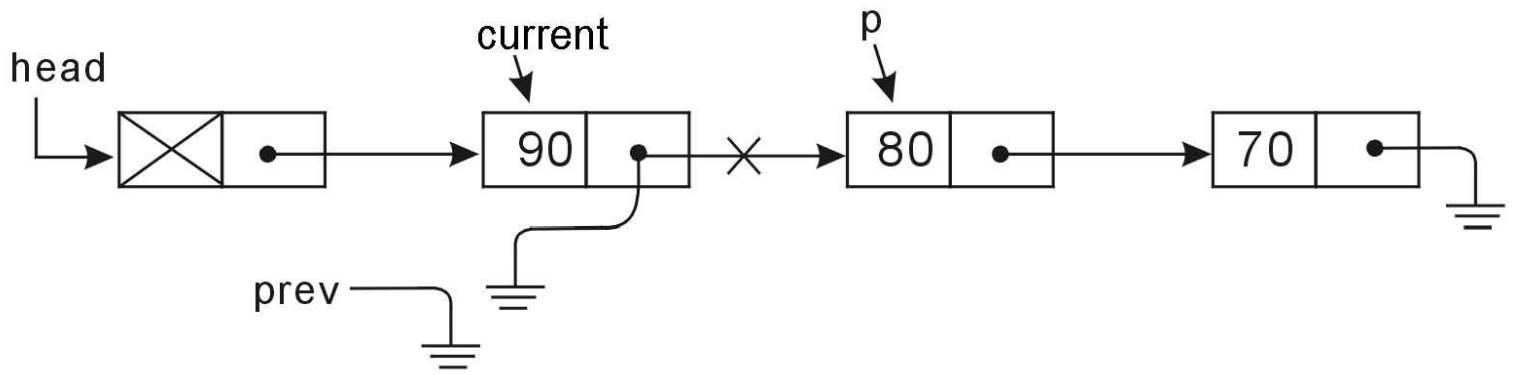
Singly




```
(1) p=head->next;
(2) current=NULL;
(3) while(p != NULL) {
                    prev=current;
                    current=p;
                    p=p->next;
                    current->next=prev;
    }
```

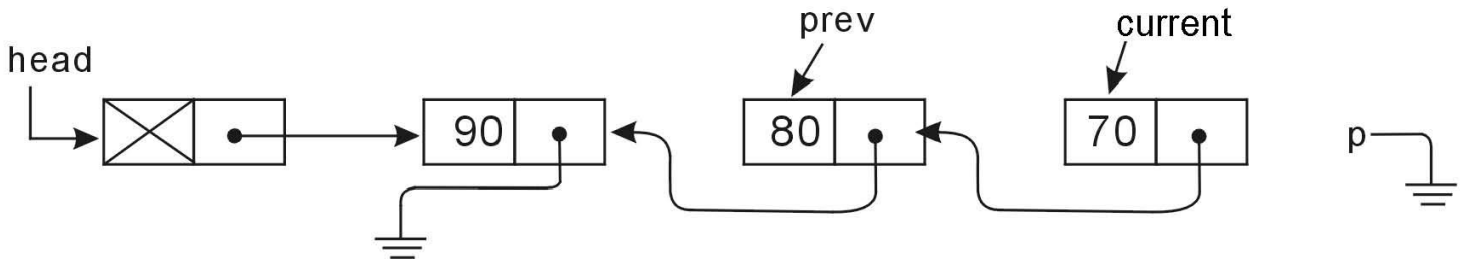


Sir



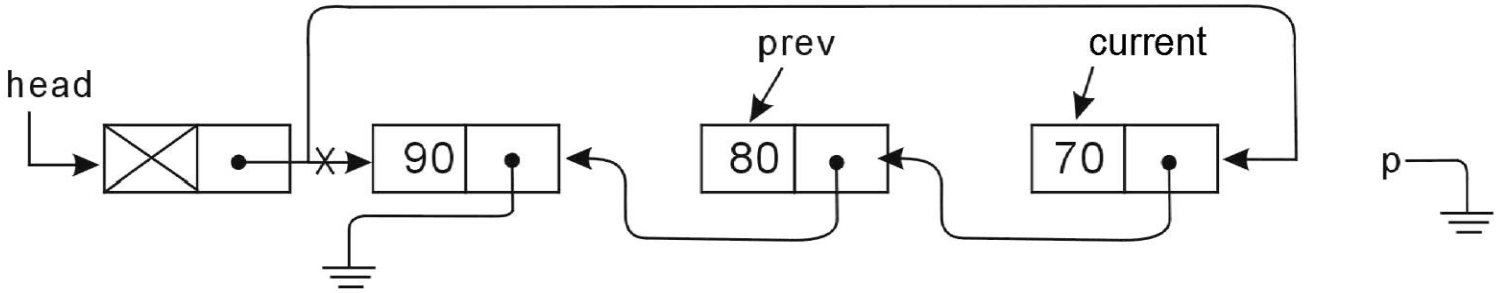

```
(1)p=head->next;
(2)current=NULL;
(3)while(p != NULL) {
                    prev=current;
                    current=p;
                    p=p->next;
                    current->next=prev;
                }
```

prev->current->p



Singly Linked List

- Finally, using head->next=current;



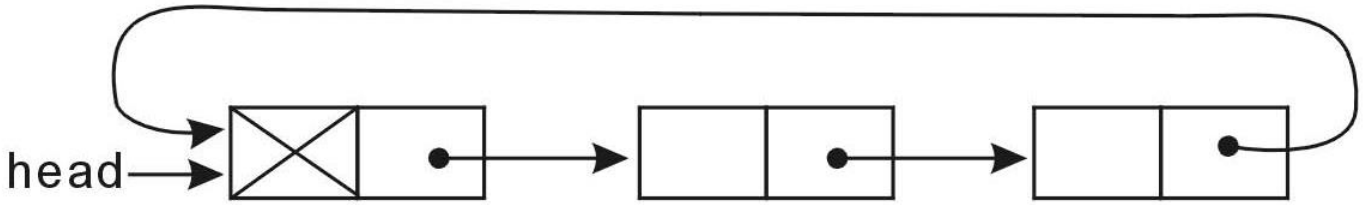
Singly Linked List

- Calculating the length of a list

```
p=head->next;
while( p != NULL) {
    count++;
    p=p->next;
}
```

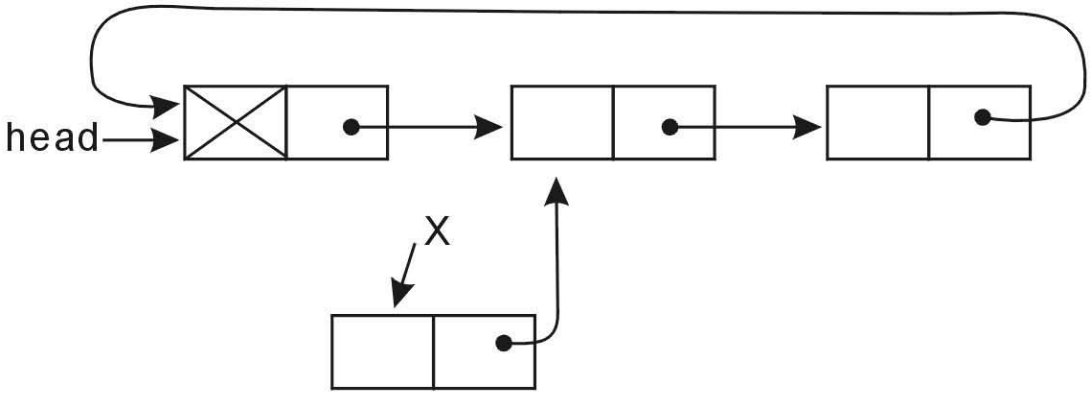
Circular List

- Adding Action
- Add a node to the front of circular list as follows.



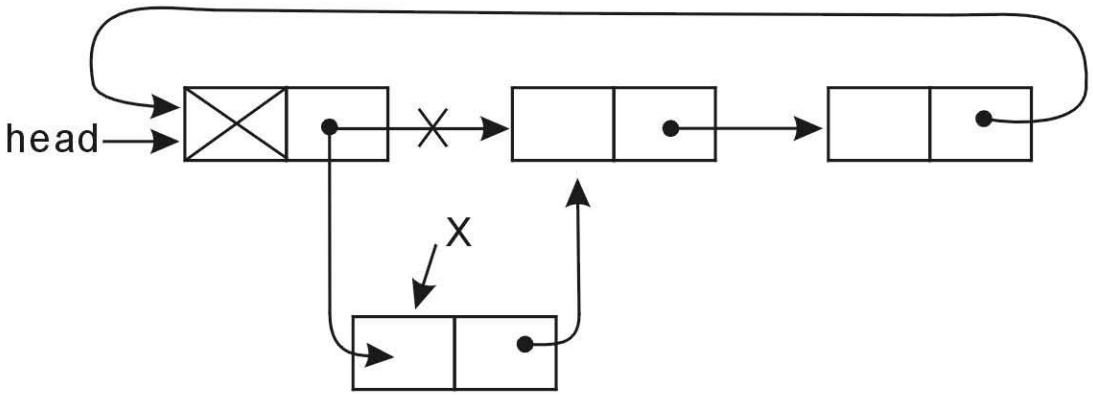
Circular List

(1) x->next=head->next;



Circular List

(2) head->next=x;

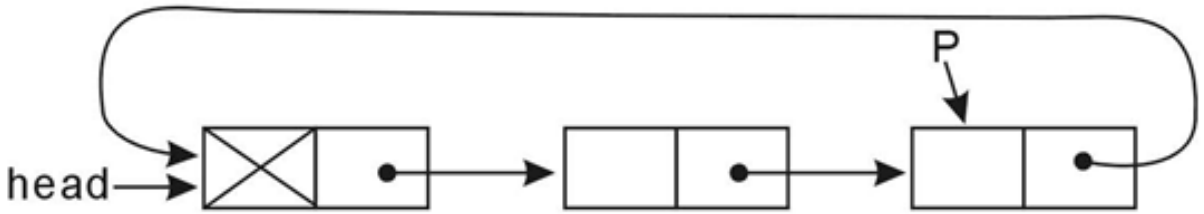


Circular List

- Add a node at the end of circular list

(1) First find where the end is

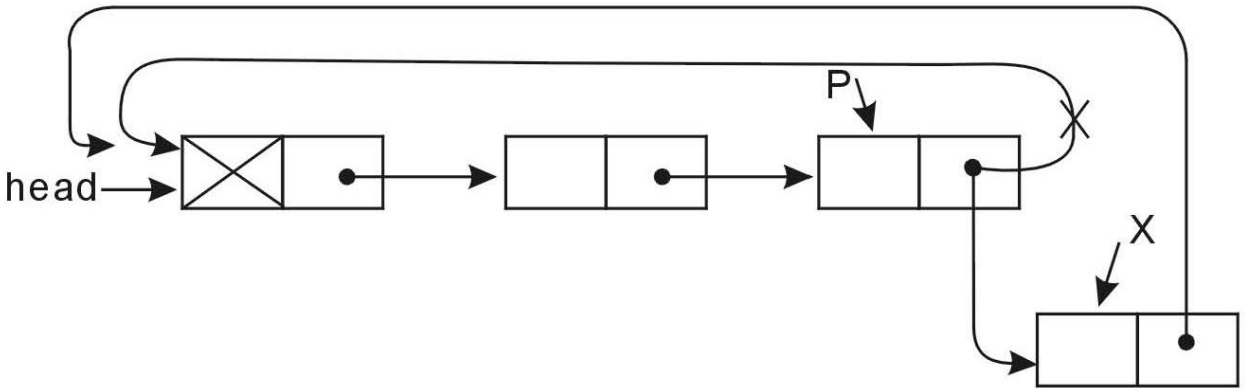
```
p=head->next;
while(p->next != head)
        p=p->next;
```



Circular List

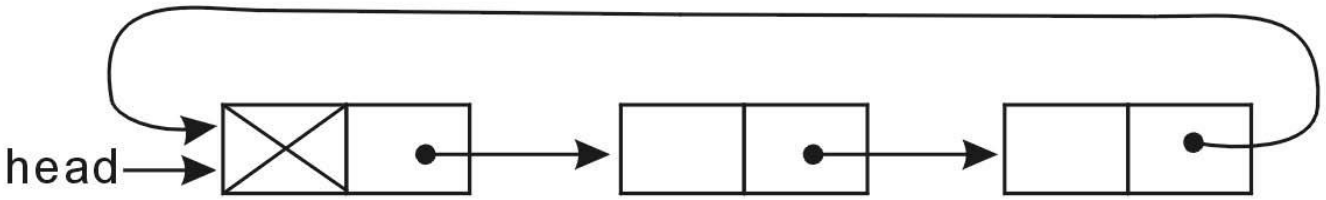
(2) p->next=x;

x->next=head;



Circular List

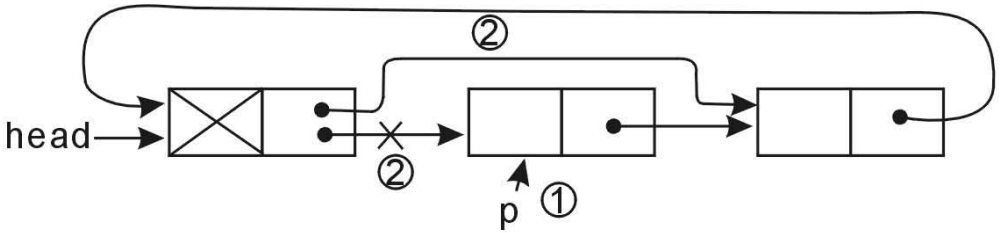
- Delete action
- Delete the front end of Circular List as follows:



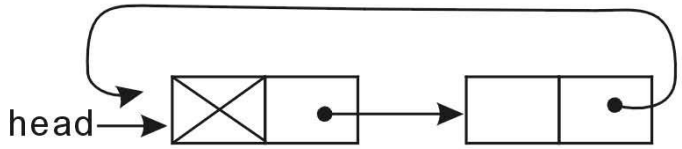
Circular List

(1) p=head->next; /* ① */

head->next=p->next; /* ② */

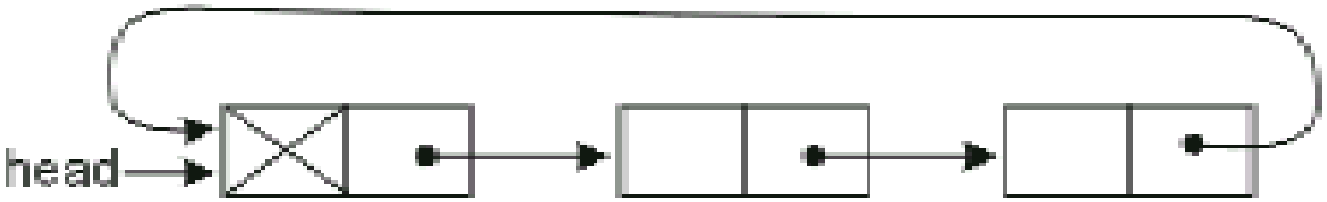


(2) free(p);



Circular List

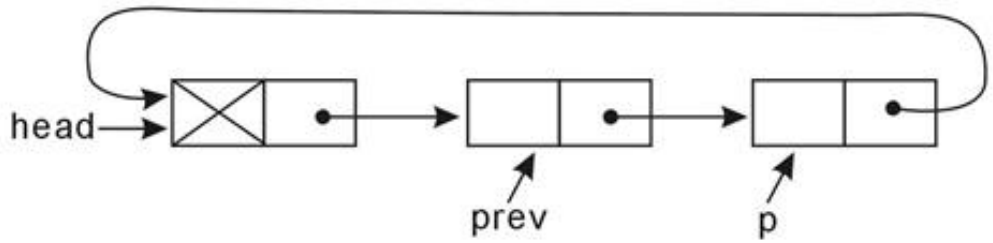
- Delete Circular List at the end as follows:



Circular List

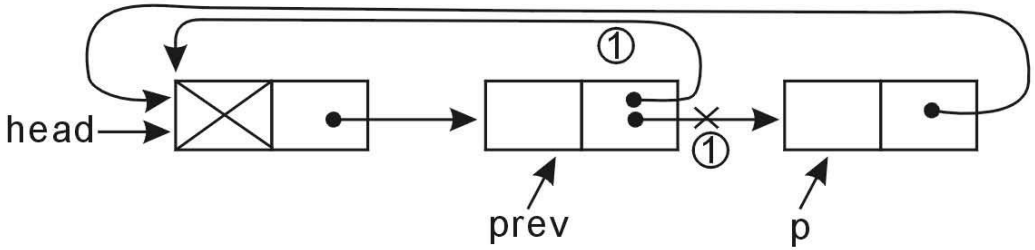
(1) Use the following program to find the end of a list and the previous node at the end

```
p=head->next;
while(p->next != head) {
        prev=p;
        p=p->next;
}
```

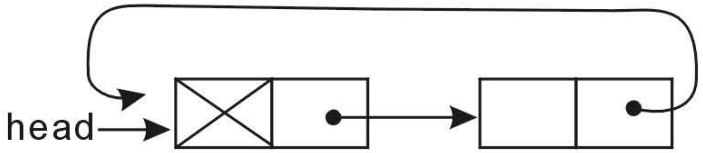


Circular List

(2) prev->next=p->next; /* ① */

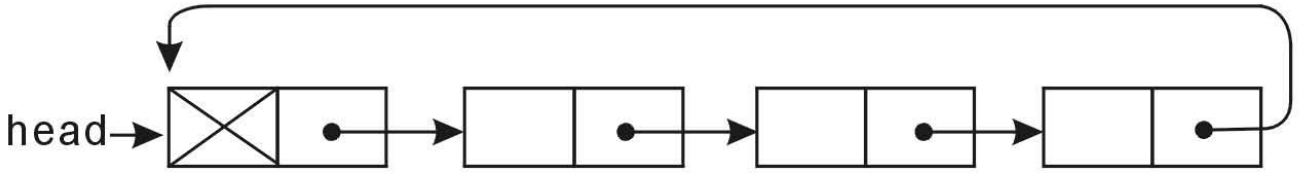


(3) free(p);



Circular List

- Calculating the length of circular list




```
p=head->next;
while (p != head) {
        count++
        p=p->next;
}
```

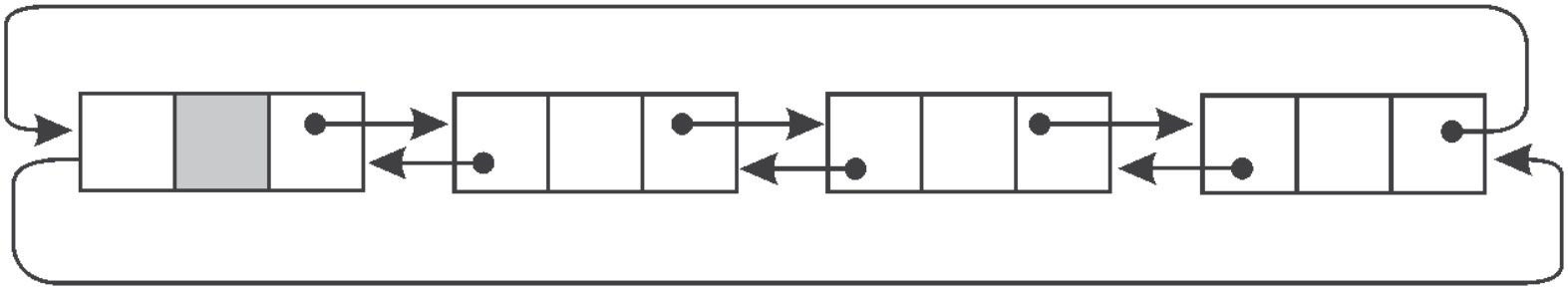
Doubly Linked List

- A doubly linked list is a list with three columns for each node.
- The data structure is as follows:

- Doubly Linked List has the following two features:
  - Assuming ptr is a pointer to any node, then

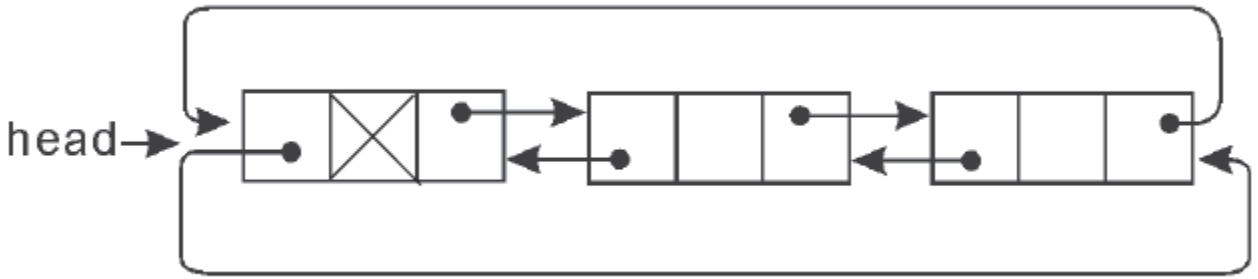
    ptr == ptr->llink->rlink == ptr->rlink->llink;

  - If this Doubly Linked List is an empty list, then there is only one list head.



Doubly Linked List

- Adding Action
- Add to the front of the Doubly Linked List.

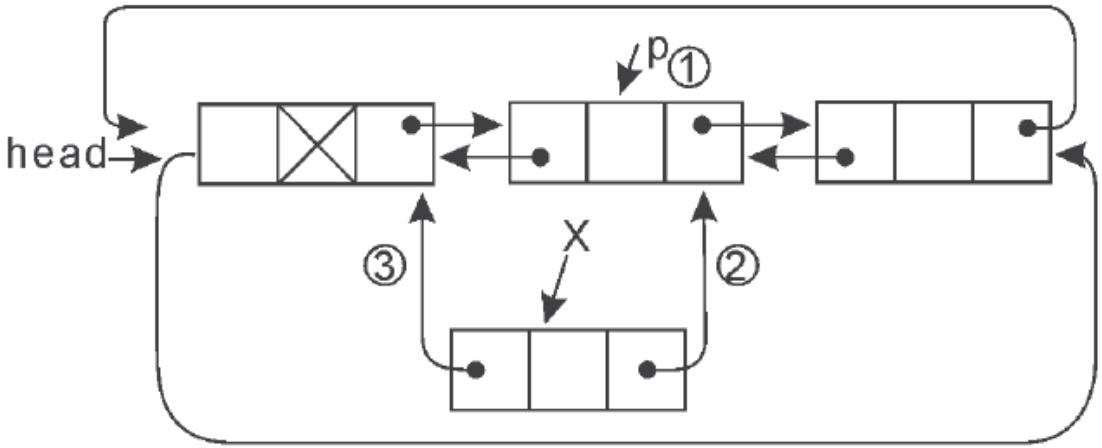


Doubly Linked List

1) p=head->rlink; /* ① */

x->rlink=p; /* ② */

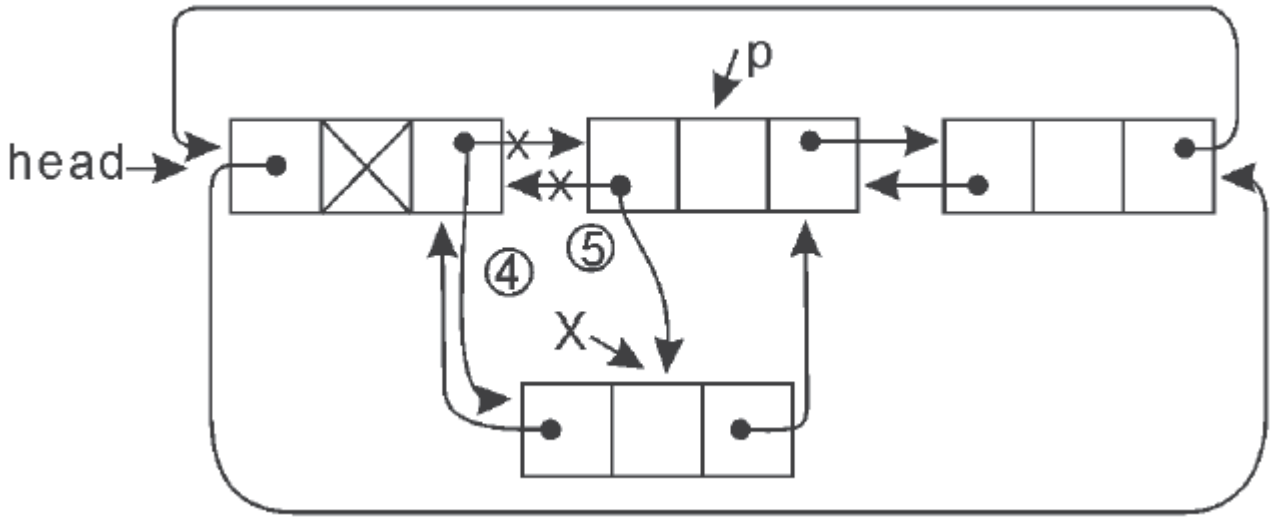
x->llink=head; /* ③ */



Doubly Linked List

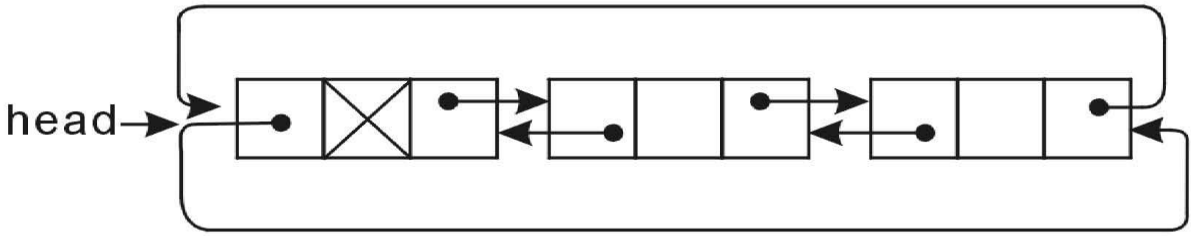
(2) head->rlink=x; /* ④ */

p->llink=x; /* ⑤ */



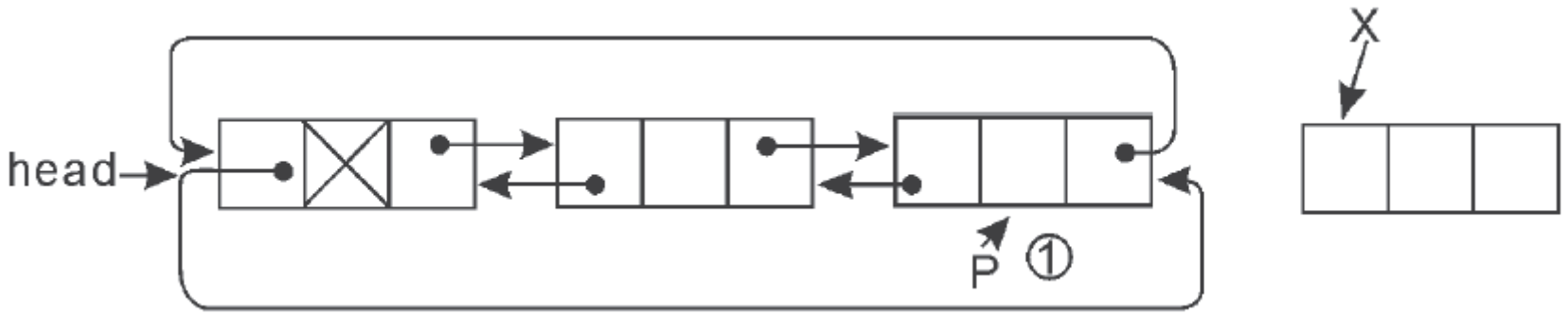
Doubly Linked List

- Add to the end of the Doubly Linked List.



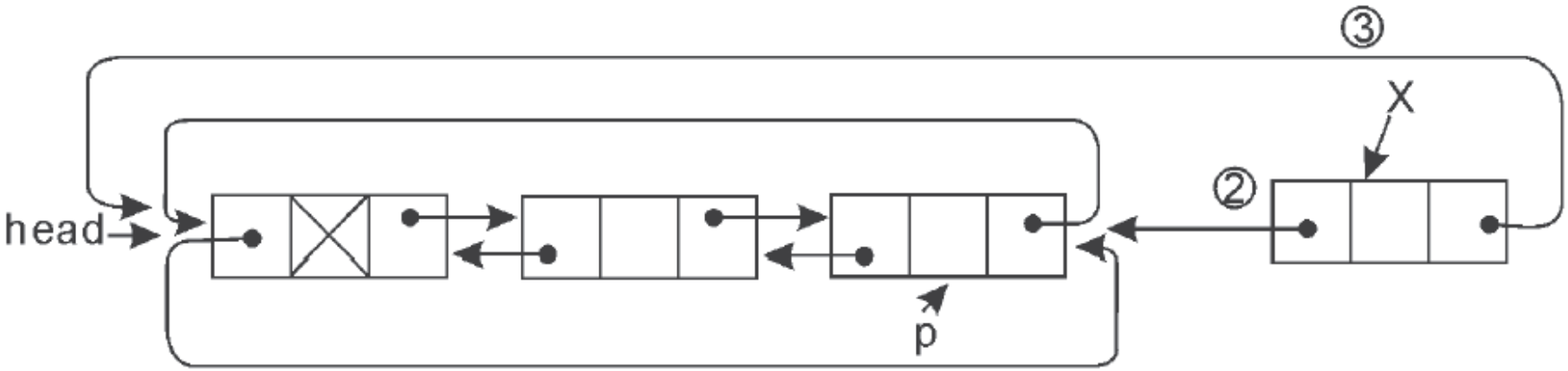
Doubly Linked List

(1) p=head->llink; /* ① */



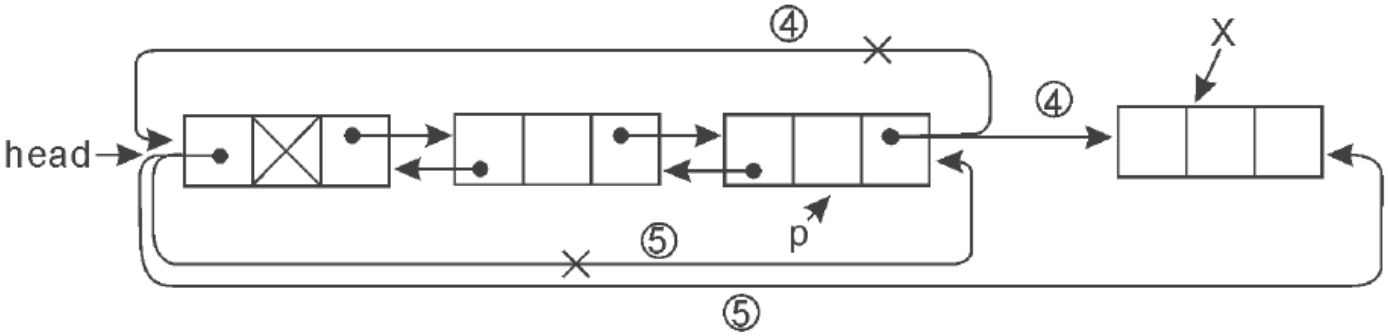
Doubly Linked List

- (2) x->llink=p; /* ② */
- x->rlink=head; /* ③ */



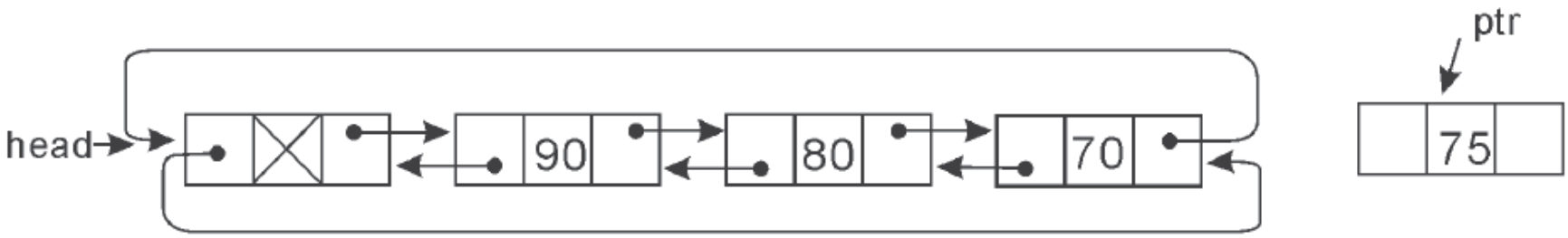
Doubly Linked List

- (3) p->rlink=x; /* ④ */
- head->llink=x; /* ⑤ */



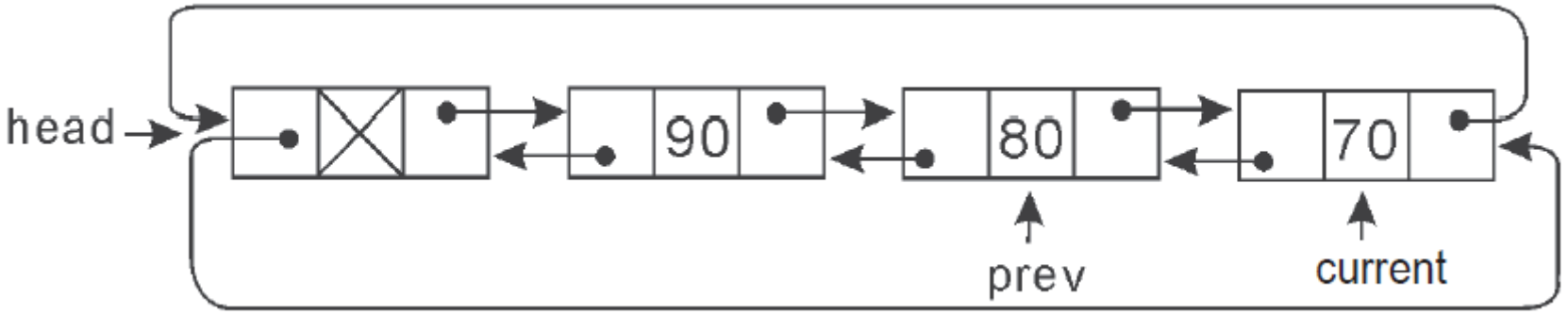
Doubly Linked List

- Join after a particular node Join after a particular node.



Doubly Linked List

```
prev=head;
current=head->rlink;
while( current != head && current->score > ptr->score) {
      prev=current;
      current=current->rlink;
}
```



Doubly Linked List

- Next, use the following four steps to complete the joining process.

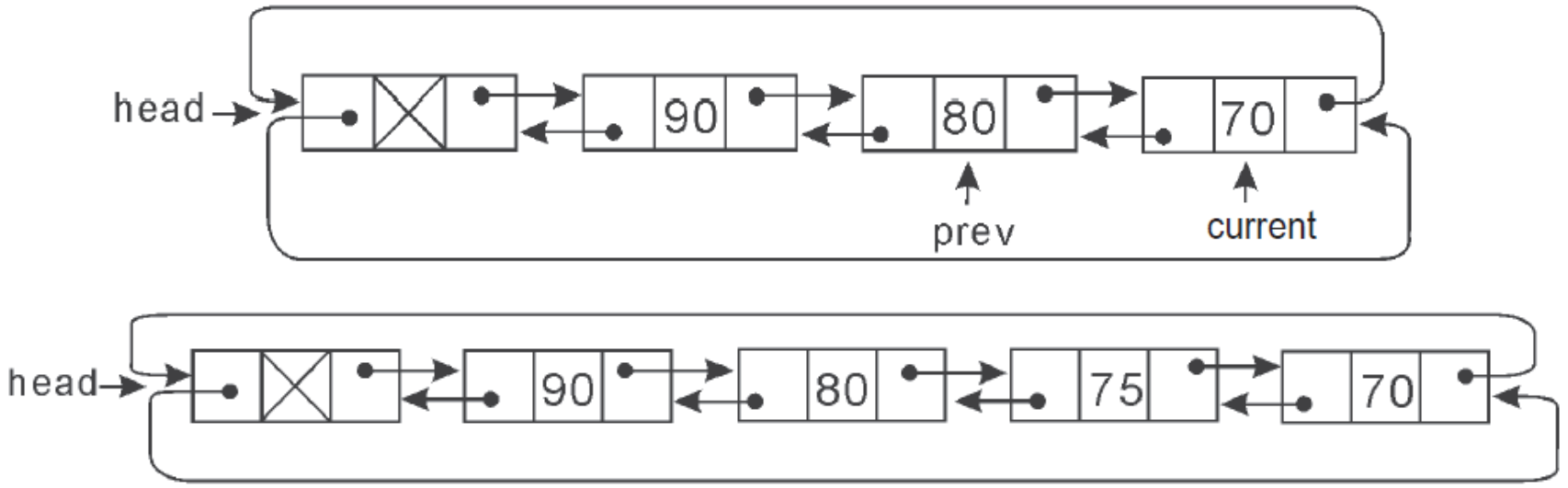
ptr->rlink=current;

ptr->llink=prev;

- Data is placed between prev and current

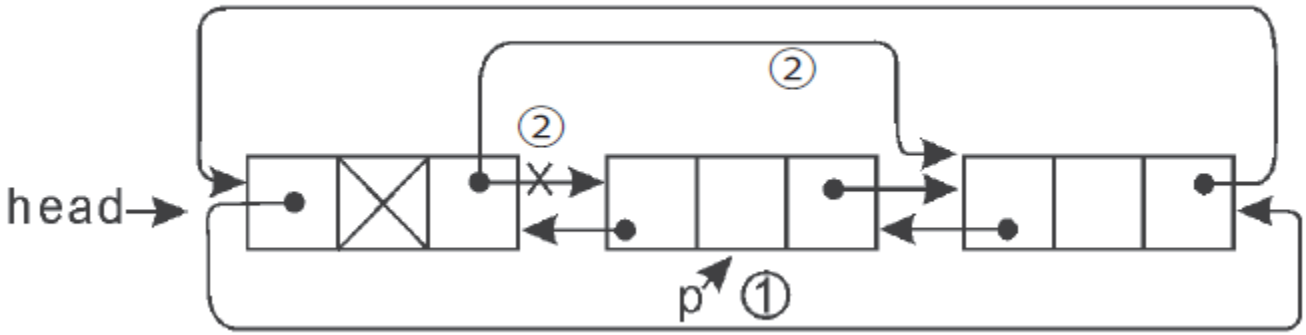
prev->rlink=ptr;

current->llink=ptr;



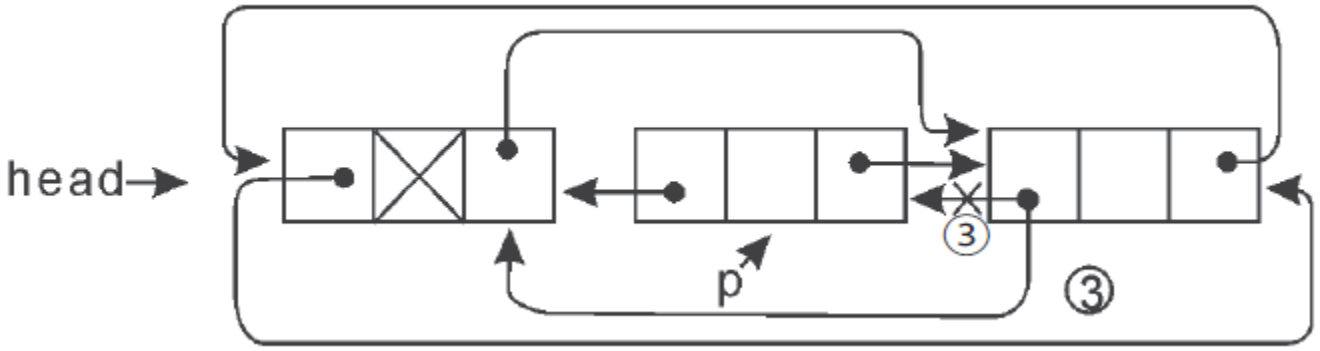
Doubly Linked List

- Delete Action
- Delete the front of Doubly Linked List.
  - (1) p=head->rlink; /* ① */
  - (2) head->rlink=p->rlink; /* ② */

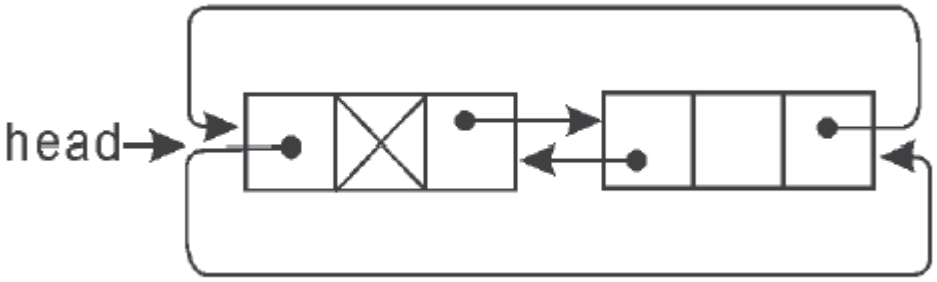


Doubly Linked List

- (3) p->rlink->llink=p->llink; /* ③ */

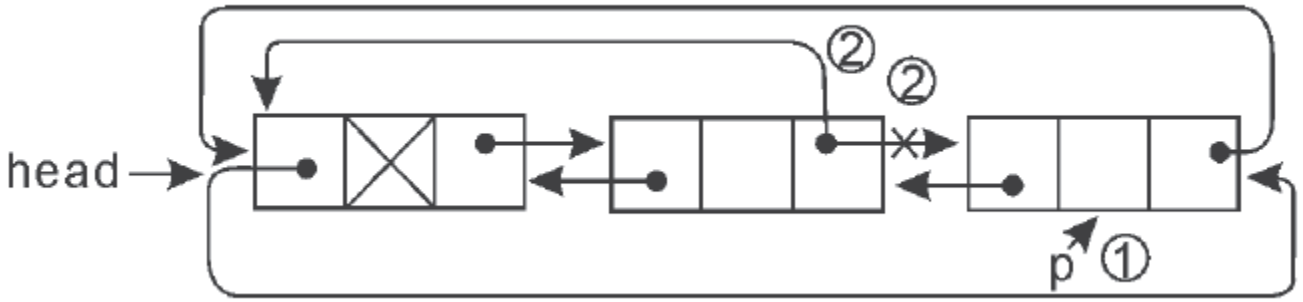


- (4) free(p);



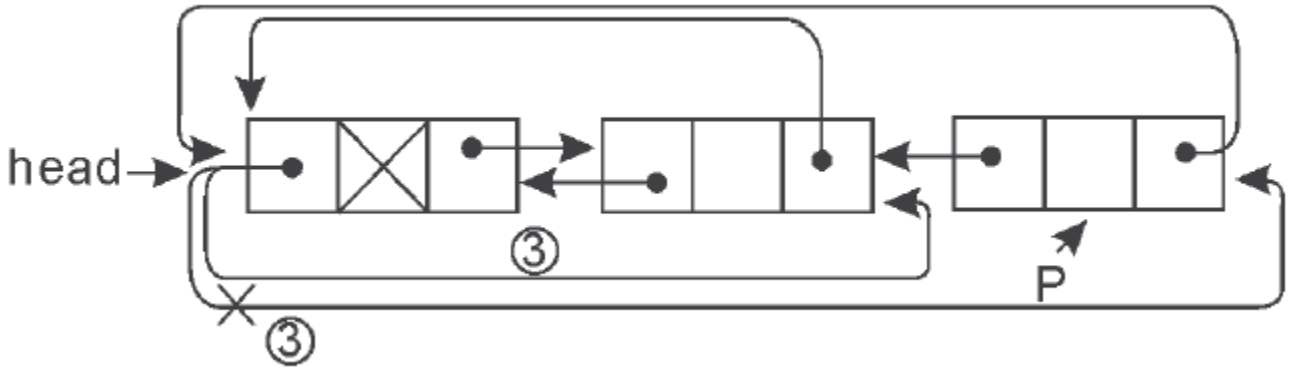
Doubly Linked List

- Delete the end of Doubly Linked List.
  - (1) p=head->llink; /* ① */
  - (2) p->llink->rlink=p->rlink; /* ② */



Doubly Linked List

- (3) head->llink=p->llink; /* ③ */



- (4) free(p)