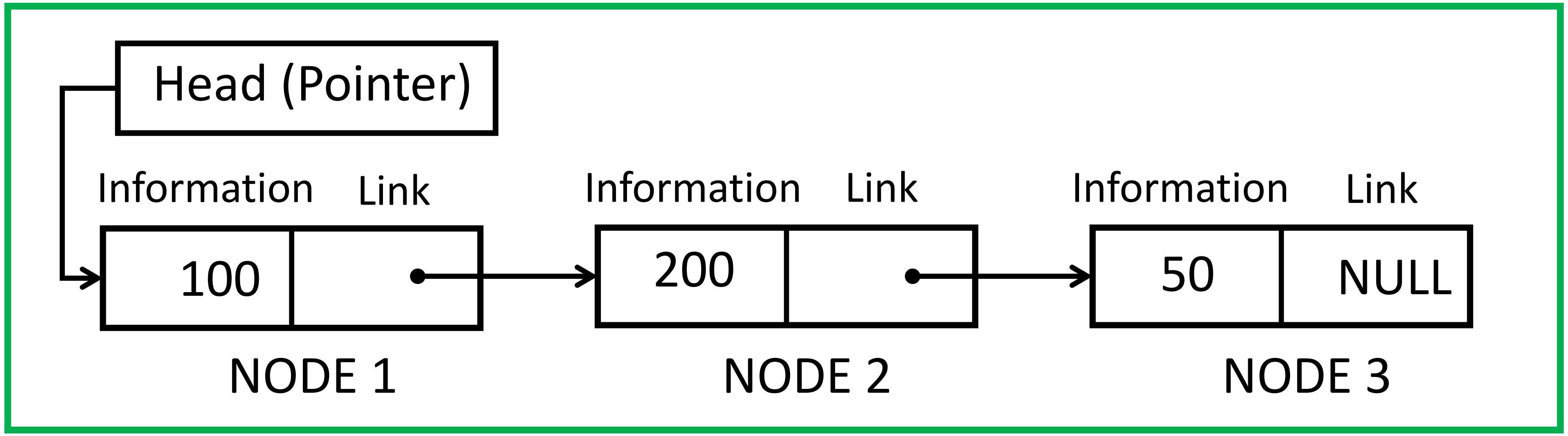
LINKED LIST

PROF. NAVRATI SAXENA

Lists

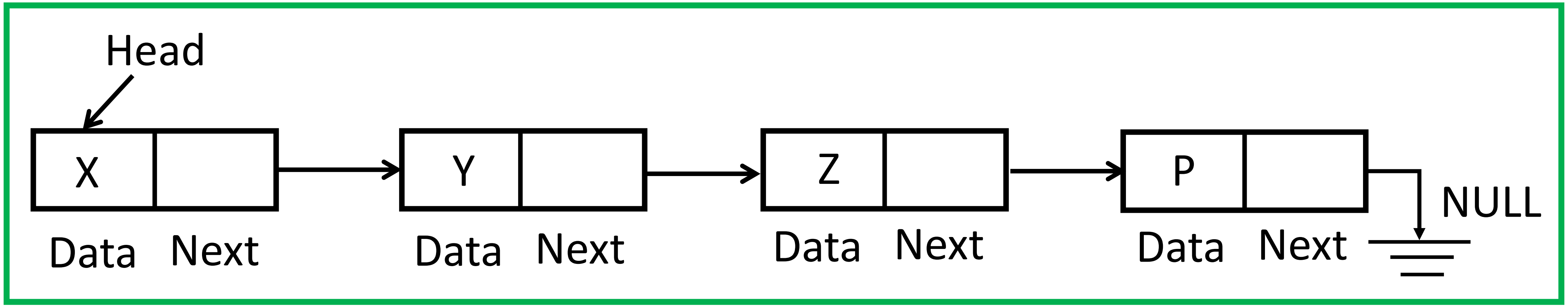
- Collection of variable number of data items
- Each item called nodes
- Most commonly used non- primitive data structure
- Each nodes is divided into two parts:

  (1) Information, (2) Link – containing link (memory address) of next node



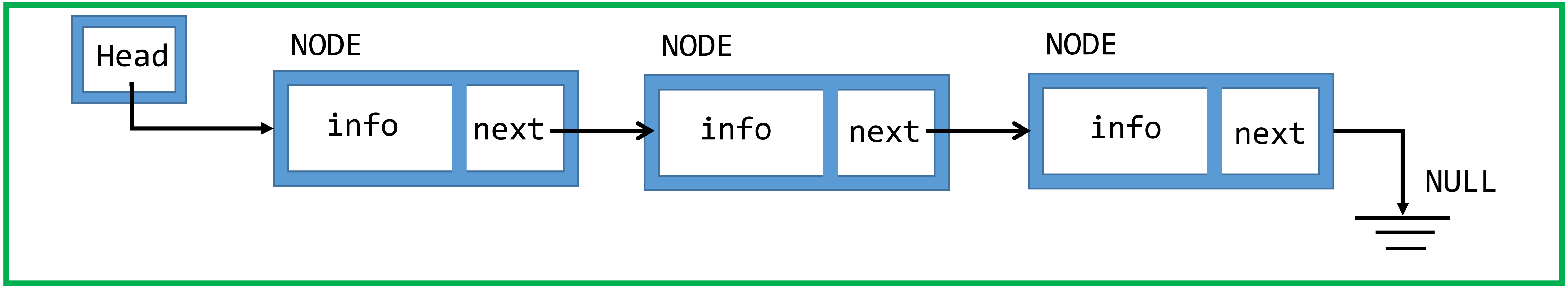
Operation on Linked List

- Creating a linked list
- Traversing a linked list
- Inserting an item into a linked list
- Deleting an item from the linked list
- Searching an item in the linked list
- Merging two or more linked lists



Creating a Linked List

```
struct Node
{
  int info;
  struct Node *next;
};
```



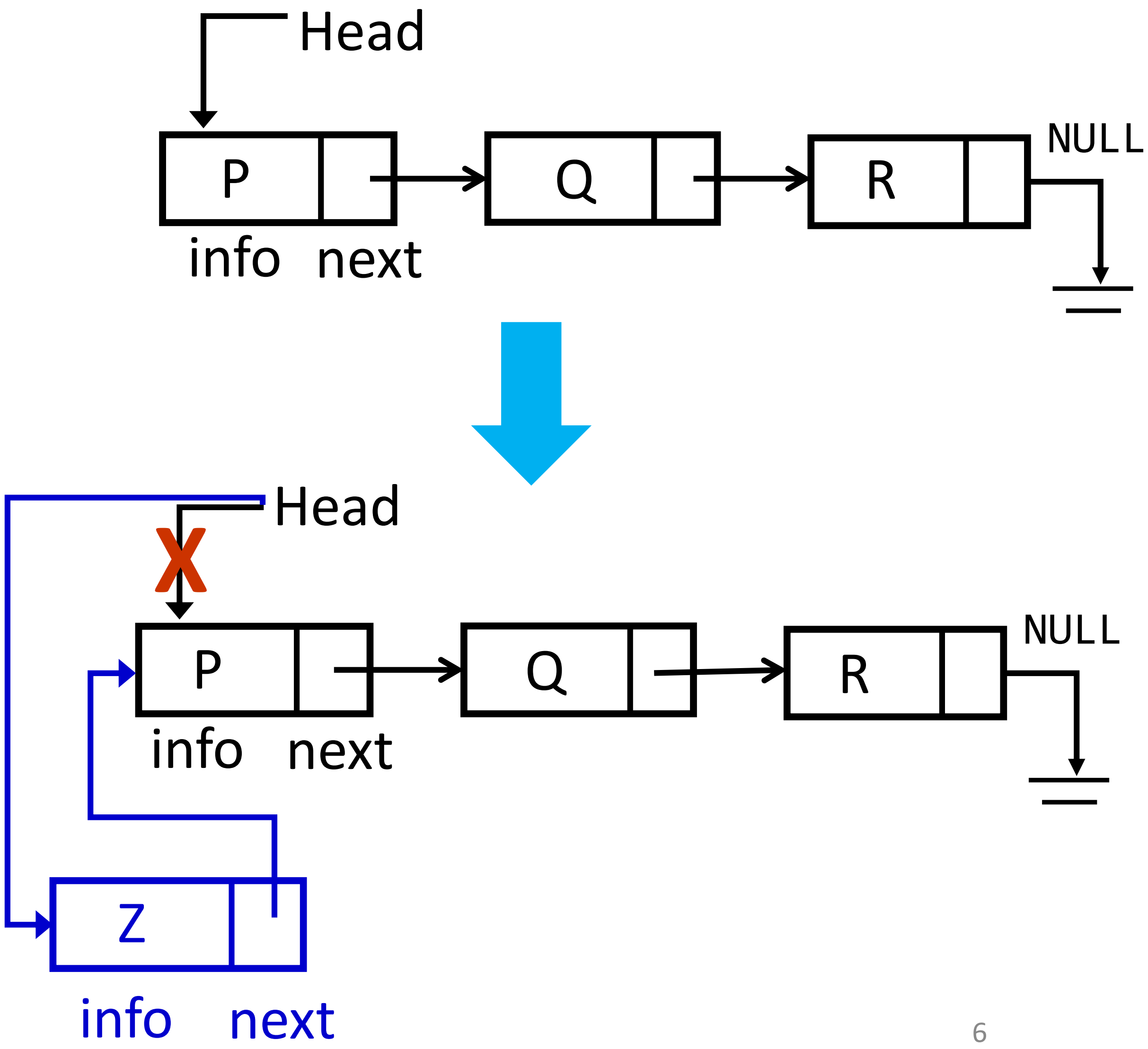
Inserting a Node into the Linked List

- Inserting a node at the beginning of the linked list
- Inserting a node at the given position.
- Inserting a node at the end of the linked list.

Inserting in the Beginning (1/2)

A four step process

1. Create a node
2. Fill data into the new node
3. Make next of new node point to where "Head" node points to
4. Move head to point to the new node



Inserting in the Beginning (2/2)

```
void insertBeginning (struct node** head, int new_info)
{
   /* 1. create a new empty node */
   struct node* new_node = (struct Node*) malloc(sizeof(struct node));
                                   /*allocates the requested memory and returns a pointer to it */
   /* 2. put in the data  */
   new_node->info  = new_info;

   /* 3. Make next of new node point to where head was pointing to that is to the
first node of the previous list */
   new_node->next = (*head);

   /* 4. move the head to point to the new node */
   (*head)    = new_node;
}
```

Inserting After a Given Node (1/2)

A five step process

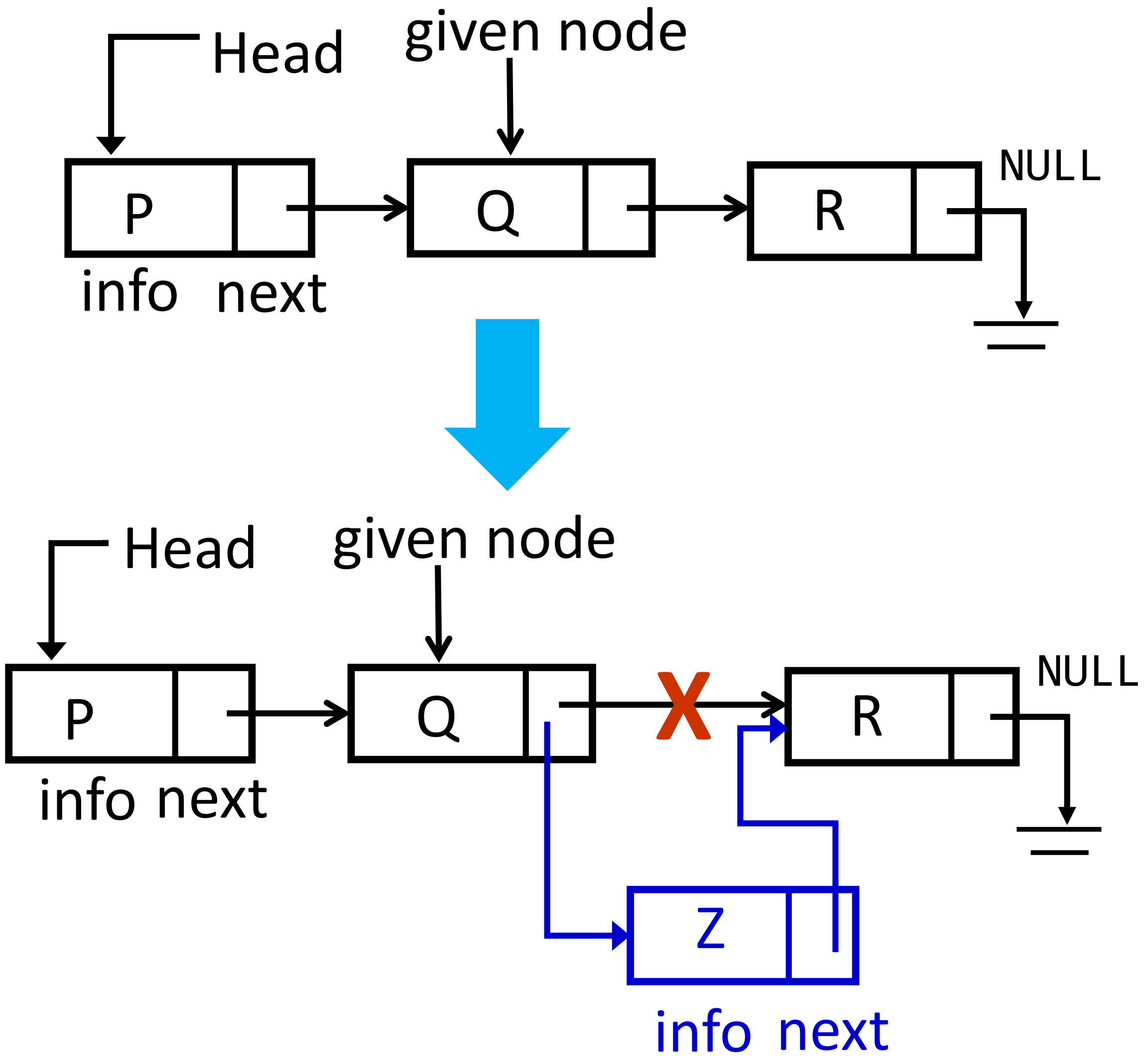
1. check if the given node is NULL

2. Create / allocate a new node

3. Put data in the new node

4. Make next of new node as next of given node

5. Make the next of given node point to the new_node

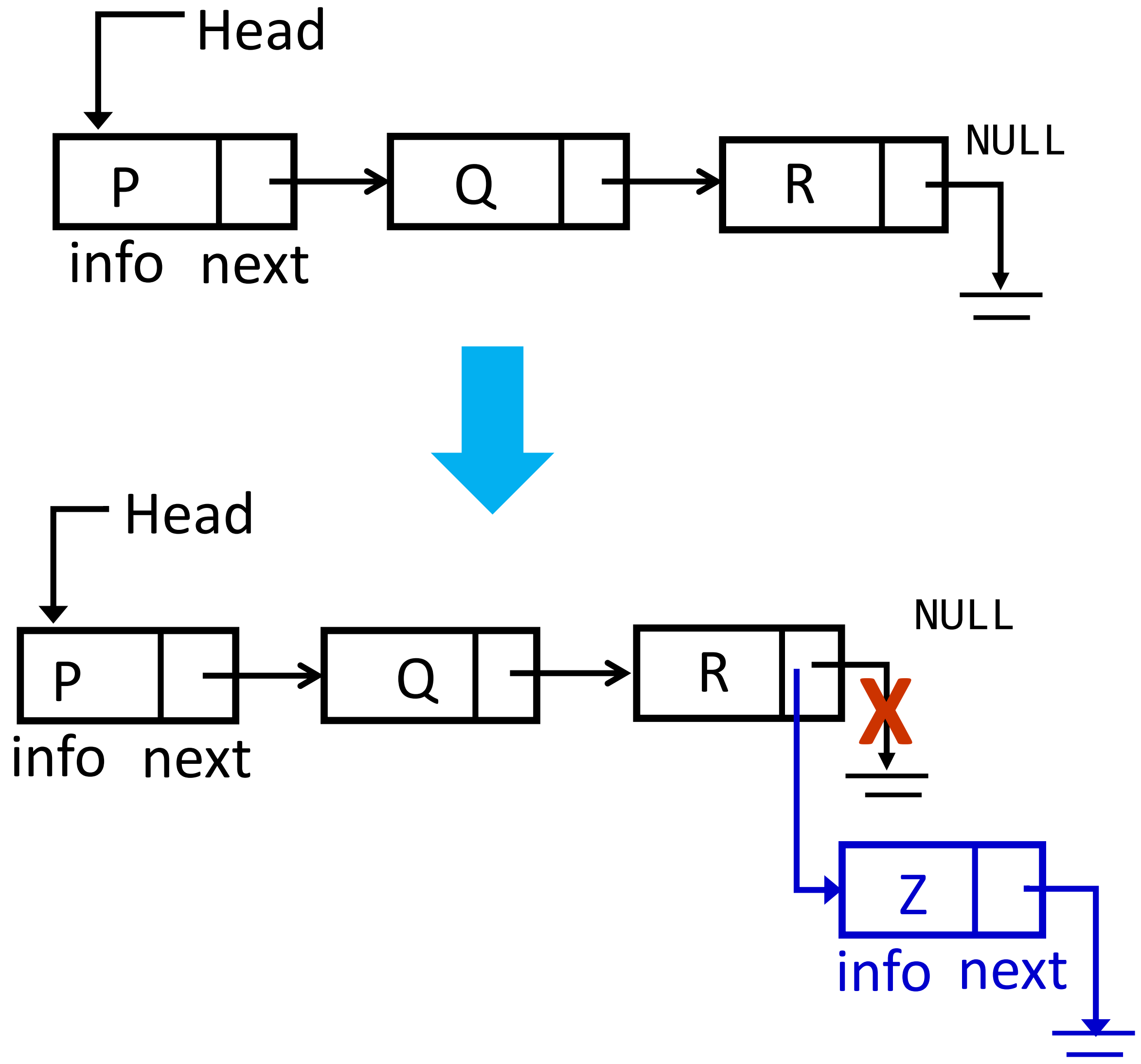


Inserting After a Given Node (2/2)

```
void insertAfter(struct node* given_node, int new_info)
{
   if (given_node == NULL)                                    /*1. check if the given node is NULL */
         {
         printf(“ERROR! The given node cannot be NULL");
         return;
         }
   struct node* new_node =(struct node*) malloc(sizeof(struct node));  /* 2. allocate new node */
   new_node->info  = new_info;                  /* 3. put in the data  */
   new_node->next = given_node->next;           /* 4. Make next of new node as next of prev_node */
   given_node->next = new_node;                 /* 5. move the next of prev_node as new_node */
}
```

Inserting at the End (1/2)

1. Create / a new allocate node
2. Put in the data
3. This new node is going to be the last node, so make next of it as NULL
4. If the Linked List is empty, then make the new node as head
5. Else traverse till the last node
6. Change the next of last node



Inserting at the End (2/2)

```
void insertEnd(struct Node** head_ref, int new_data)
{
   struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));     /* 1. allocate node */
   struct Node *last = *head_ref;          /* Copy the head node */
   new_node->data  = new_data; /* 2. put in the data  */
   new_node->next = NULL;                  /* 3. New node = last node, make next  = NULL*/
   if (*head_ref == NULL)                  /* 4. If the Linked List is empty, then make the new node as head */
         {
         *head_ref = new_node;
         return;
         }
   while (last->next != NULL)              /* 5. Else traverse till the last node */
      last = last->next;
   last->next = new_node;                  /* 6. Change the next of last node */
}
```

Deleting a Node from the Linked List

- Deleting a node at the beginning of the linked list
- Deleting a node at the given position.
- Deleting a node at the end of the linked list.

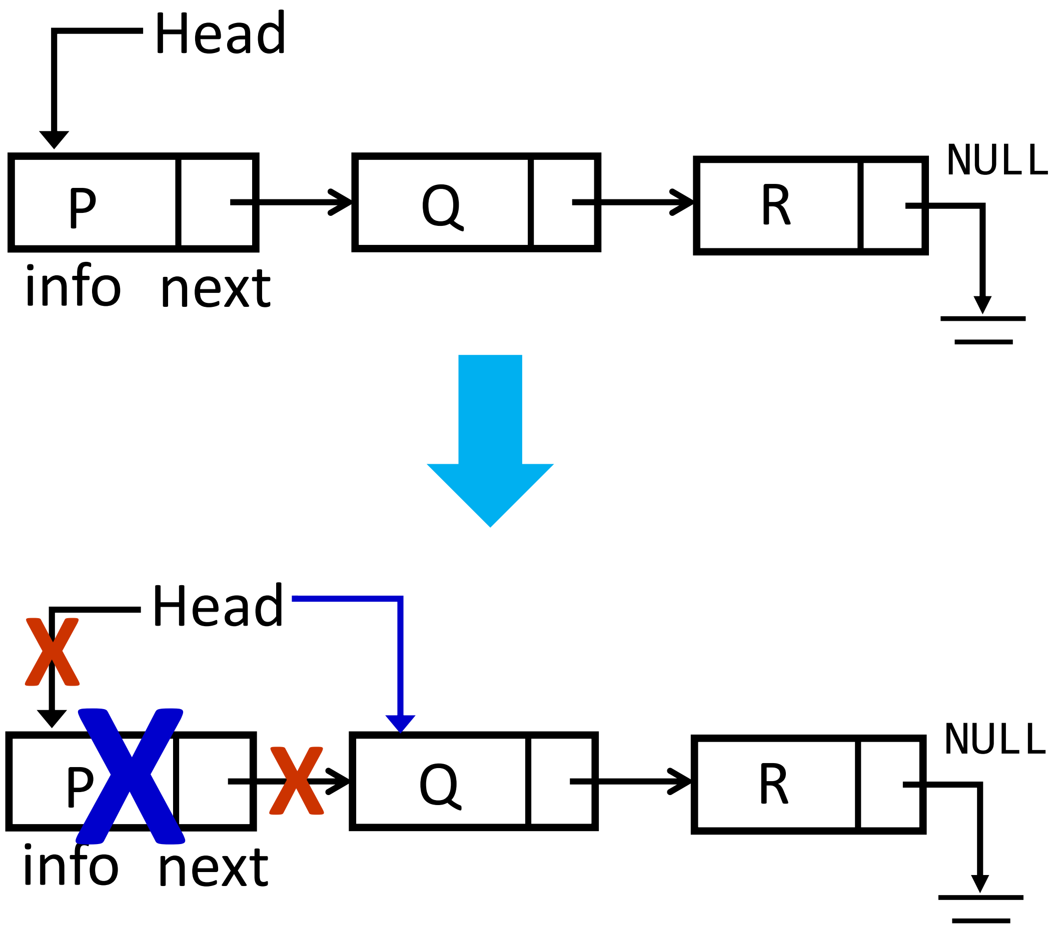
Deleting a Node from the Beginning

DELETE THE FIRST NODE

- A two-step process

1. Make second node as head

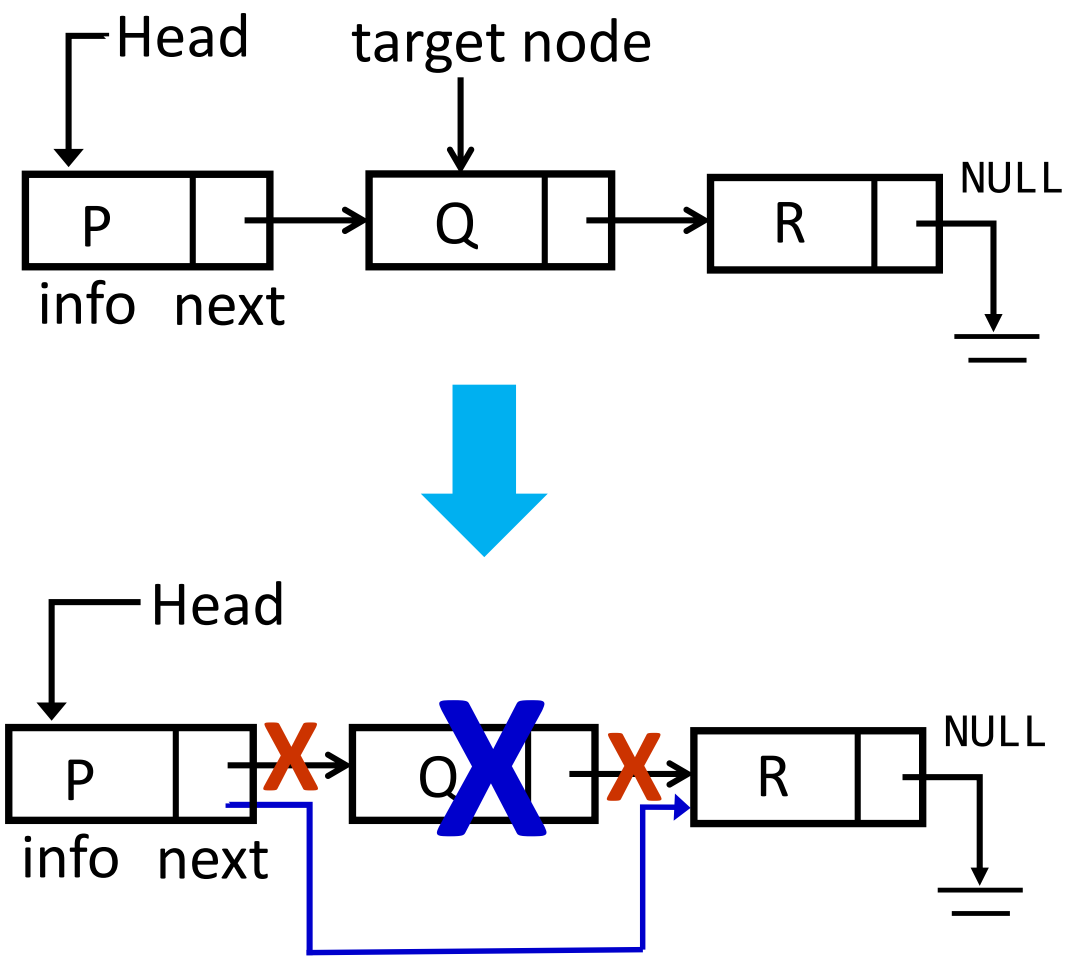
2. Delete memory allocated for first node.



Deleting a Node From a Given Position

DELETE A NODE FROM A GIVEN POSITION

1. Find previous node of the target node (node to be deleted)
2. Change the next of previous node to next of the given node
3. Free memory for the node to be deleted



Deleting Node from the End

DELETE THE LAST NODE

1.If the first node is *null* or there is only one node,

return *null*

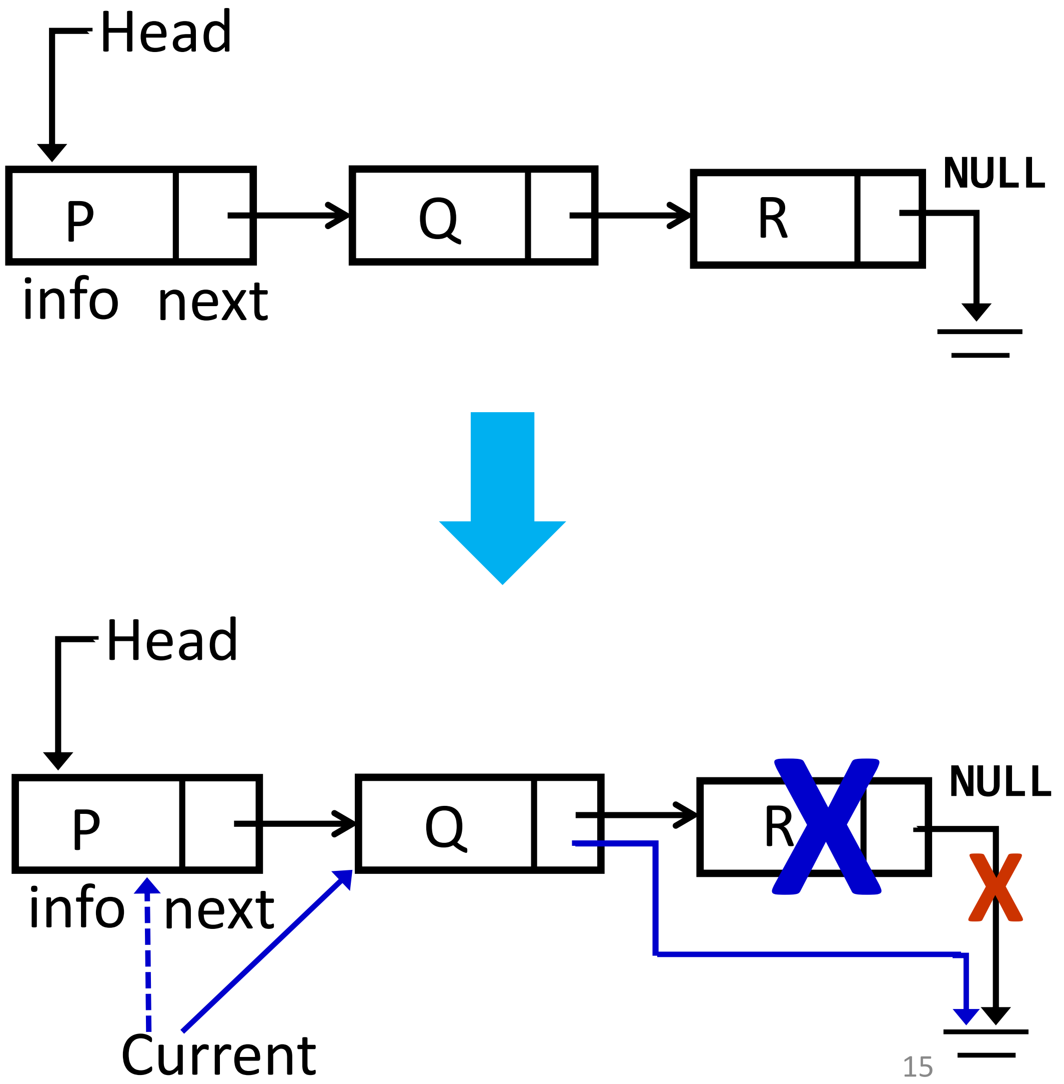
2. Use a temporary pointer *“Current”*

3. Initialize *Current* to the first node (Head)

4.Traverse the linked list till the second-last node,

i.e. until (Current -> next) -> next is NULL

3. Delete the next node of the second-last node



Deleting Node from the End

DELETE THE LAST NODE

1.If the first node is *null* or there is only one node, then return *null*

```
if headNode == null then return null
if headNode.nextNode == null then free head and return null
```

2. Traverse the linked list till the second last node

```
while secondLast.nextNode.nextNode != null
    secondLast = secondLast.nextNode
```

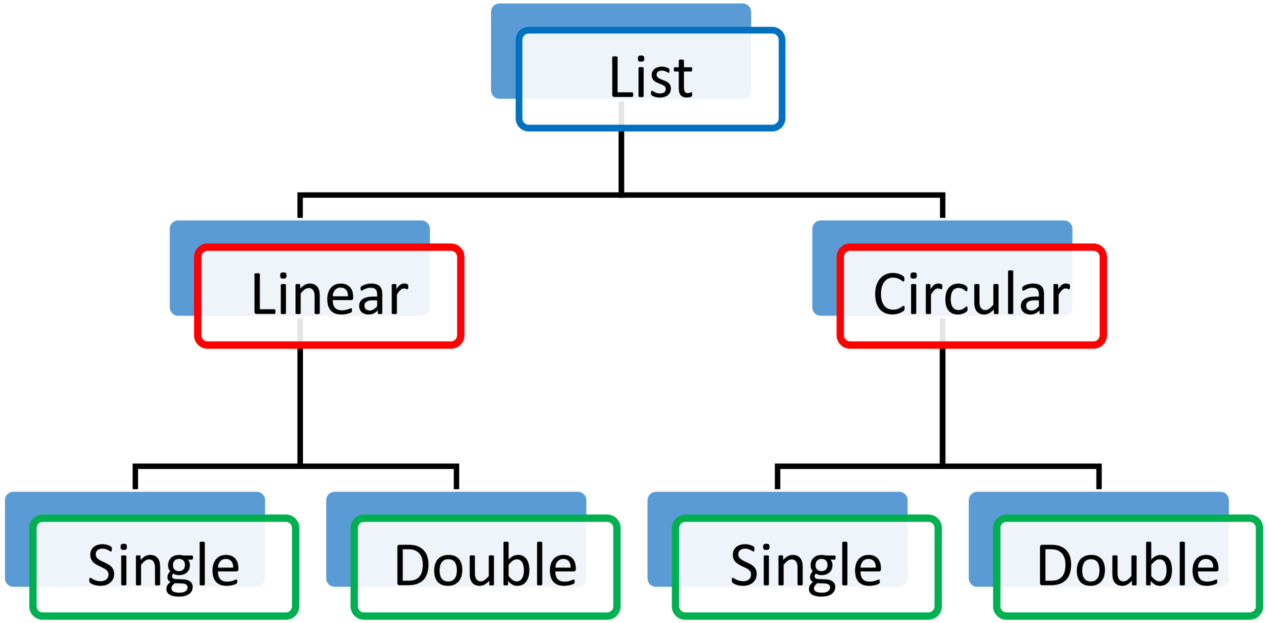
3. Delete the last node, i.e. the next node of the second last node

delete (*secondLast*.nextNode)

Set the value of next of second last node to null

Types of Lists

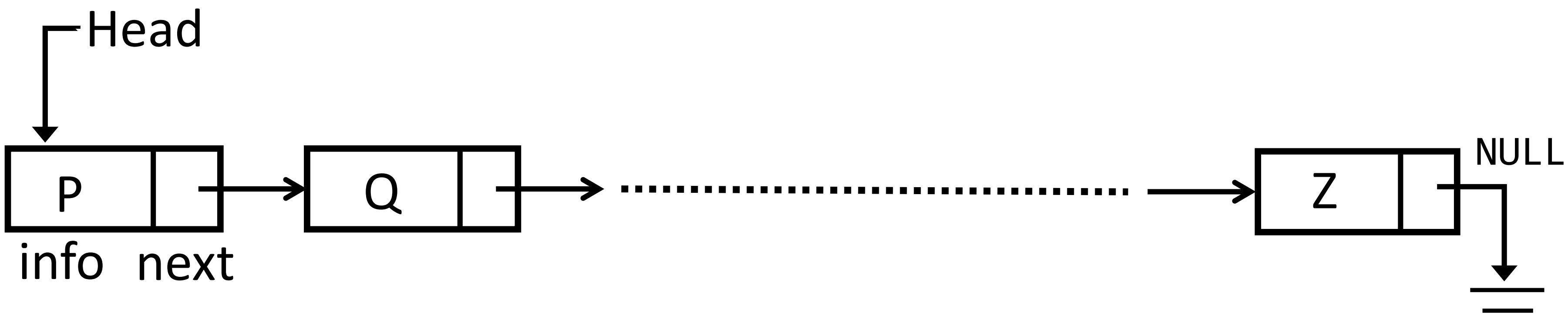
- Single linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list



Singly Linked List (SLL)

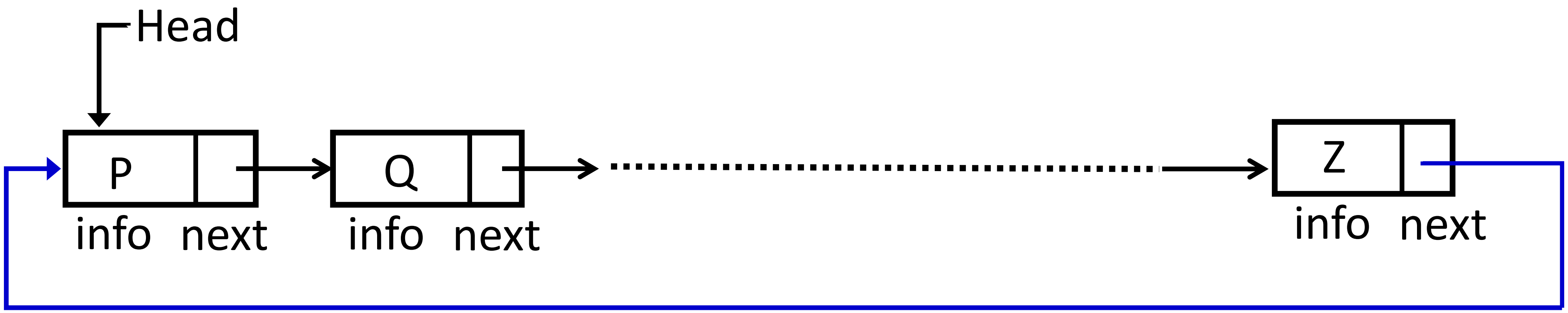
A singly linked list contains two fields in each node

- The **information** field contains the data of that node
- The **link or next** field contains the memory address of the next node.
    - Only one link field in each node - singly linked list
- Item navigation is forward only



Single Circular Linked List

- Last item contains link of the first element as next and the first element has a link to the last element as previous
- In a circular linked list every node is accessible from any node

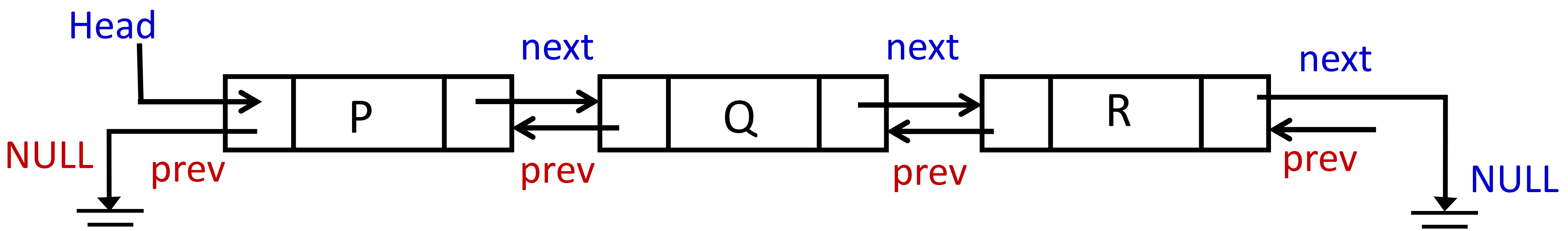


Circular, Singly Linked List

Doubly Linked List (DLL)

- Items can be navigated forward and backward
- Each node is points both to the next node and also to the previous node
- In doubly linked list each node contains three parts:
  1. **Next**: A pointer field that contains the address of the next node
  2. **Previous**: A pointer that contains the address of the previous node
  3. **Information**: Actual data
- If Previous = NULL => This is the first node of the list
- If Next = NULL => This is the l a s t node of the list

```
struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
```



Singly vs. Doubly Linked Lists

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction

2) The delete operation in DLL is more efficient (if pointer to the node to be deleted is given)

3) We can quickly insert a new node before a given node

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer

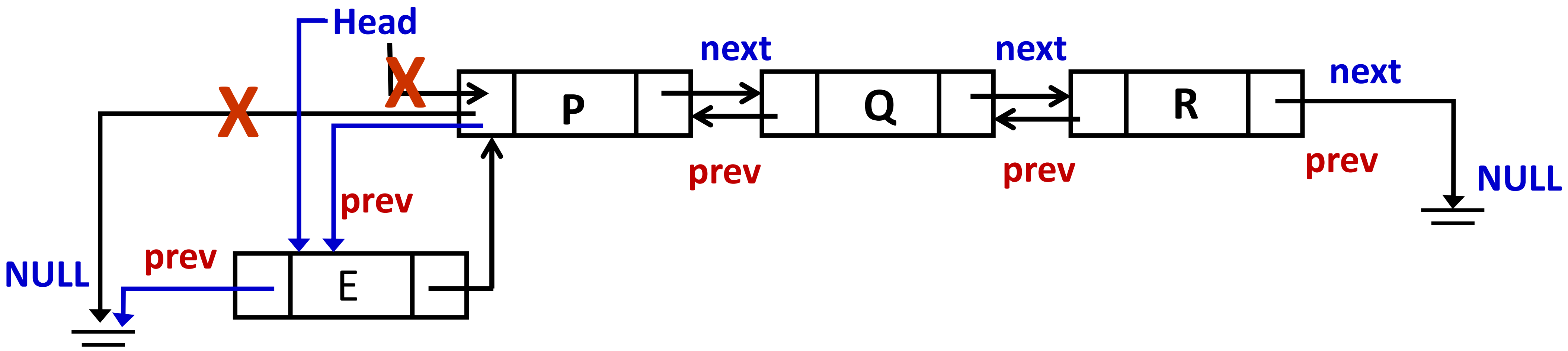
2) All operations require an extra pointer previous to be maintained

Insertion in DLL

1. At the front of the DLL
2. At the end of the DLL
3. After a given node
4. Before a given node

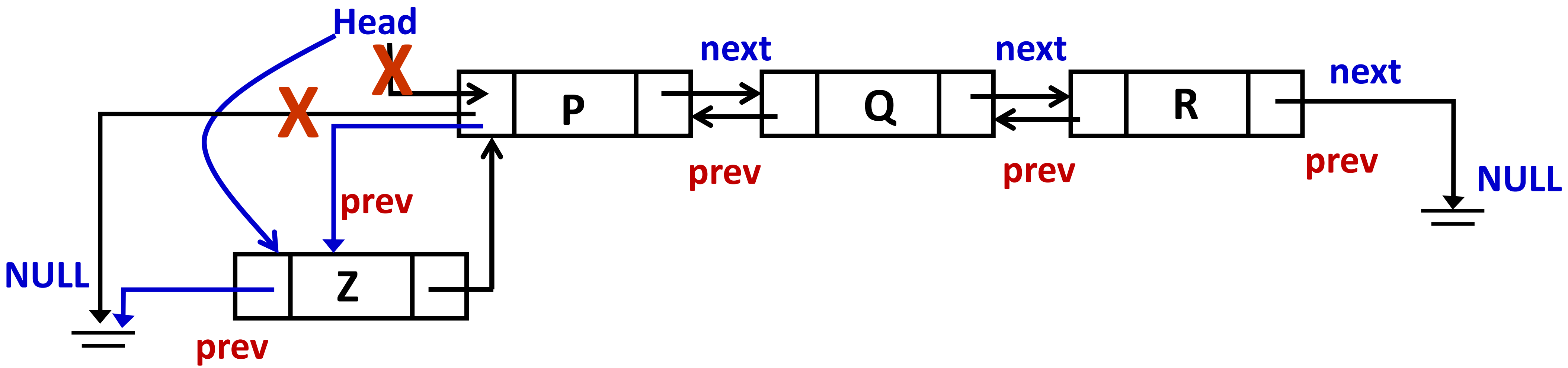
Insertion in the Beginning (1/2)

1. Allocate a new node
2. Put in the data in the new node
3. Make next of the new node as head and previous as NULL
4. Change prev of head node to new node
5. Move the head to point to the new node



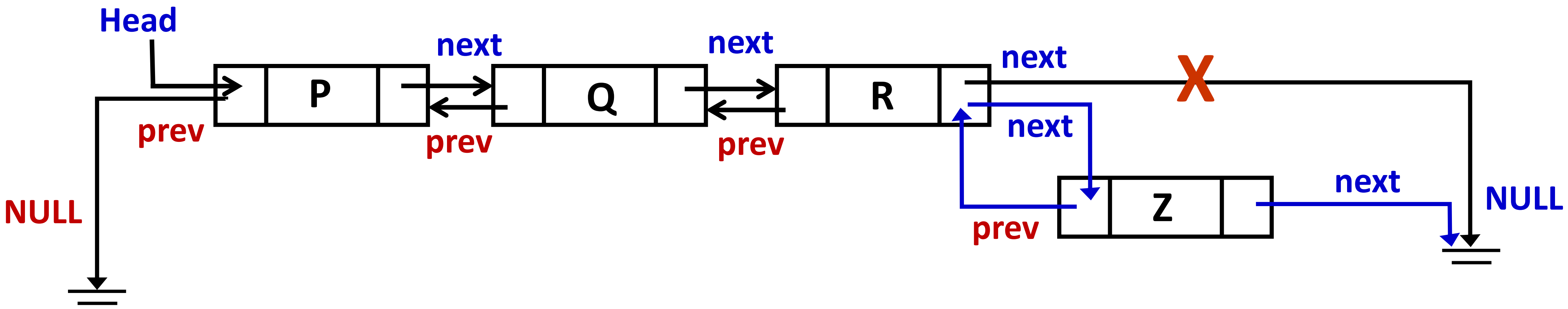
Insertion in the Beginning (2/2)

```
void insertBeginningDouble(struct node** head, int new_info)
{
        struct node* new_node = (struct node*)malloc(sizeof(struct node));
        new_node->data = new_info;
        new_node->next = (*head);
        new_node->prev = NULL;
        if ((*head) != NULL)
             (*head)->prev = new_node;
        (*head) = new_node;
}
```



Insertion at the End (1/2)

```
1. Allocate node
2. Put in the data
3. This new node is going to be the last node, so make its next = NULL
4. If the Linked List is empty, then make the new node as head
5. Else traverse till the last node
6. Change the next of last node
7. Make last node as previous of new node
```

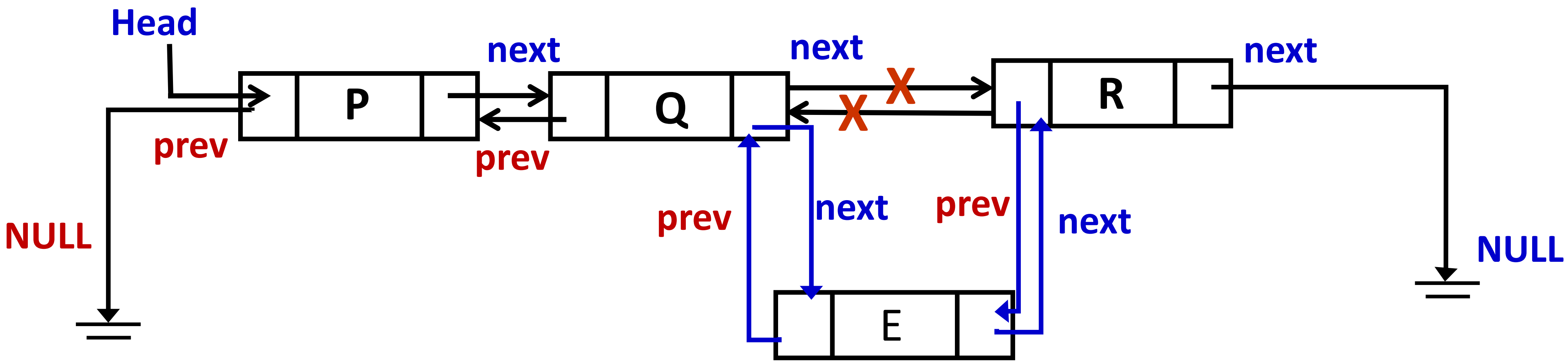


Insertion at the End (2/2)

```
void insertEndDouble(struct node** head_ref, int new_info)
{
       struct node* new_node = (struct Node*)malloc(sizeof(struct node));
       struct node* last = *head;
       new_node->info = new_info;
       new_node->next = NULL;
       if (*head == NULL)
             {
             new_node->prev = NULL;
             *head = new_node;
             return;
             }
        while (last->next != NULL)
             last = last->next;
        last->next = new_node;
        new_node->prev = last;
      return;
}
```

Insertion after a Given Node (1/2)

```
1. Check if the given prev_node is NULL
2. Allocate new node
3. Put in the data
4. Make next of new node as next of prev_node
5. Make the next of prev_node as new_node
6. Make prev_node as previous of new_node
7. Change previous of new_node's next node
```

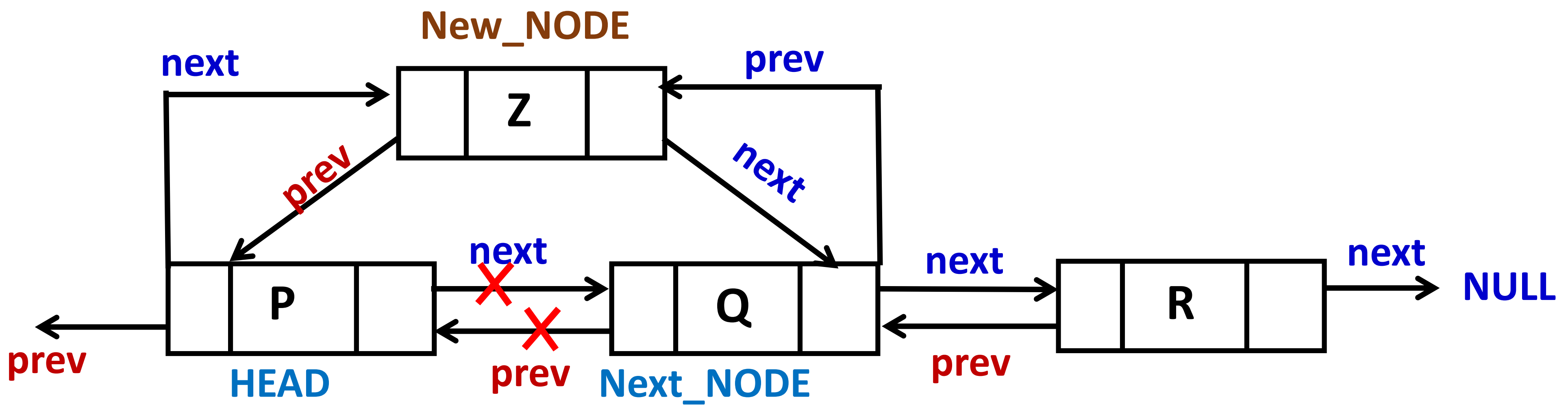


Insertion after a Given Node (2/2)

```
void insertAfter(struct node* given_node, int new_info)
{
        if (given_node == NULL) {
             printf(“error! cannot be NULL");
        return;
    }
        struct node* new_node = (struct Node*)malloc(sizeof(struct node));
        new_node->info = new_info;
        new_node->next = given_node->next;
        given_node->next = new_node;
        new_node->prev = given_node;
        if (new_node->next != NULL)
             new_node->next->prev = new_node;
}
```

Insertion before a Given Node (1/2)

```
1. Check if the given next_node is NULL
2. Allocate new node
3. Put in the new information in the new node
4. Make prev of new node as prev of next_node
5. Make the prev of next_node as new_node
6. Make next_node as next of new_node
7. Change next of new_node's previous node
8. If the prev of new_node is NULL, it will be the new head node
```



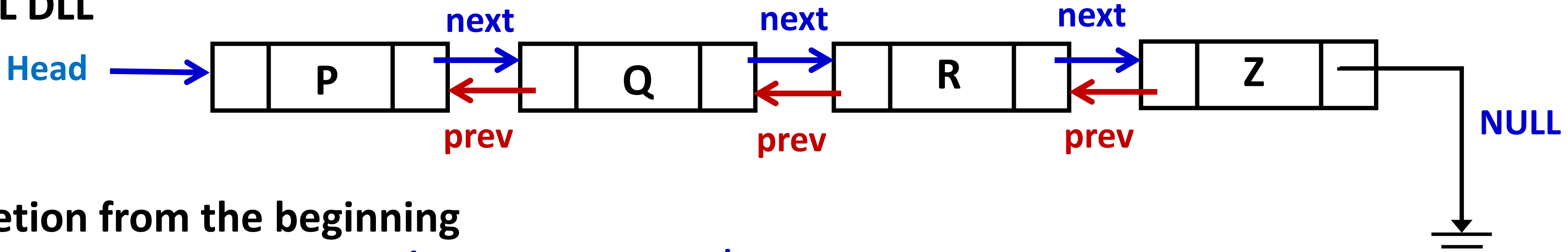
Insertion before a Given Node (2/2)

```
void insertBefore(struct node** head, struct node* next_node, int new_info)
{
    if (next_node == NULL) {
        printf("the given next node cannot be NULL");
        return;
    }
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->info = new_info;
    new_node->prev = next_node->prev;
    next_node->prev = new_node;
    new_node->next = next_node;
    if (new_node->prev != NULL)
        new_node->prev->next = new_node;
    else
        (*head) = new_node;
}
```

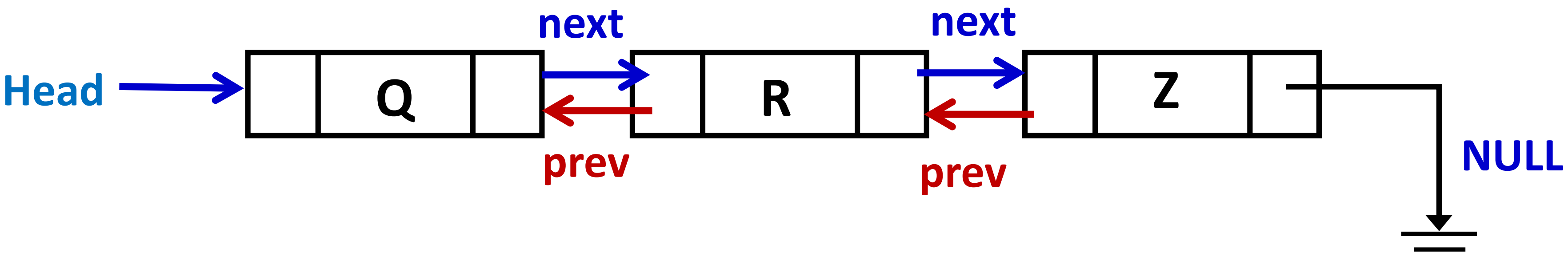
Deletion from a DLL

Deletion from Beginning, Middle and End

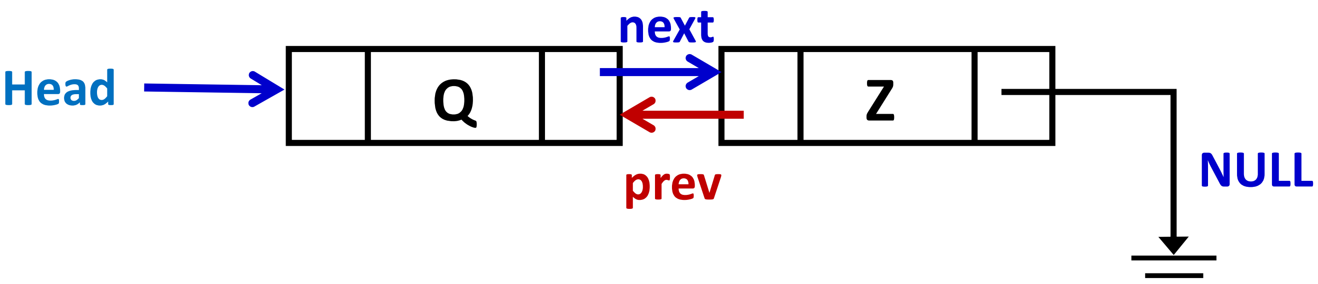
ORIGINAL DLL



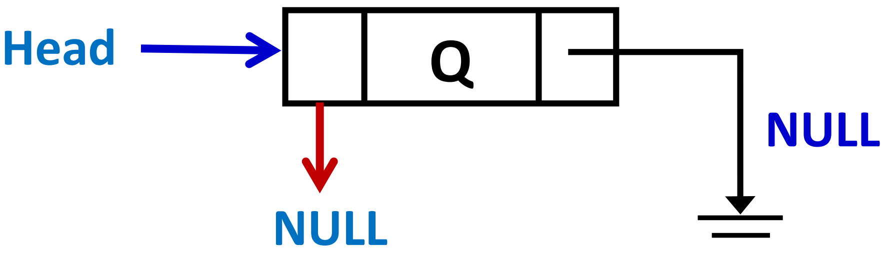
After deletion from the beginning



After deletion from the middle



After deletion of last node



Deletion from a DLL

1. Let the node to be deleted be *target*.
2. If node to be deleted is the head, then change the head pointer to next (current head)

   `if headnode == target then headnode = del.nextNode`
3. Set *next* of previous to *target*, if previous to *target* exists i.e., change next only if node to be deleted is NOT the last node

   ```
   if target.nextNode != NULL
          target.nextNode.previousNode = target.previousNode
   ```
4. Change prev only if node to be deleted is NOT the first node i.e., set *prev* of next to *target*, if next to *target* exists

   `if target.previousNode != NULL target.nextNode.nextNode = target.next`
5. Finally, free the memory occupied by del

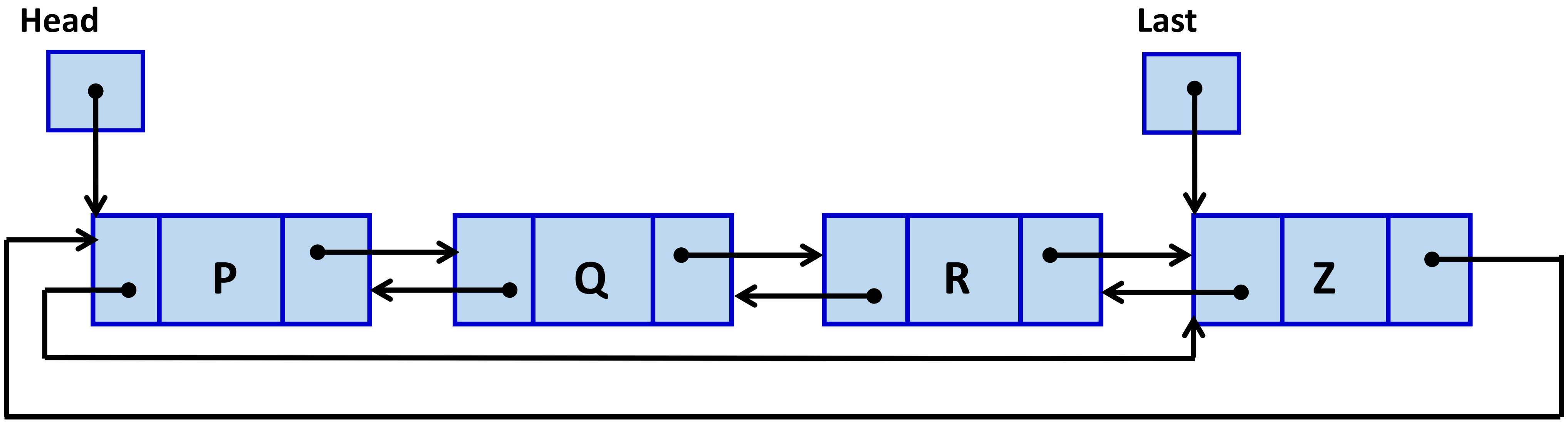
Deletion from a DLL

```
void deleteNode(struct node** head, struct node* target)
{
        if (*head == NULL || target == NULL)
              return;
        if (*head == target)
              *head = target->next;
        if (target->next != NULL)
              target->next->prev = target->prev;

    if (target->prev != NULL)
              target->prev->next = target->next;
              free(del);
    return;
}
```

Doubly Circular Linked List

- In doubly linked list each node contains three parts:
  - **Next**: It is a pointer field that contains the address of the next node
  - **Previous**: It is a pointer field that contains the address of the previous node
  - **Information**: It contains the actual data
- Next of the last node points to the first node
- Previous of the first node points to the last node



Thank you!