Linked List

- Based on the position of the new node being inserted, the insertion is categorized into the following categories.

1)At the beginning of the linked list

2)At the end of the Linked List

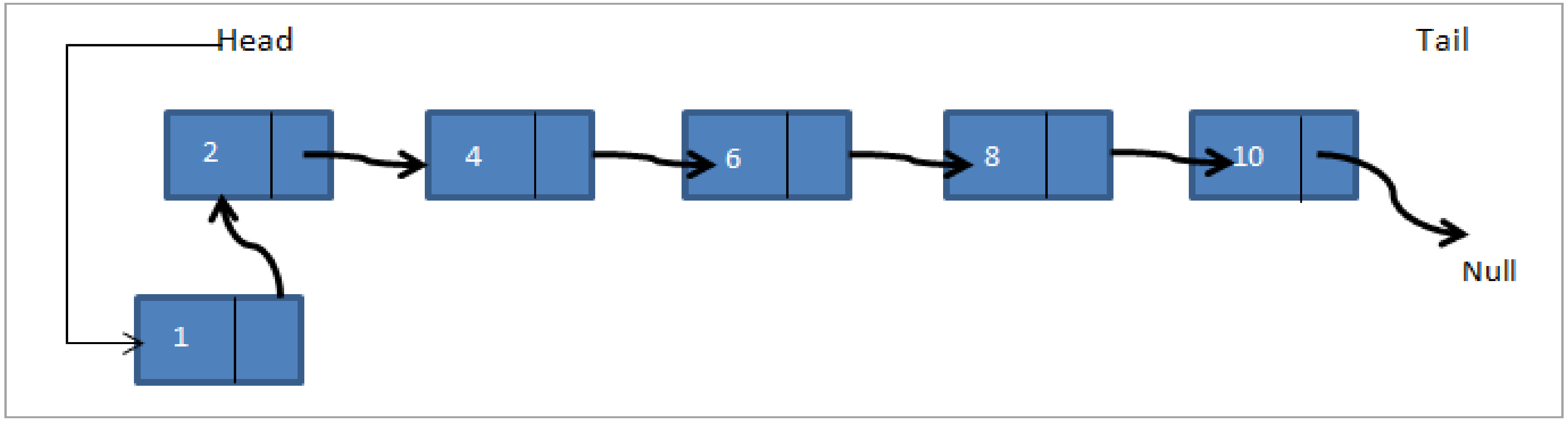
3) At the specified node(After the given Node)

Operations

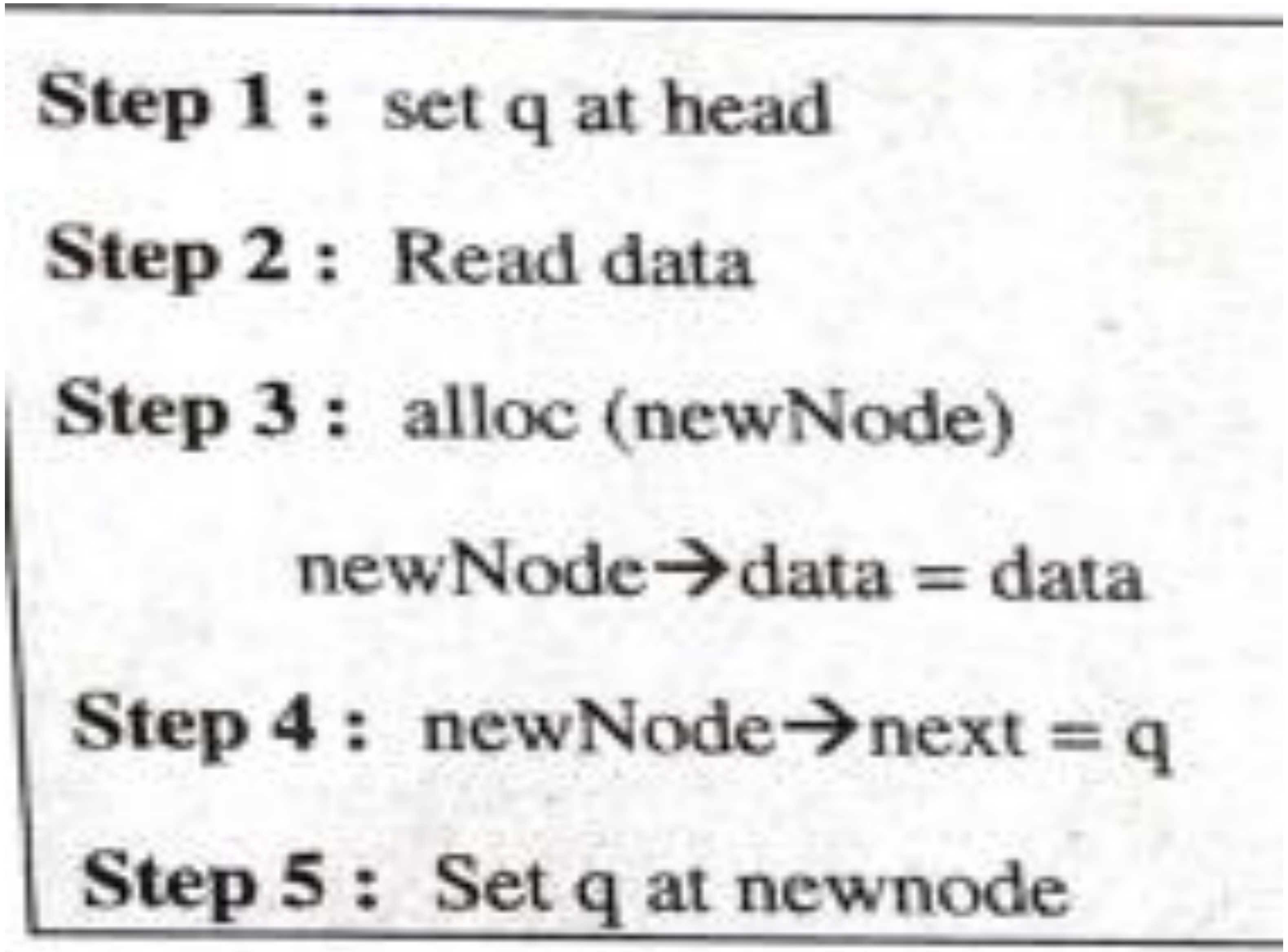
Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.

Insertion at beginning

A linked list is shown below 2->4->6->8->10.
If we want to add a new node 1, as the first node of the list, then the head pointing to node 2 will now point to 1 and the next pointer of node 1 will have a memory address of node 2 as shown in the below figure.



Thus the new linked list becomes 1->2->4->6->8->10.



```
Step 1 :  set q at head
Step 2 :  Read data
Step 3 :  alloc (newNode)
          newNode→data = data
Step 4 :  newNode→next = q
Step 5 :  Set q at newnode
```

```
void addatbeg ( struct node **q, int num )
{
    struct node *temp, *newnode ;
    newnode = malloc ( sizeof ( struct node ) ) ;
    newnode -> data = num ;
    newnode -> next = *q;
    *q = newnode;
}
```

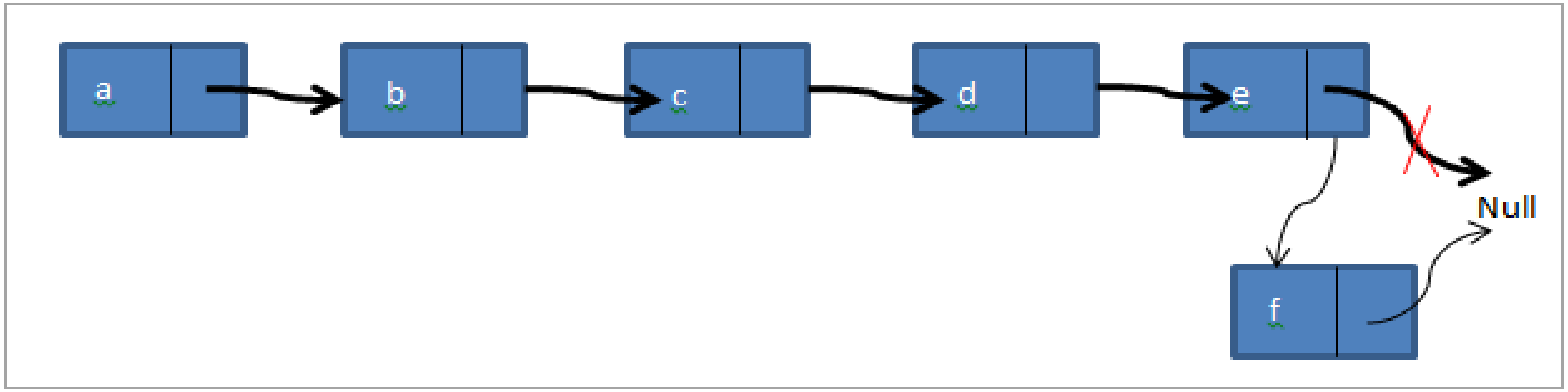
The address of previous first node is assigned to next part of new node which makes new node as first node

Insertions

Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.

Insertion

In the this case, we add a new node at the end of the linked list. Consider we have the same linked list a->b->c->d->e and we need to add a node f to the end of the list. The linked list will look as shown below after adding the node.



Thus we create a new node f. Then the tail pointer pointing to null is pointed to f and the next pointer of node f is pointed to null.

Singly Linked List

- **Inserting At End of the list**

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **NULL**).
- **Step 6 -** Set **temp** → **next** = **newNode**.

Inserting At End of the list

```
void insertAtEnd(int value)
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = value;
newNode->next = NULL;
if(head == NULL)
head = newNode;
else
```

Inserting At End of the list

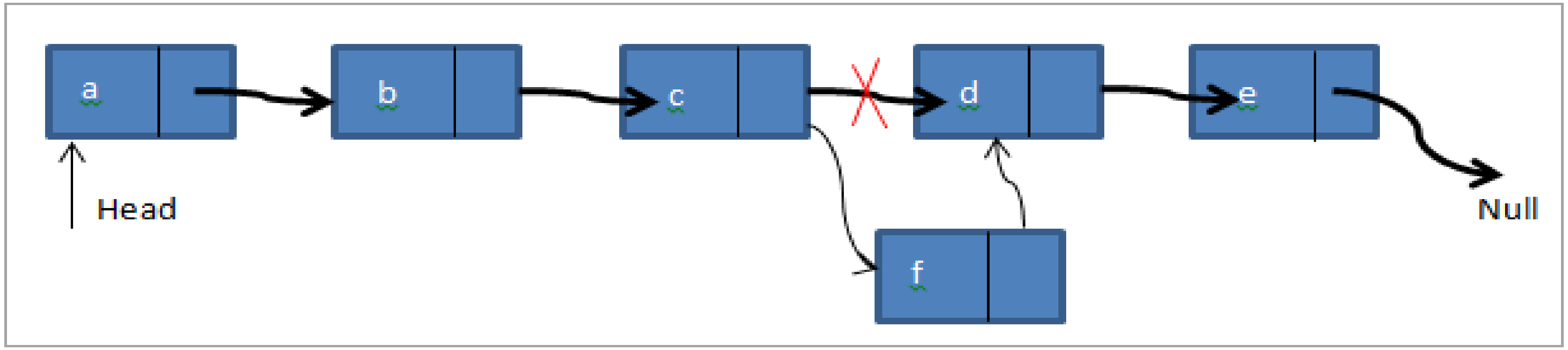
```
{
struct Node *temp = head;
while(temp->next != NULL)
temp = temp->next;
temp->next = newNode;
}
printf("\nOne node inserted!!!\n");
```

Operations

Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Insertions

Here, a node is given and we have to add a new node after the given node. In the below-linked list a->b->c->d ->e, if we want to add a node f after node c then the linked list will look as follows:



Thus in the above diagram, we check if the given node is present. If it's present, we create a new node f. Then we point the next pointer of node c to point to the new node f. The next pointer of the node f now points to node d.

Singly Linked List

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode** → **next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Singly Linked List

- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).

- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.

- **Step 7 -** Finally, Set '**newNode** → **next** = **temp** → **next**' and '**temp** → **next** = **newNode**'

Inserting At Specific location in the list (After a Node)

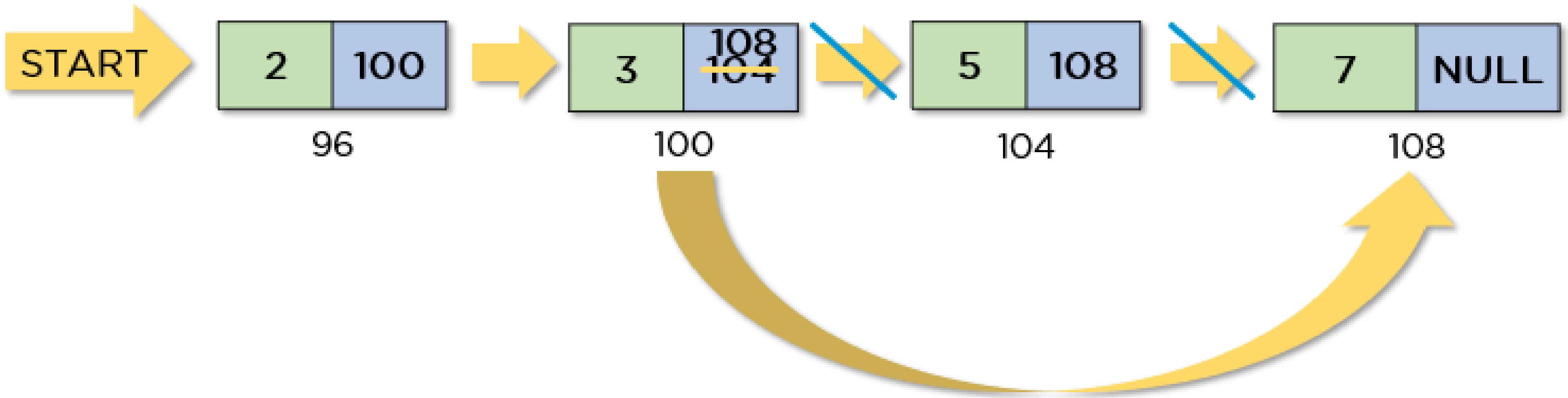
```
void insertBetween(int value, int loc1, int loc2)
{
struct Node *newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
if(head == NULL)
{
newNode->next = NULL;
head = newNode;
}
else
```

Inserting At Specific location in the list (After a Node)

```
{
struct Node *temp = head;
while(temp->data != loc1 && temp->data != loc2)
temp = temp->next;
newNode->next = temp->next;
temp->next = newNode;
}
printf("\nOne node inserted!!!\n");
}
```

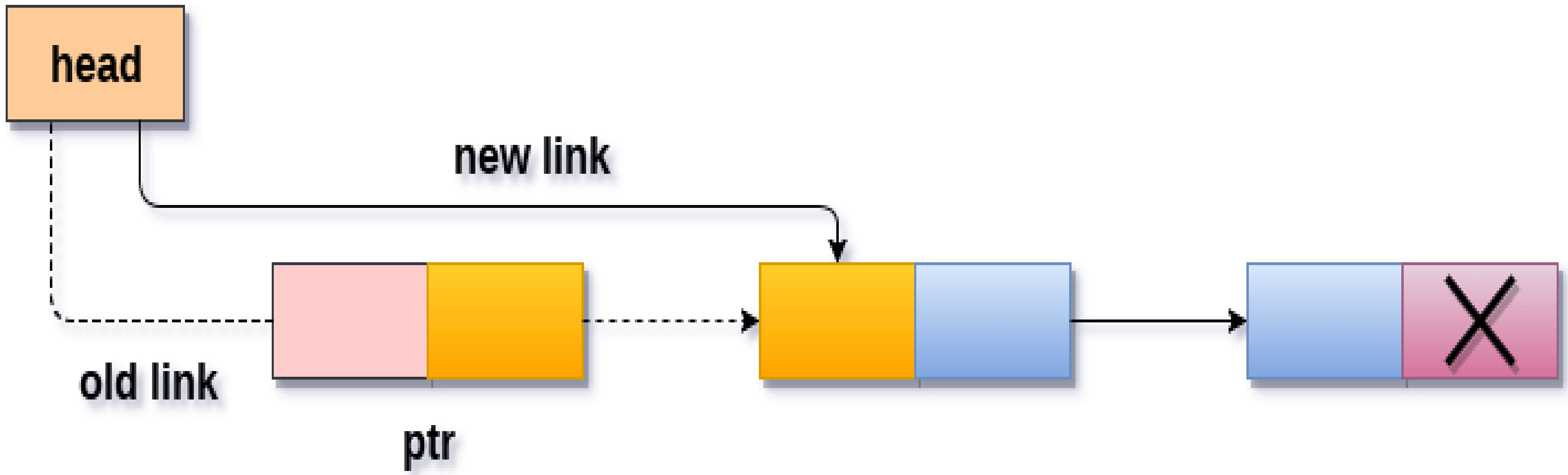
Deletion

- The Deletion of a node from a linked list can be performed at different positions.
- Based on the position of the node being deleted



Deletion

Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.



```
ptr = head
head = ptr -> next
free(ptr)
```

Deleting a node from the beginning

Singly Linked List

- **Deleting from Beginning of the list**

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

Deleting from Beginning of the list

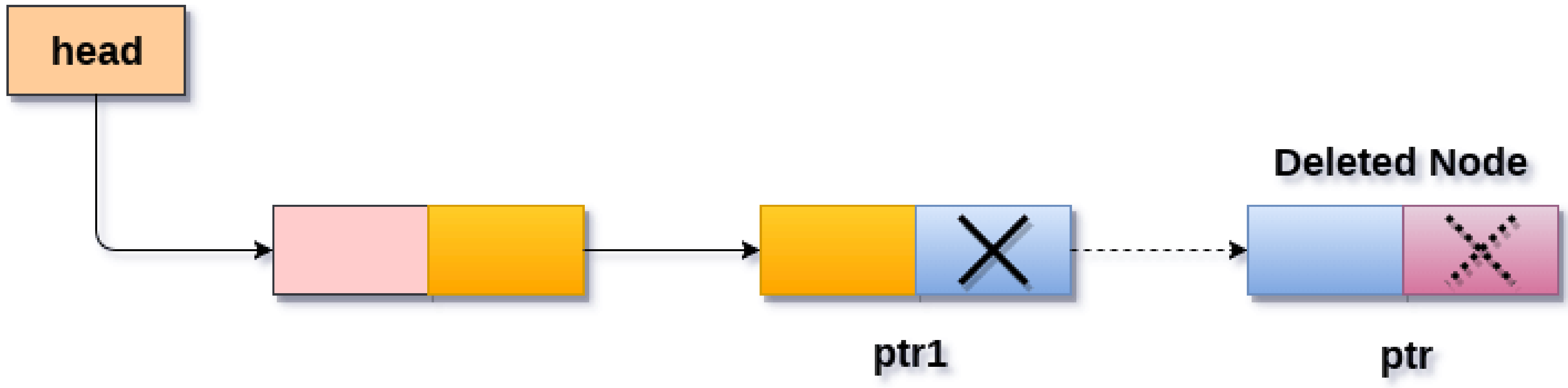
```
void removeBeginning()
{
if(head == NULL)
printf("\n\nList is Empty!!!");
else
{
struct Node *temp = head;
if(head->next == NULL)
{
head = NULL;
free(temp);
}
```

Deleting from Beginning of the list

```
else
{
head = temp->next;
free(temp);
printf("\nOne node deleted!!!\n\n");
}
}
}
```

Deletion

Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.



ptr1 -> next = Null
free(ptr)

Deleting a node from the last

Singly Linked List

- **Deleting from End of the list**

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **next** == **NULL**)

Singly Linked List

- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
- **Step 7 -** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting from End of the list

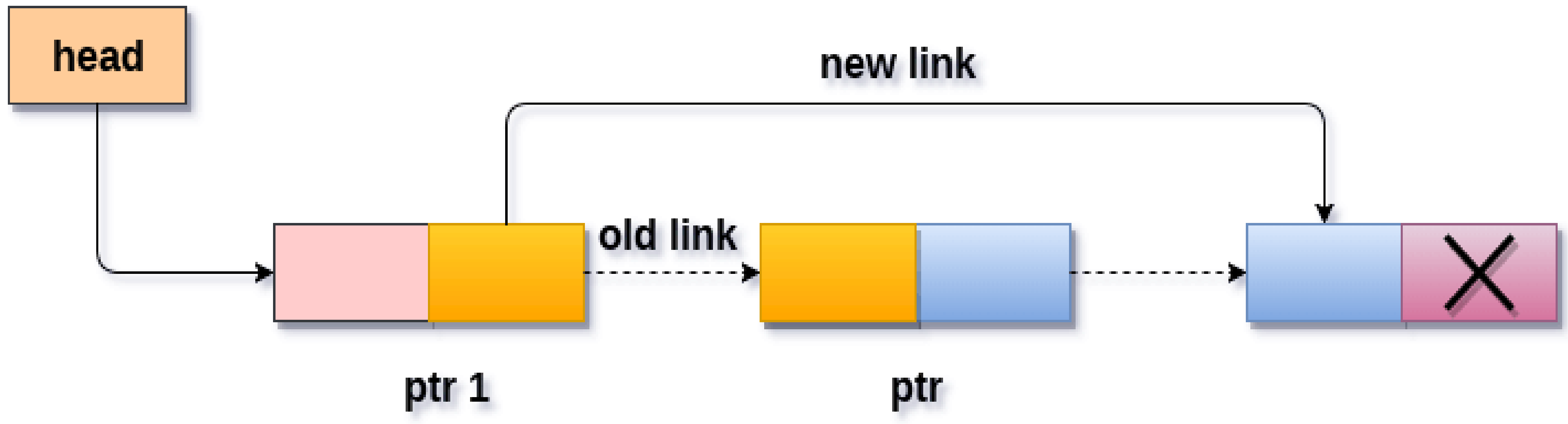
```
void removeEnd()
{
if(head == NULL)
{
printf("\nList is Empty!!!\n");
}
else
{
struct Node *temp1 = head,*temp2;
if(head->next == NULL)
head = NULL;
```

Deleting from End of the list

```
else
{
while(temp1->next != NULL)
{
temp2 = temp1;
temp1 = temp1->next;
}
temp2->next = NULL;
}
free(temp1);
printf("\nOne node deleted!!!\n\n");
}
}
```

Deletion

Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.



ptr1 -> next = ptr -> next
free(ptr)

Deletion a node from specified position

Singly Linked List

- **Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Singly Linked List

- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- ▪ **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- ▪ **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- ▪ **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).

Singly Linked List

- **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Deleting a Specific Node from the list

```
void removeSpecific(int delValue)
{
struct Node *temp1 = head, *temp2;
while(temp1->data != delValue)
{
if(temp1 -> next == NULL)
{
printf("\nGiven node not found in the list!!!");
goto functionEnd;
```

Deleting a Specific Node from the list

```
}
temp2 = temp1;
temp1 = temp1 -> next;
}
temp2 -> next = temp1 -> next;
free(temp1);
printf("\nOne node deleted!!!\n\n");
functionEnd:
}
```

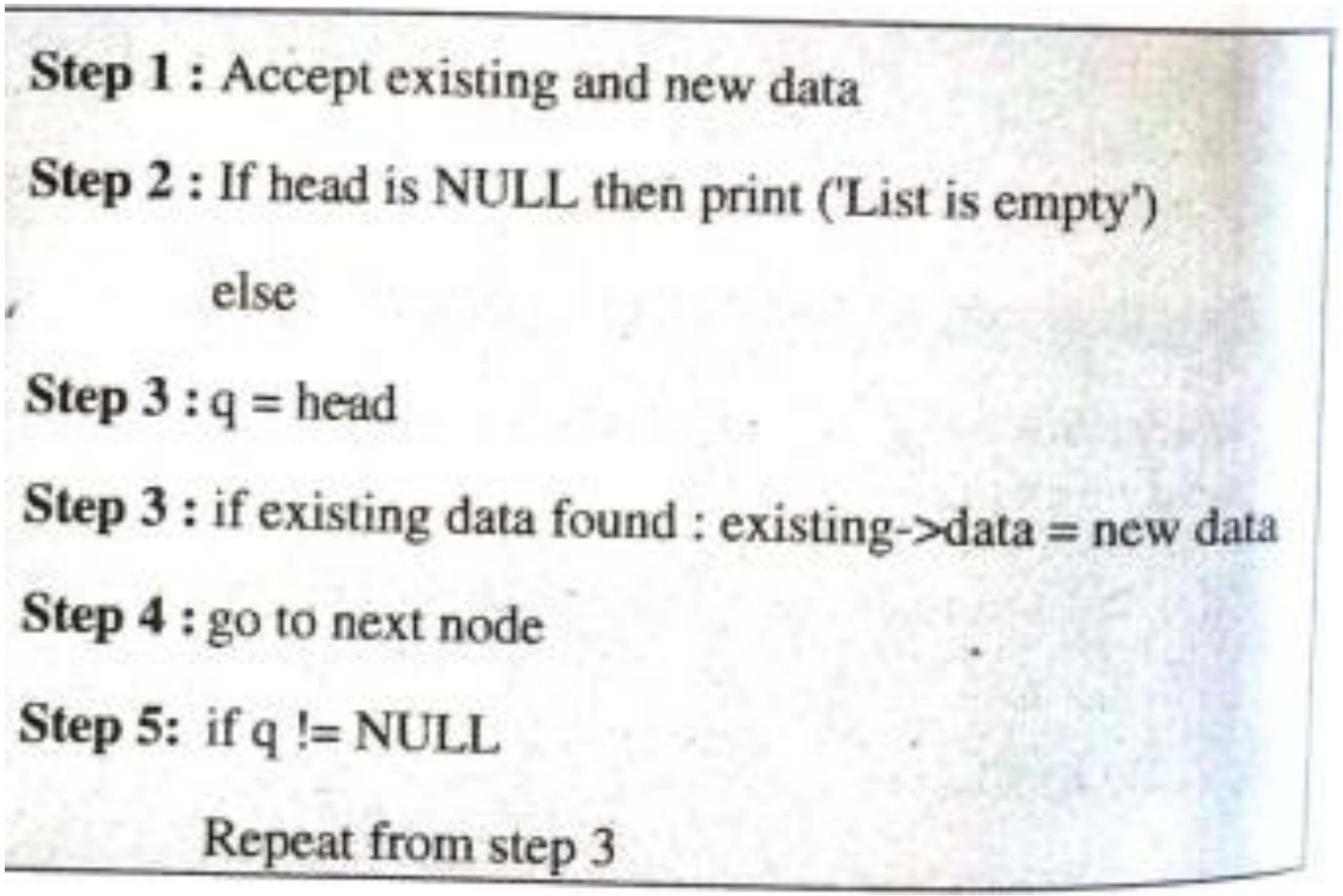
Advantages

- You can perform operations like insertion and deletion with ease
- It is a dynamic data structure, i.e., it does not have a fixed size
- It doesn’t require the movement of nodes for insertion and deletion
- It doesn’t need elements to be stored in consecutive memory spaces
- It does not waste space as it uses space according to the requirement

Disadvantages

- It requires more storage space because it also stores the next pointer with data
- If you have to reach any node, then you have to go through every node before it
- You can not traverse it from anywhere but the head node
- It requires a different amount of time to access any elements
- Sorting is complex in this linked list

Update in SLL

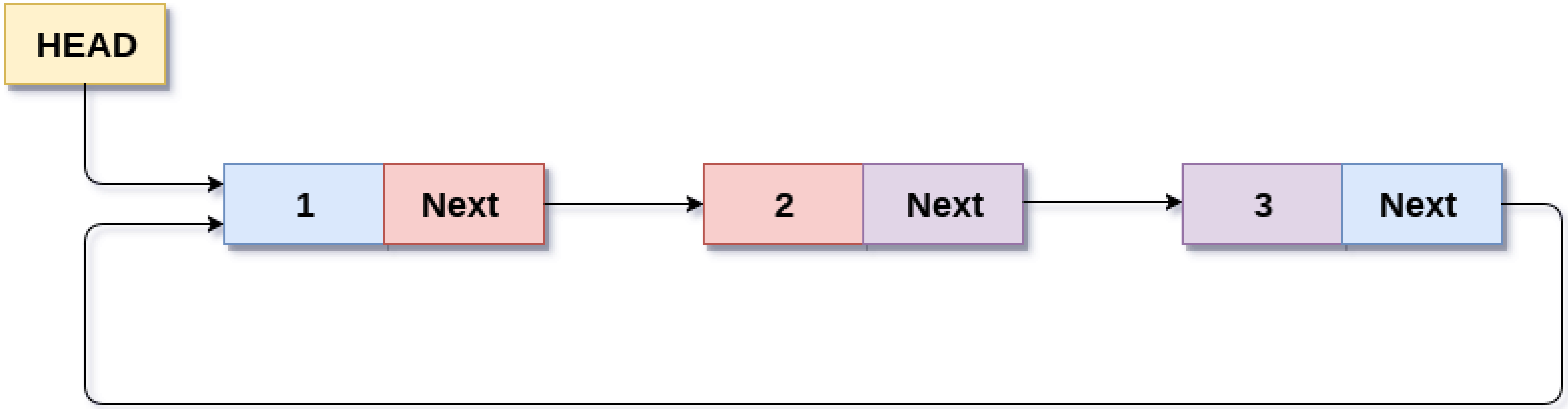


```
Step 1 : Accept existing and new data
Step 2 : If head is NULL then print ('List is empty')
         else
Step 3 : q = head
Step 3 : if existing data found : existing->data = new data
Step 4 : go to next node
Step 5: if q != NULL
         Repeat from step 3
```

```
void display ( struct node *q, int existing, int new)
{
/* address of head is passed to q */
    while ( q != NULL )
    {
        if(q -> data == existing)
        q->data =  new;
        q = q -> next ;
    }
}
```

Circular Linked List

- A circular Linked list is a unidirectional linked list. So, you can traverse it in only one direction. But this type of linked list has its last node pointing to the head node. So while traversing, you need to be careful and stop traversing when you revisit the head node.
- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

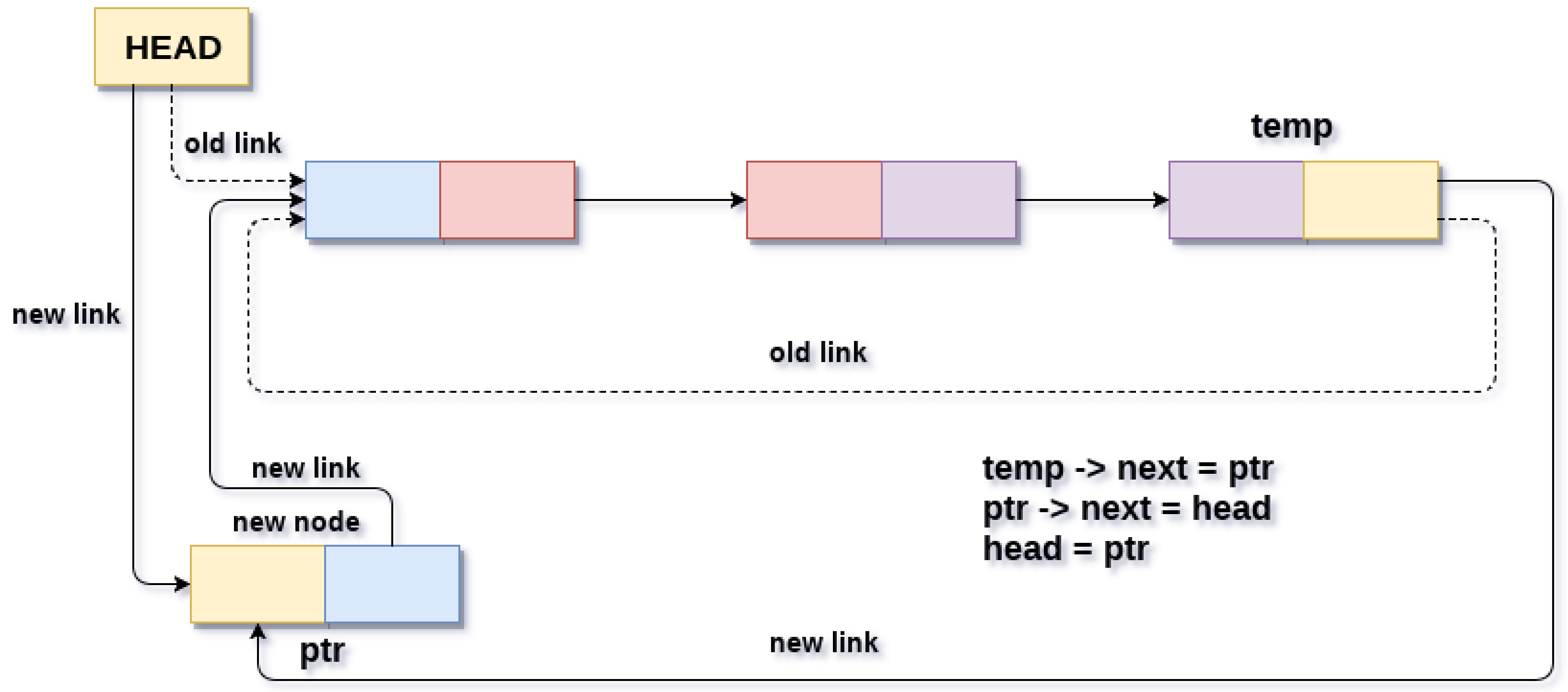


Circular Singly Linked List

Operations

- ❖Insertion
- ❖Deletion
- ❖Display

Insertion into circular singly linked list at beginning

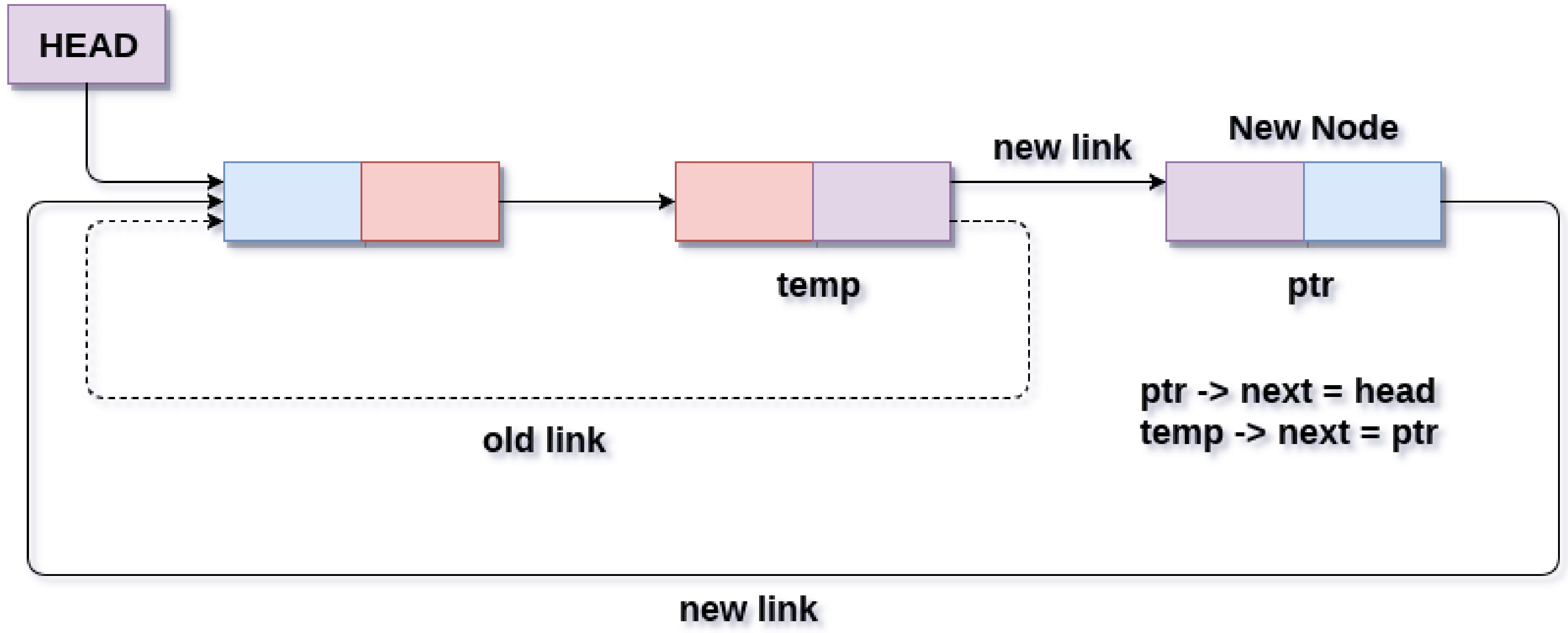


temp -> next = ptr
ptr -> next = head
head = ptr

Insertion into circular singly linked list at beginning

```
Step 1: IF PTR = NULL
          Write OVERFLOW
          Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = HEAD
Step 9: SET TEMP → NEXT = NEW_NODE
Step 10: SET HEAD = NEW_NODE
Step 11: EXIT
```

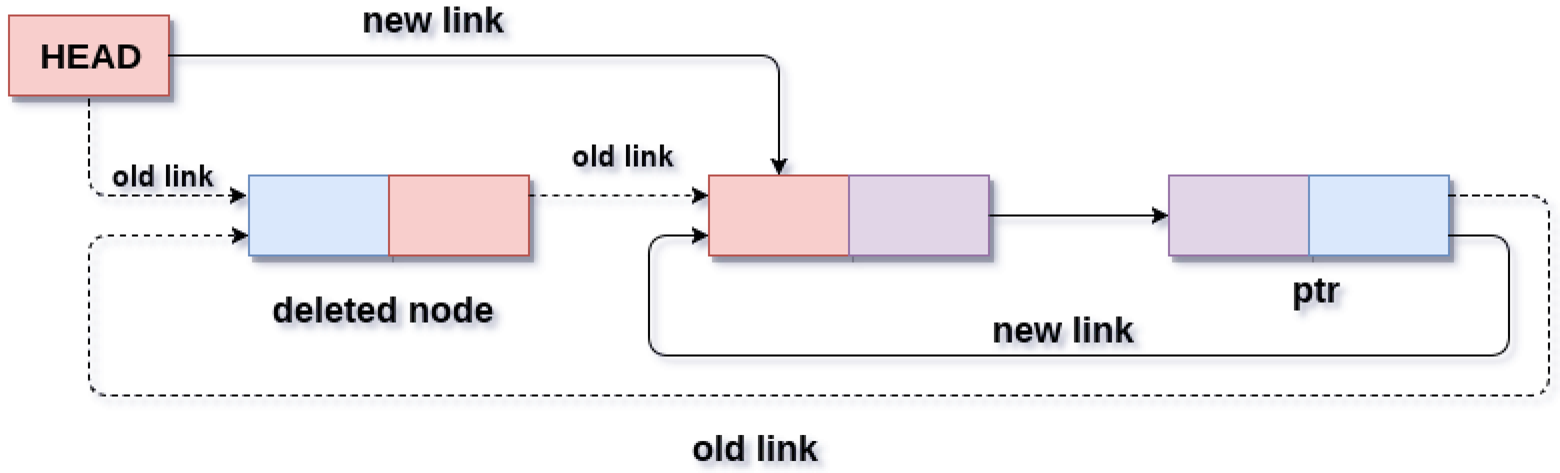
Insertion into circular singly linked list at the end



Insertion into circular singly linked list at end

```
• Step 1: IF PTR = NULL
•   Write OVERFLOW
     Go to Step 1
    [END OF IF]
• Step 2: SET NEW_NODE = PTR
• Step 3: SET PTR = PTR -> NEXT
• Step 4: SET NEW_NODE -> DATA = VAL
• Step 5: SET NEW_NODE -> NEXT = HEAD
• Step 6: SET TEMP = HEAD
• Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD
• Step 8: SET TEMP = TEMP -> NEXT
• [END OF LOOP]
• Step 9: SET TEMP -> NEXT = NEW_NODE
• Step 10: EXIT
```

Deletion in circular singly linked list at beginning

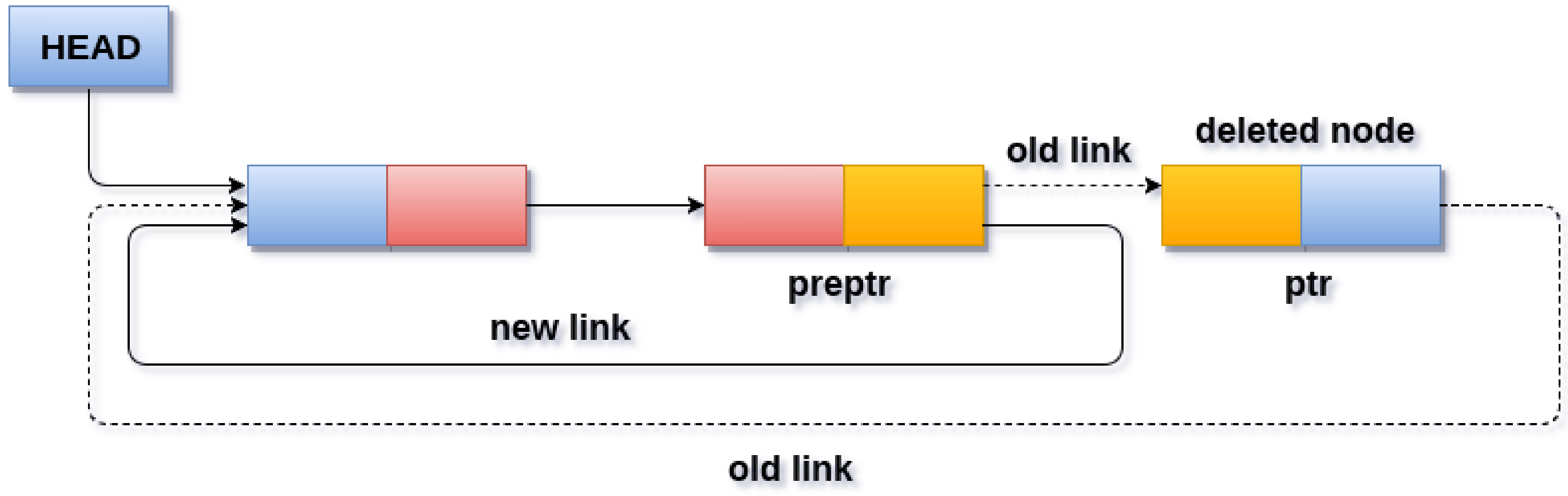


```
ptr -> next = head -> next
free head
head = ptr -> next
```

Deletion in circular singly linked list at beginning

- **Step 1:** IF HEAD = NULL
- Write UNDERFLOW
  Go to Step 8
  [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Step 4 while PTR → NEXT != HEAD
- **Step 4:** SET PTR = PTR → next
- [END OF LOOP]
- **Step 5:** SET PTR → NEXT = HEAD → NEXT
- **Step 6:** FREE HEAD
- **Step 7:** SET HEAD = PTR → NEXT
- **Step 8:** EXIT

Deletion in Circular singly linked list at the end



preptr -> next = head
free ptr

Deletion in circular singly linked list at end

- **Step 1:** IF HEAD = NULL
-   Write UNDERFLOW
     Go to Step 8
    [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT
- [END OF LOOP]
- **Step 6:** SET PREPTR -> NEXT = HEAD
- **Step 7:** FREE PTR
- **Step 8:** EXIT

Searching in CLL

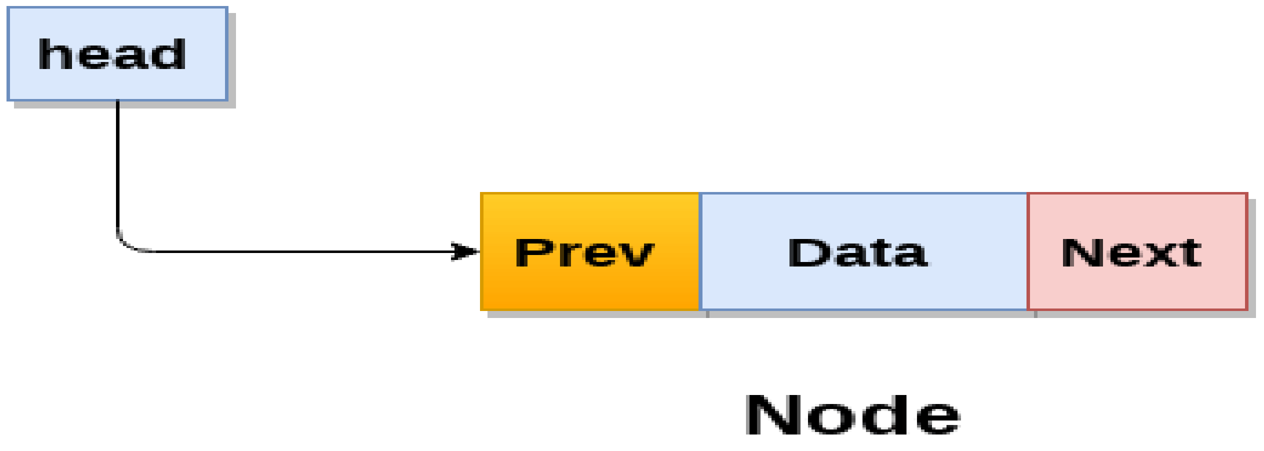
- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL
- WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
- **STEP 4:** IF HEAD → DATA = ITEM
- WRITE i+1 RETURN [END OF IF]**STEP 5:** REPEAT STEP 5 TO 7 UNTIL PTR->next != head
- **STEP 6:** if ptr → data = item
- write i+1
  RETURN
  End of IF
- **STEP 7:** I = I + 1
- **STEP 8:** PTR = PTR → NEXT
- [END OF LOOP]
- **STEP 9:** EXIT

Traversing in CLL

- **STEP 1:** SET PTR = HEAD
- **STEP 2:** IF PTR = NULL
- WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
- **STEP 4:** REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
- **STEP 5:** PRINT PTR → DATA
- **STEP 6:** PTR = PTR → NEXT
- [END OF LOOP]
- **STEP 7:** PRINT PTR→ DATA
- **STEP 8:** EXIT

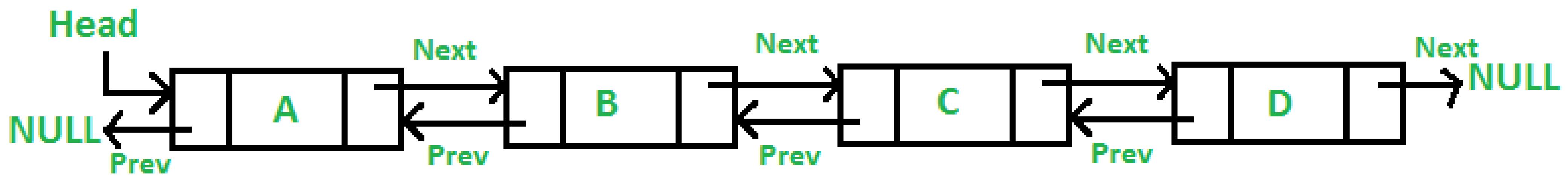
Doubly Linked List

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).
- A sample node in a doubly linked list is shown in the figure.



DLL

- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Advantages of DLL

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a 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 the previous pointer.

Disadvantages of DLL

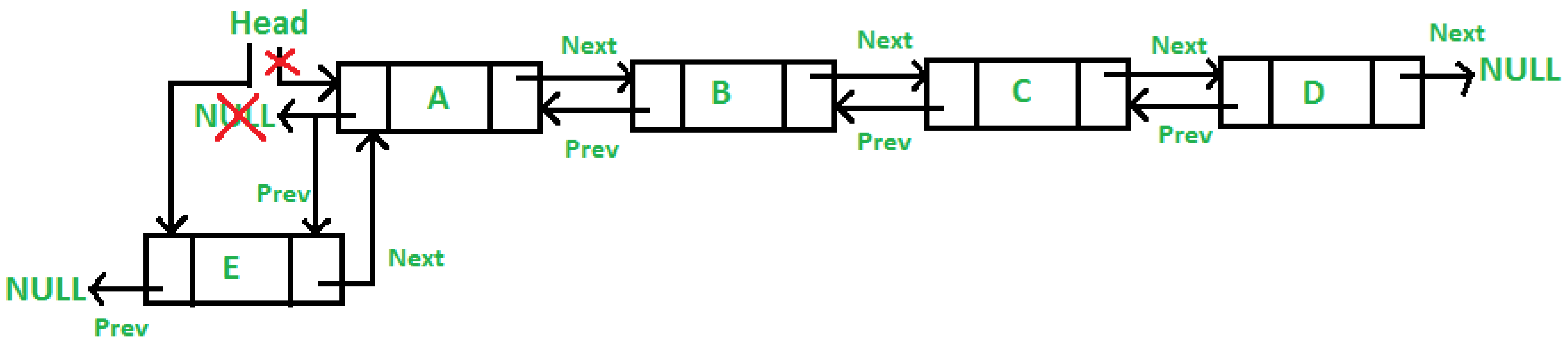
- Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer.
- All operations require an extra pointer previous to be maintained.
- For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

Operations

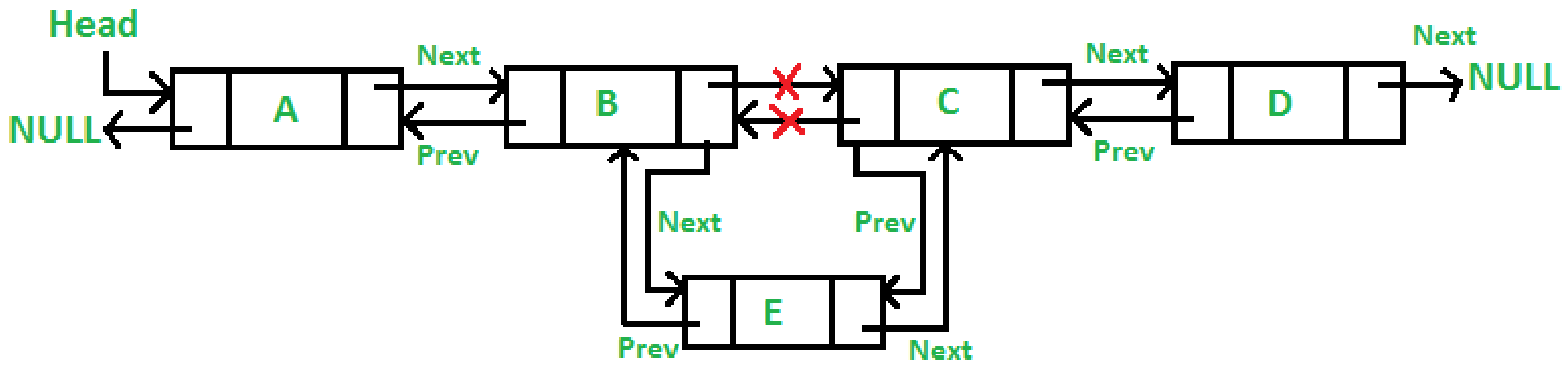
- **Operations on Double Linked List**
- In a double linked list, we perform the following operations...
- 1. Insertion
- 2. Deletion
- 3. Display

Insertion:

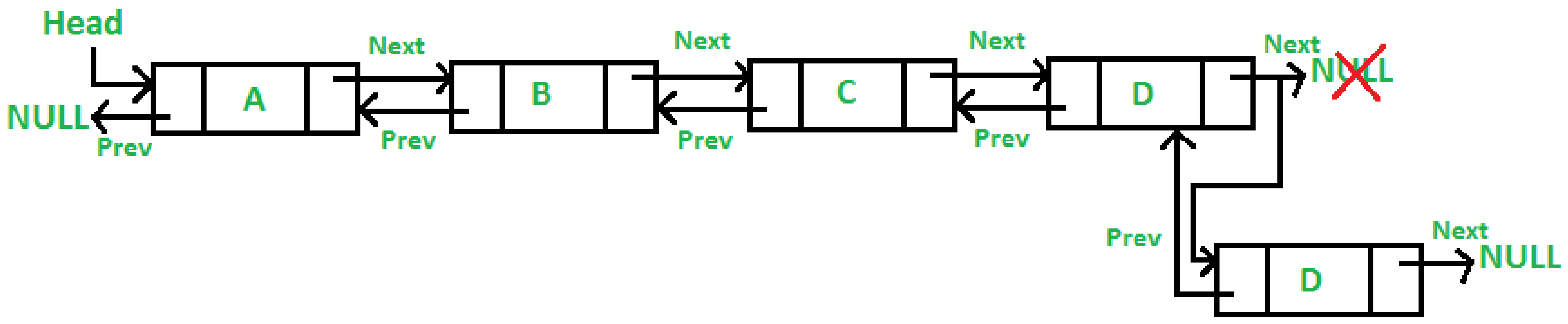
Add a node at the front:



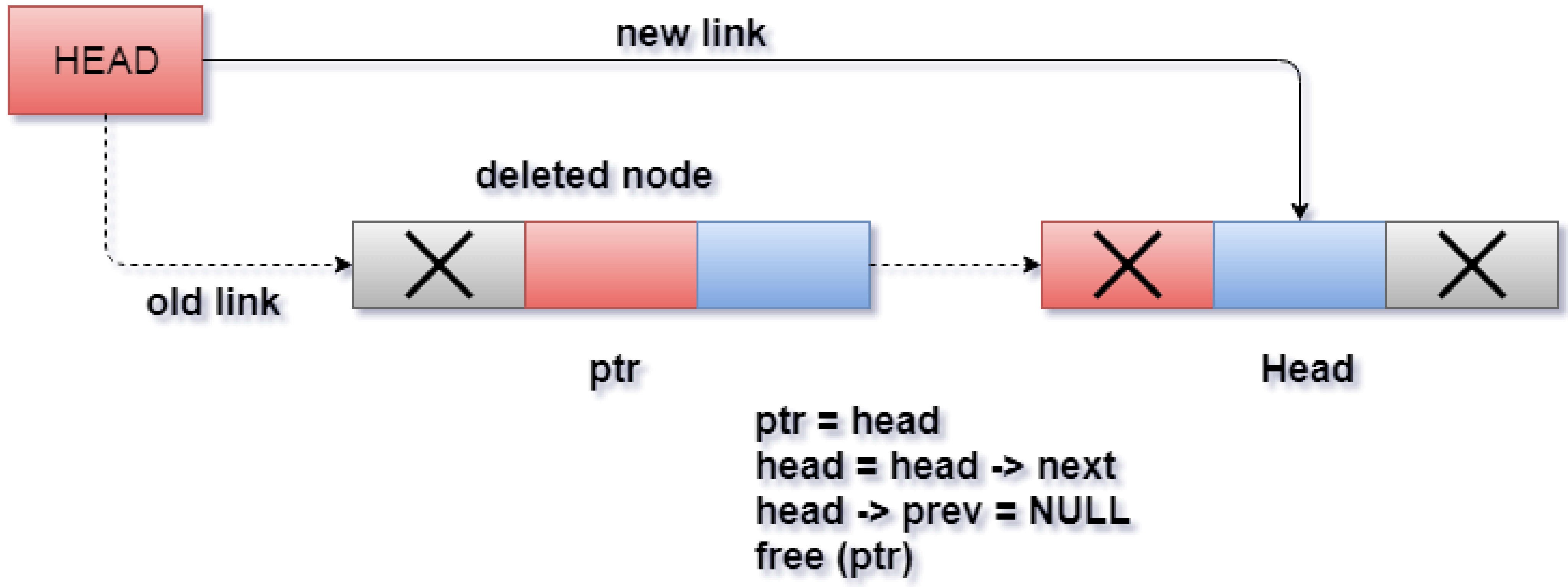
Add a node after a given node



Add a node at the end



Deletion in doubly linked list at the beginning



```
ptr = head
head = head -> next
head -> prev = NULL
free (ptr)
```

Deletion in doubly linked list from beginning

```
STEP 1: IF HEAD = NULL
        WRITE UNDERFLOW
        GOTO STEP 6

STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD → NEXT
STEP 4: SET HEAD → PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT
```

•