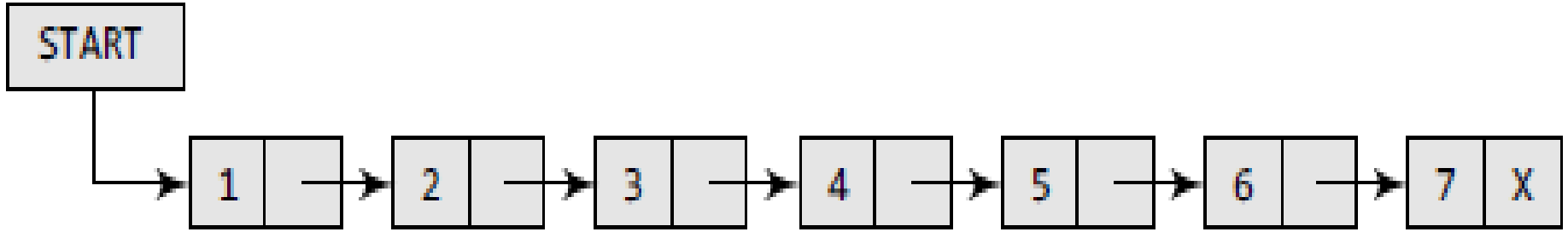
Chapter 3

Linked List

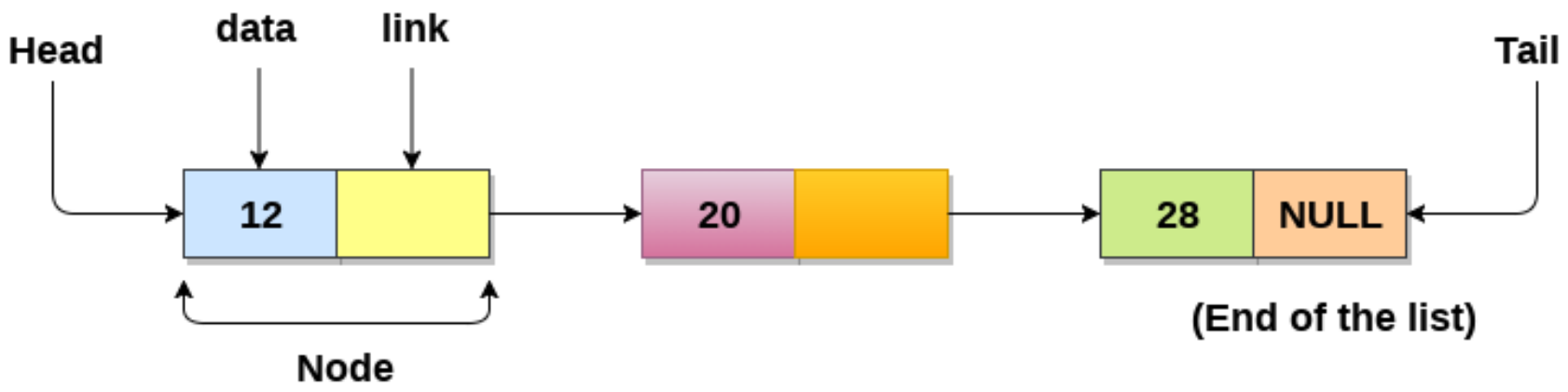


Linked Lists

- A linked list, in simple terms, is a linear collection of data elements.
- These data elements are called *nodes*.
- Linked list is a data structure which in turn can be used to implement other data structures.
- Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

In C, we can implement a linked list using the following code:

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



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.

- list size is limited to the memory size and doesn't need to be declared in advance.

- Empty node can not be present in the linked list.

- We can store values of primitive types or objects in the singly linked list.

Real time Uses of Linked List

1. Polynomial Manipulation representation
2. Addition of long positive integers
3. Representation of sparse matrices
4. Addition of long positive integers
5. Symbol table creation
6. Mailing list
7. Memory management
8. Linked allocation of files
9. Multiple precision arithmetic etc

Why use linked list over array?

- **Array contains following limitations:**
  - The size of array must be known in advance before using it in the program.
  - Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
  - All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.
- **Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,**
  - It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
  - Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

- In the figure, we can see that the variable START is used to store the address of the first node.
- Here, in this example, START= 1, so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node.
- From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.
- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

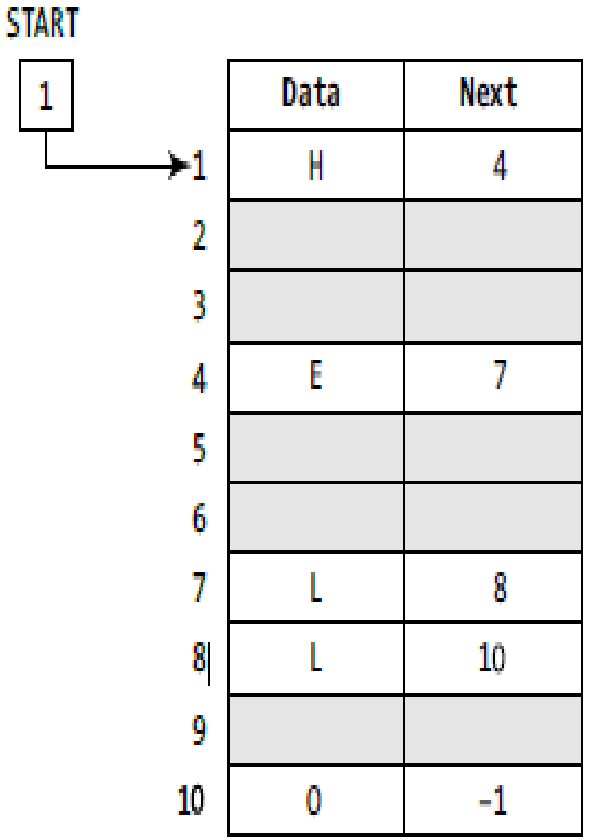


Figure 6.2 START pointing to the first element of the linked list in the memory

Basic Operations in the Linked Lists

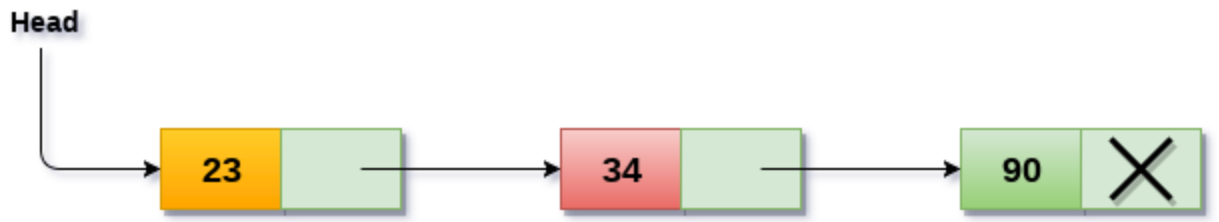
- The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key.

- These operations are performed on Singly Linked Lists as given below −
  1. **Insertion** − Adds an element at the beginning of the list.
  2. **Deletion** − Deletes an element at the beginning of the list.
  3. **Display** − Displays the complete list.
  4. **Search** − Searches an element using the given key.
  5. **Delete** − Deletes an element using the given key.

Types Of Linked List:

1. Singly Linked List

- It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- The node contains a pointer to the next node means that the node stores the address of the next node in the sequence.
- A single linked list allows the **traversal of data only in one way.** Below is the image for the same:



Operations on Singly Linked List

- **Node Creation:**

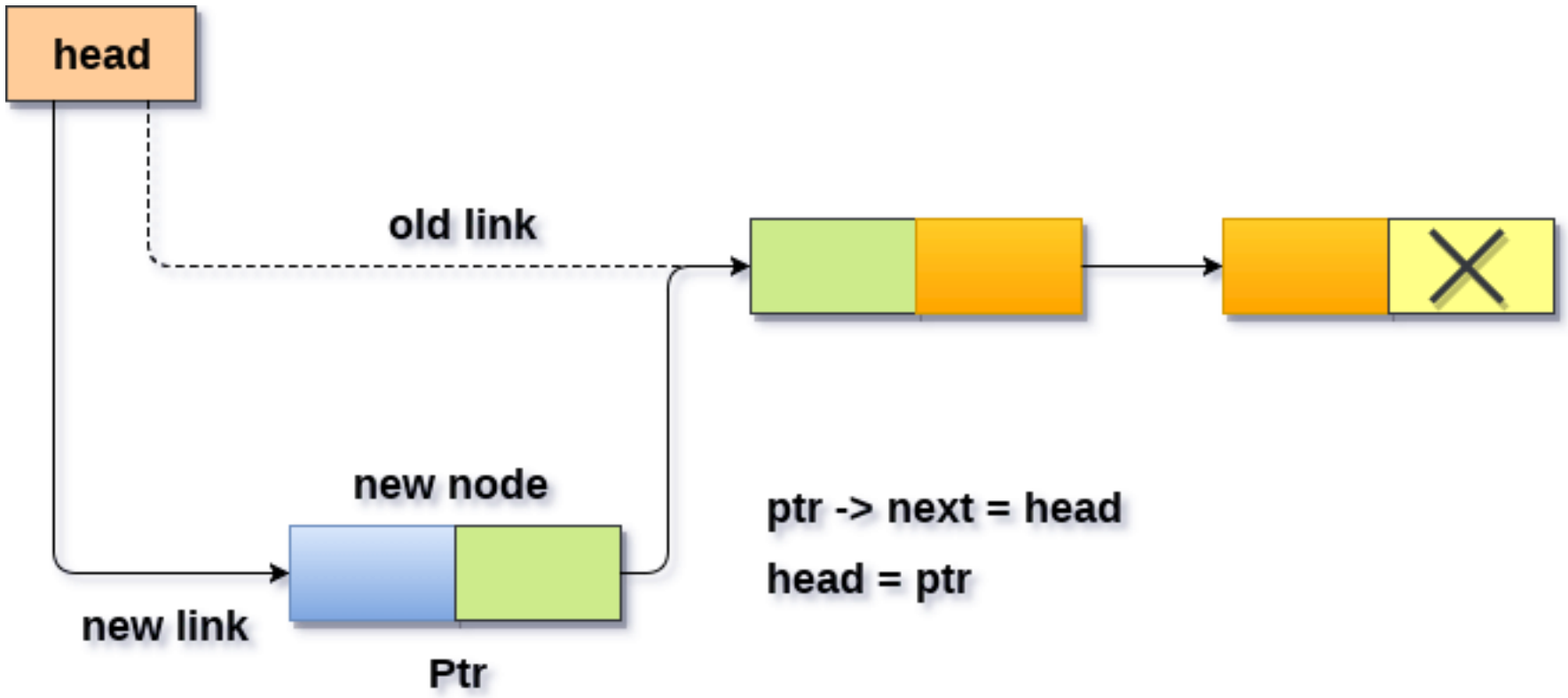
```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *) malloc(sizeof(struct node *));
```

Operations on Singly Linked List

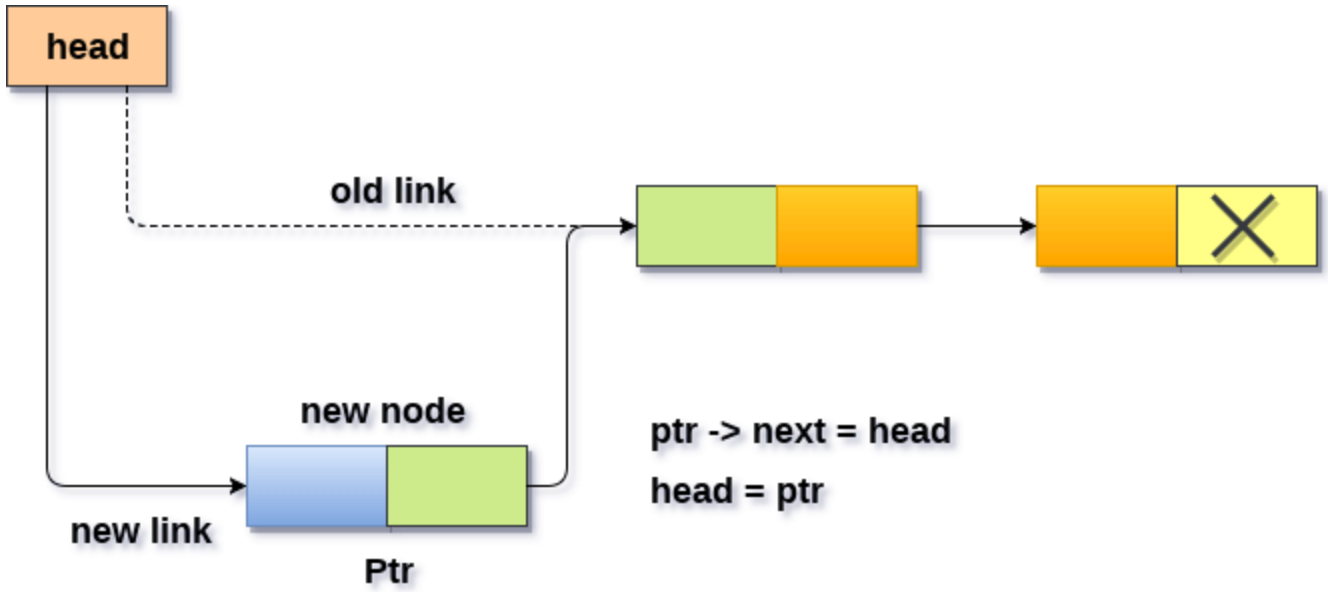
- Insertion:
  - The insertion into a singly linked list can be performed at different positions.
  - Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	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.
2	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.
3	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. .

Insertion in singly linked list at beginning



- **Algorithm:**



Step 1: IF PTR = NULL

Write OVERFLOW
Go to Step 7
[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 HEAD = NEW_NODE

Step 7: EXIT

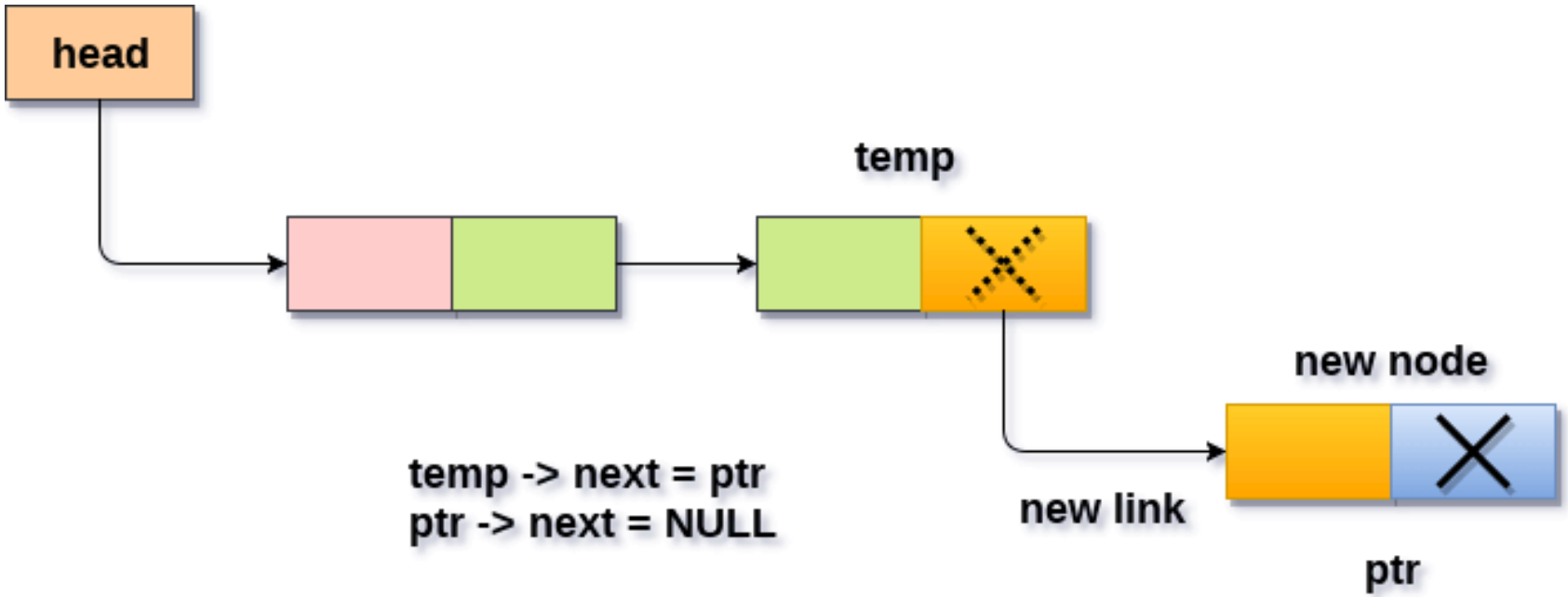
Insertion in singly linked list at the end

- In order to insert a node at the last, there are two following scenarios which need to be mentioned.
  1. The node is being added to an empty list
  2. The node is being added to the end of the linked list

The node is being added to an empty list

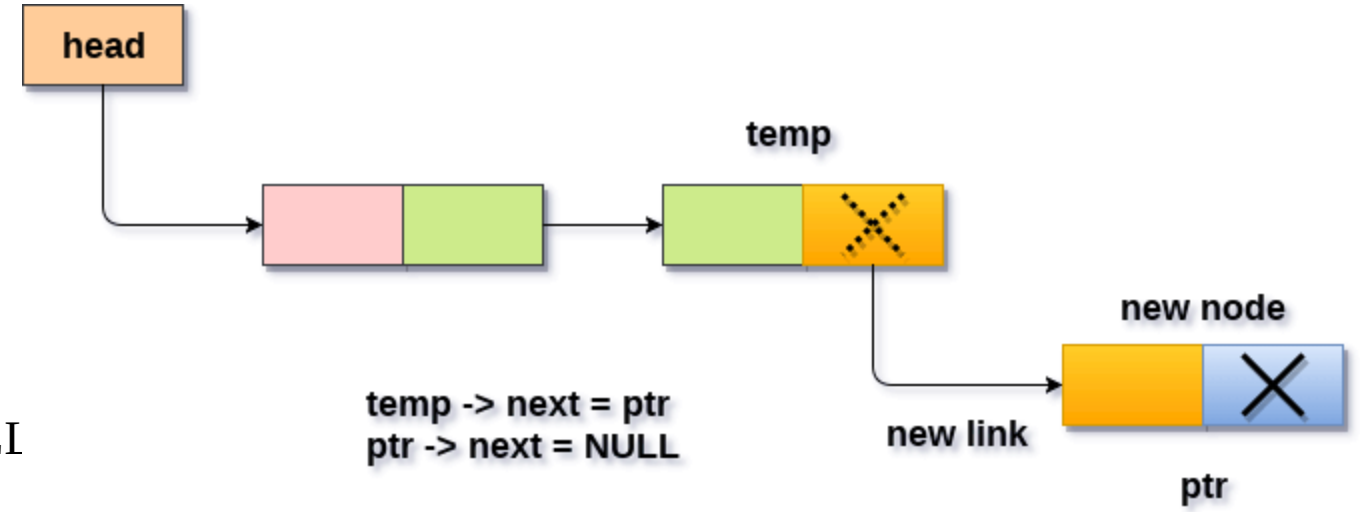
- in the first case,
  - The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.
    - ptr->data = item;
    - ptr -> next = NULL;
  - Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.
    - Head = ptr

The node is being added to the end of the linked list



Inserting node at the last into a non-empty list

- **Algorithm**
  - **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 = NULL
  - **Step 6:** SET PTR = HEAD
  - **Step 7:** Repeat Step 8 while PTR - > NEXT != NULI
  - **Step 8:** SET PTR = PTR - > NEXT
    [END OF LOOP]
  - **Step 9:** SET PTR - > NEXT = NEW_NODE
  - **Step 10:** EXIT



Inserting node at the last into a non-empty list

The node is being added to the end of the linked list

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.
  - Temp = head

- Then, traverse through the entire linked list using the statements:
  - **while** (temp→ next != NULL)
  - temp = temp → next;

- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part of the temp node (which is currently the last node of the list) point to the new node (ptr) .

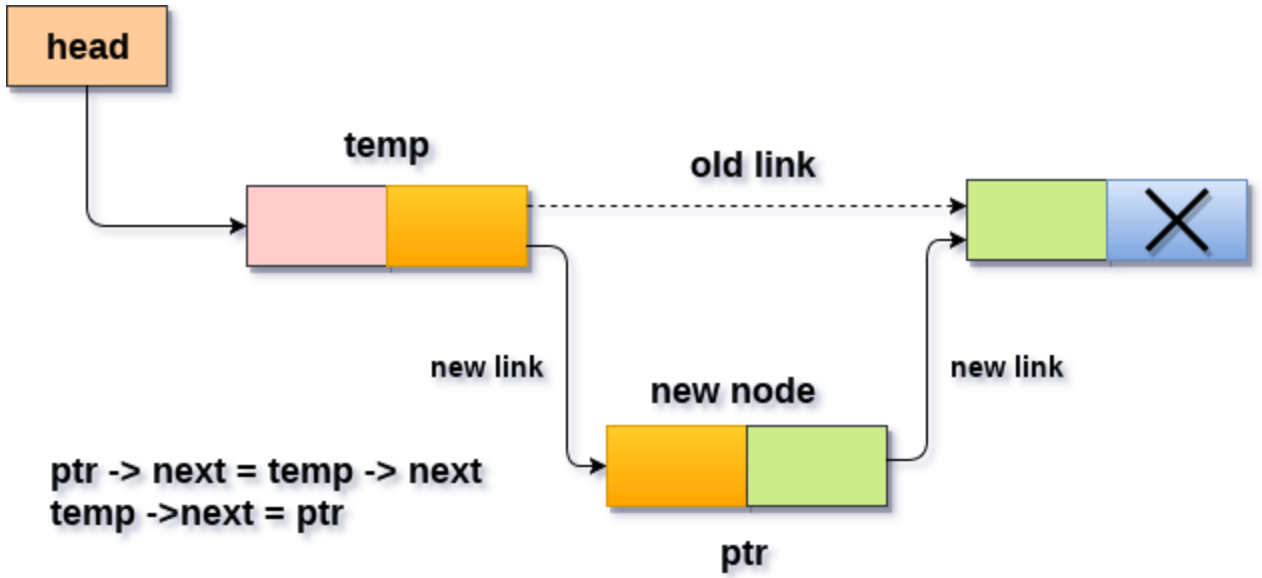
Insertion in singly linked list after specified Node

1. In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted.

- This will be done by using the following statements.

```
emp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                return;
            }

        }
```



2. Allocate the space for the new node and add the item to the data part of it.

- This will be done by using the following statements.

```
ptr = (struct node *) malloc (sizeof(struct node));
ptr->data = item;
```

3. Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). This will be done by using the following statements.

```
ptr→ next = temp → next
```

4. now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

```
temp ->next = ptr;
```

Algorithm

STEP 1: IF PTR = NULL

WRITE OVERFLOW
GOTO STEP 12
END OF IF

STEP 2: SET NEW_NODE = PTR

STEP 3: NEW_NODE → DATA = VAL

STEP 4: SET TEMP = HEAD

STEP 5: SET I = 0

STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>

STEP 7: TEMP = TEMP → NEXT

STEP 8: IF TEMP = NULL

WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12
END OF IF
END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT

STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW_NODE

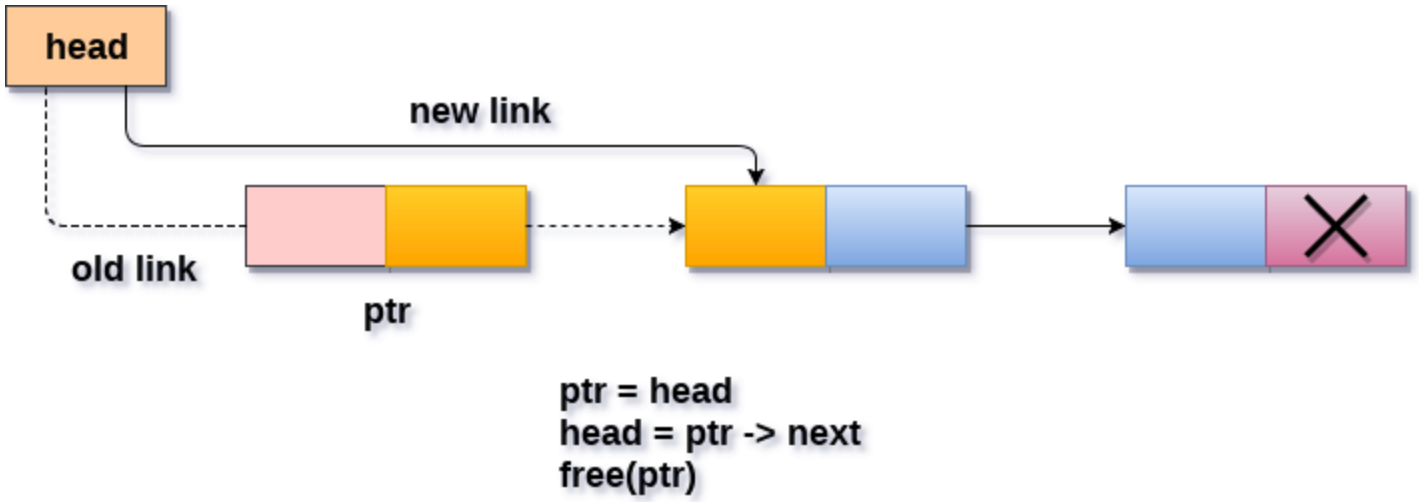
STEP 12: EXIT

Deletion and Traversing

SN	Operation	Description
1	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.
2	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.
3	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.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Deletion in singly linked list at beginning

- Algorithm
- **Step 1:** IF HEAD = NULL
- Write UNDERFLOW
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = HEAD
- **Step 3:** SET HEAD = HEAD -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** EXIT



Deletion in singly linked list at the end

- There are two scenarios in which, a node is deleted from the end of the linked list.
  1. There is only one node in the list and that needs to be deleted.
  2. There are more than one node in the list and the last node of the list will be deleted.

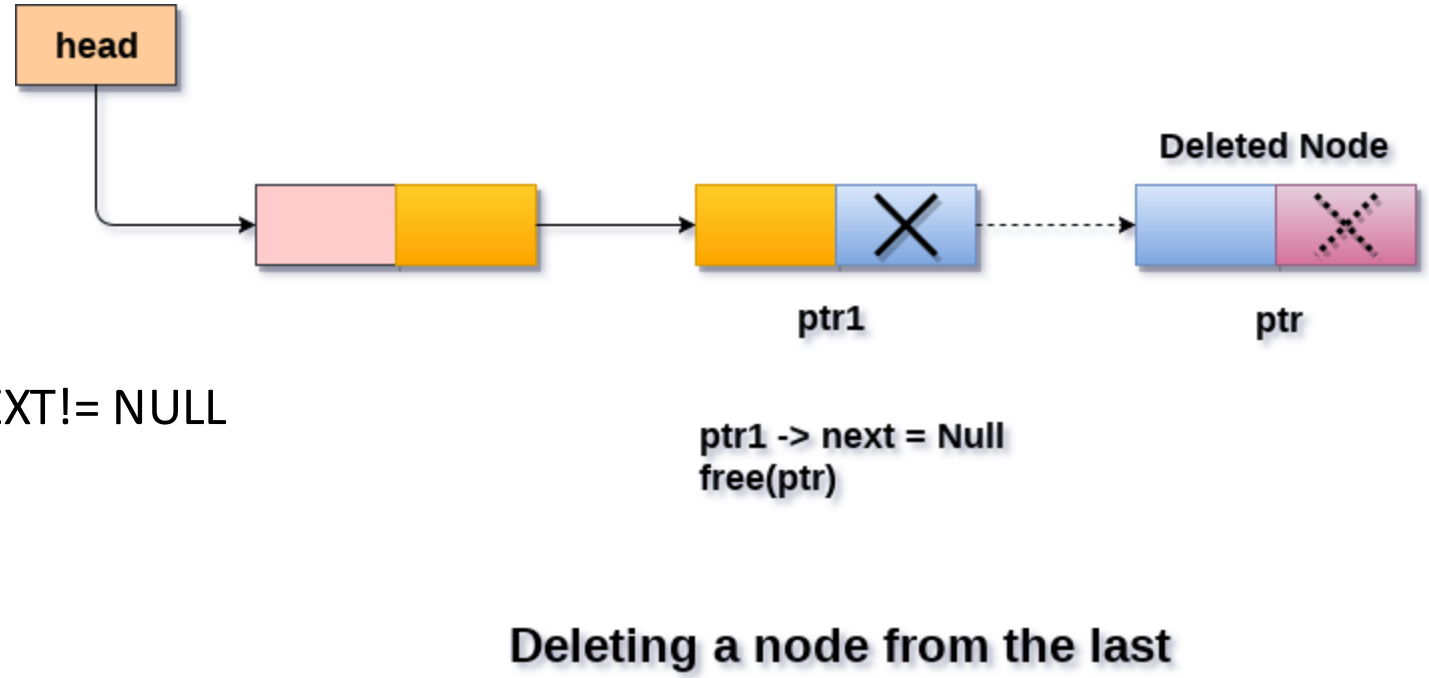
here is only one node in the list and that needs to be deleted.

- the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

```
ptr = head
        head = NULL
        free(ptr)
```

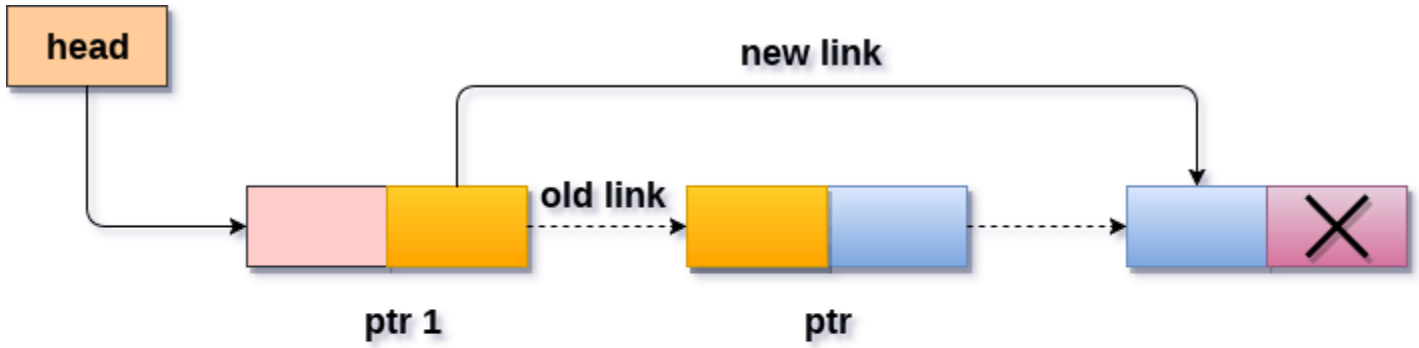
There are more than one node in the list and the last node of the list will be deleted.

- Algorithm
- **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!= NULL
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT
- [END OF LOOP]
- **Step 6:** SET PREPTR -> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT



Deletion in singly linked list after the specified node :

- In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted.
- We need to keep track of the two nodes.
- The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.



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

Algorithm

- **STEP 1:** IF HEAD = NULL
- WRITE UNDERFLOW
  GOTO STEP 10
  END OF IF
- **STEP 2:** SET TEMP = HEAD
- **STEP 3:** SET I = 0
- **STEP 4:** REPEAT STEP 5 TO 8 UNTIL I
- **STEP 5:** TEMP1 = TEMP
- **STEP 6:** TEMP = TEMP → NEXT
- **STEP 7:** IF TEMP = NULL
- WRITE "DESIRED NODE NOT PRESENT"
  GOTO STEP 12
  END OF IF
- **STEP 8:** I = I+1
- END OF LOOP
- **STEP 9:** TEMP1 → NEXT = TEMP → NEXT
- **STEP 10:** FREE TEMP
- **STEP 11:** EXIT

Traversing in singly linked list

- Traversing is the most common operation that is performed in almost every scenario of singly linked list.
- Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following statements.

```
ptr = head;
      while (ptr!=NULL)
         {
            ptr = ptr -> next;
         }
```

Algorithm

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

[END OF LOOP]

STEP 7: EXIT

Searching in singly linked list

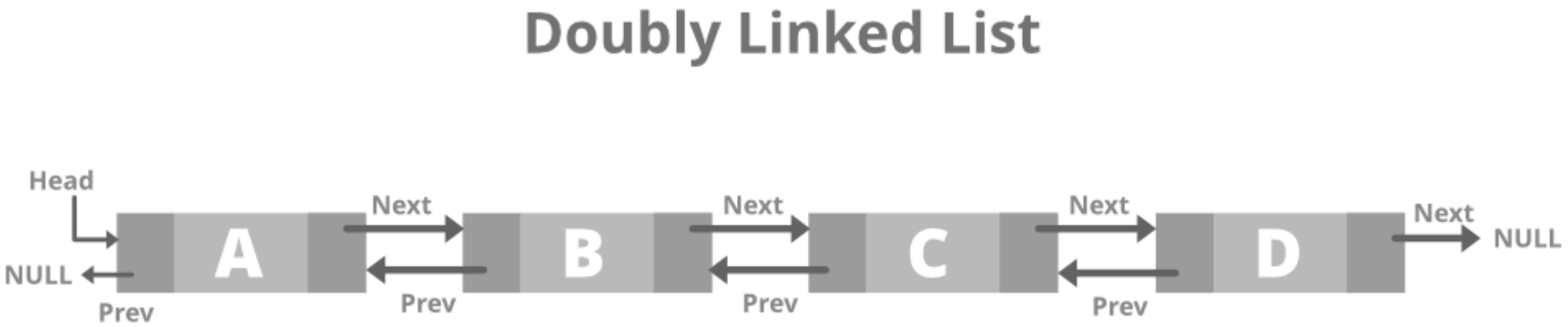
- Searching is performed in order to find the location of a particular element in the list.
- Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element.
- If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

- **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:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL
- **STEP 5:** if ptr → data = item
- write i+1
  End of IF
- **STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT
- [END OF LOOP]
- **STEP 8:** EXIT

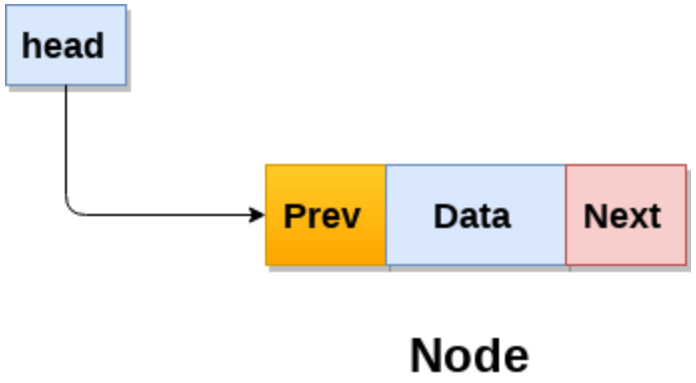
2. Doubly Linked List

- A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.
- Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



Doubly linked list

- It 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.



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

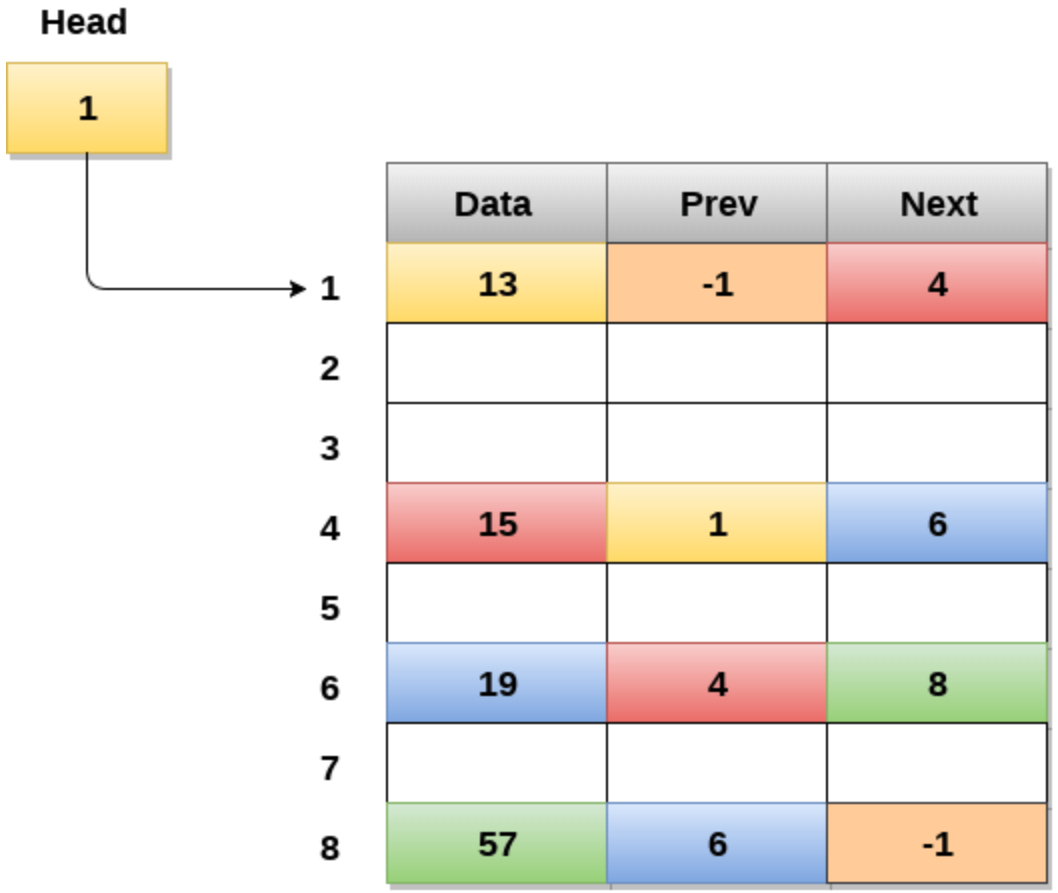
Doubly Linked List

- In C, structure of a node in doubly linked list can be given as :

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

Memory Representation of a doubly linked list

- In the following image, the first element of the list that is i.e. 13 stored at address 1.
- The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null.
- The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
- We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

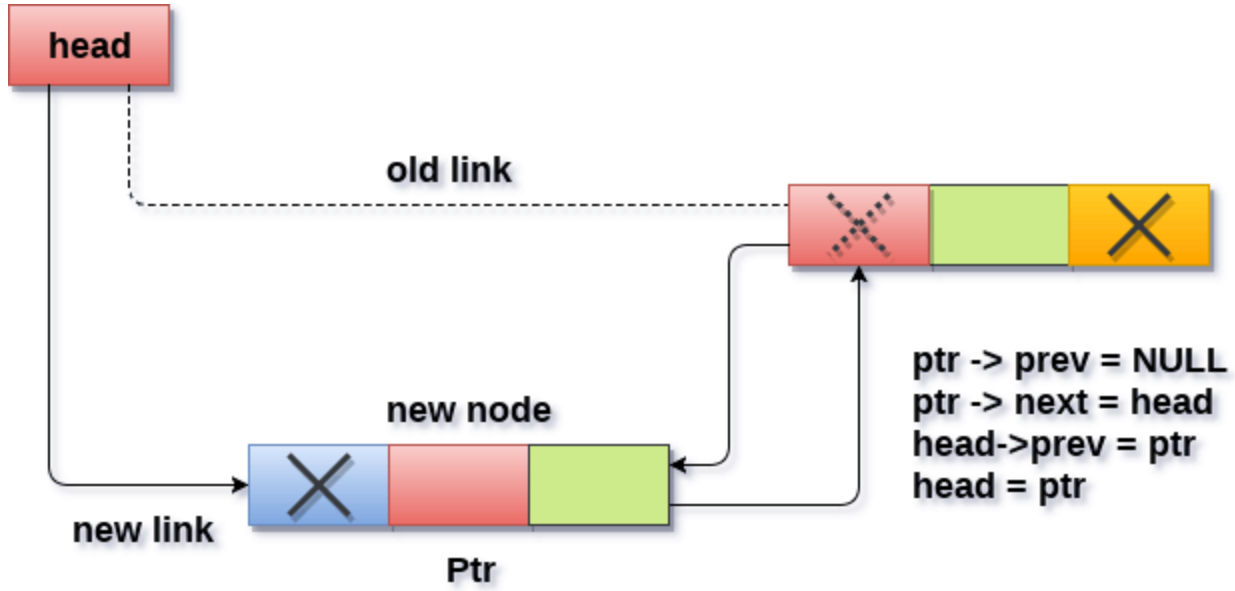
Operations on doubly linked list

- **Node Creation**

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

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

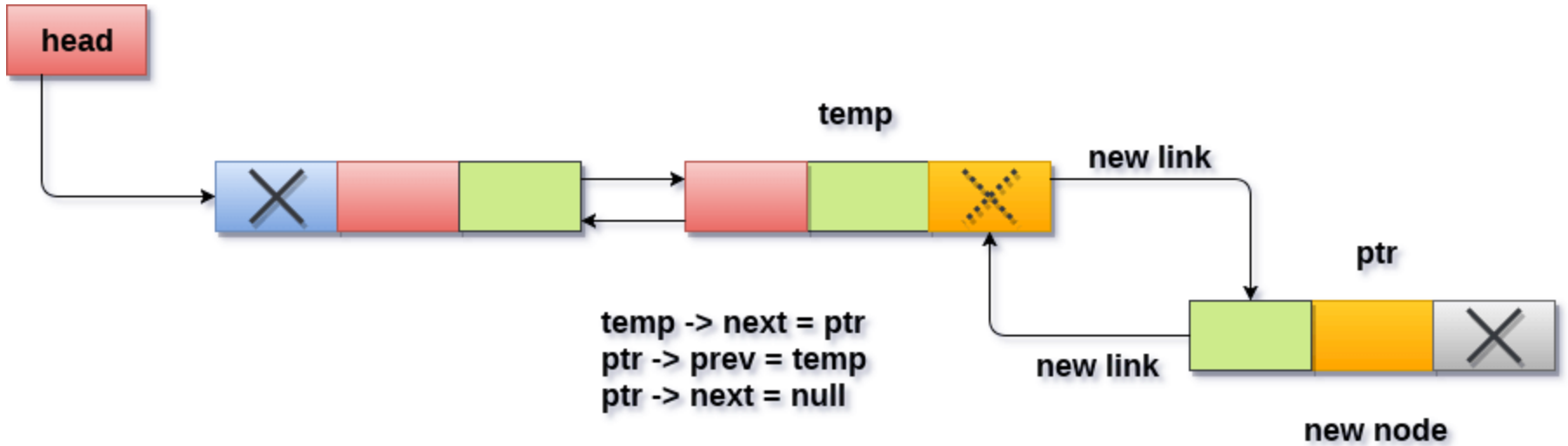
Insertion in doubly linked list at beginning



Insertion into doubly linked list at beginning

- **Step 1:** IF ptr = NULL
- Write OVERFLOW
  Go to Step 9
  [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 -> PREV = NULL
- **Step 6:** SET NEW_NODE -> NEXT = START
- **Step 7:** SET head -> PREV = NEW_NODE
- **Step 8:** SET head = NEW_NODE
- **Step 9:** EXIT

Insertion in doubly linked list at the end

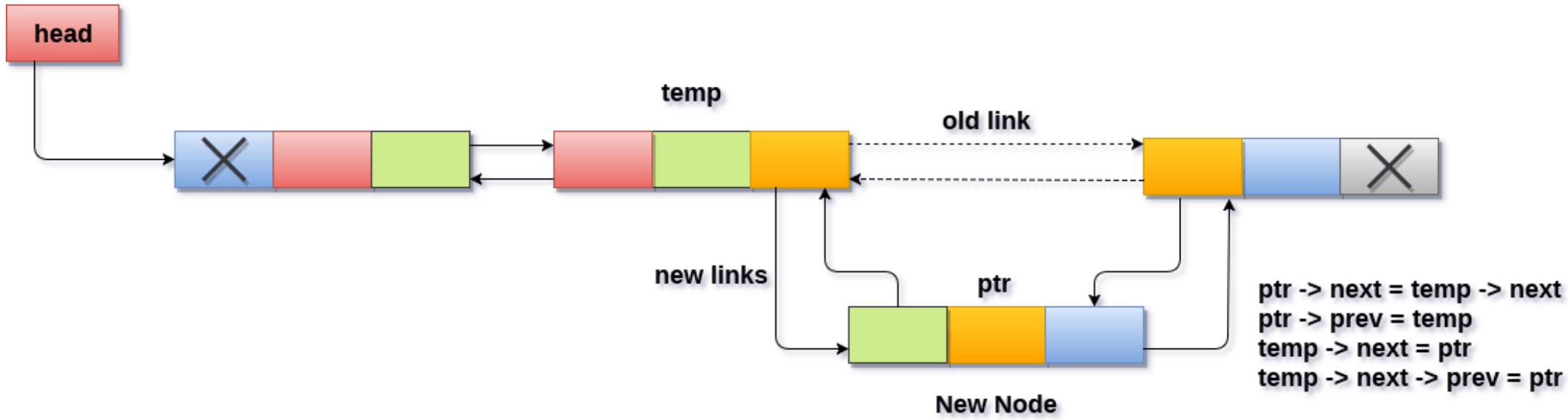


Insertion into doubly linked list at the end

Algorithm

- **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 NEW_NODE -> NEXT = NULL
- **Step 6:** SET TEMP = START
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != NULL
- **Step 8:** SET TEMP = TEMP -> NEXT
- [END OF LOOP]
- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10C:** SET NEW_NODE -> PREV = TEMP
- **Step 11:** EXIT

Insertion in doubly linked list after Specified node



Insertion into doubly linked list after specified node

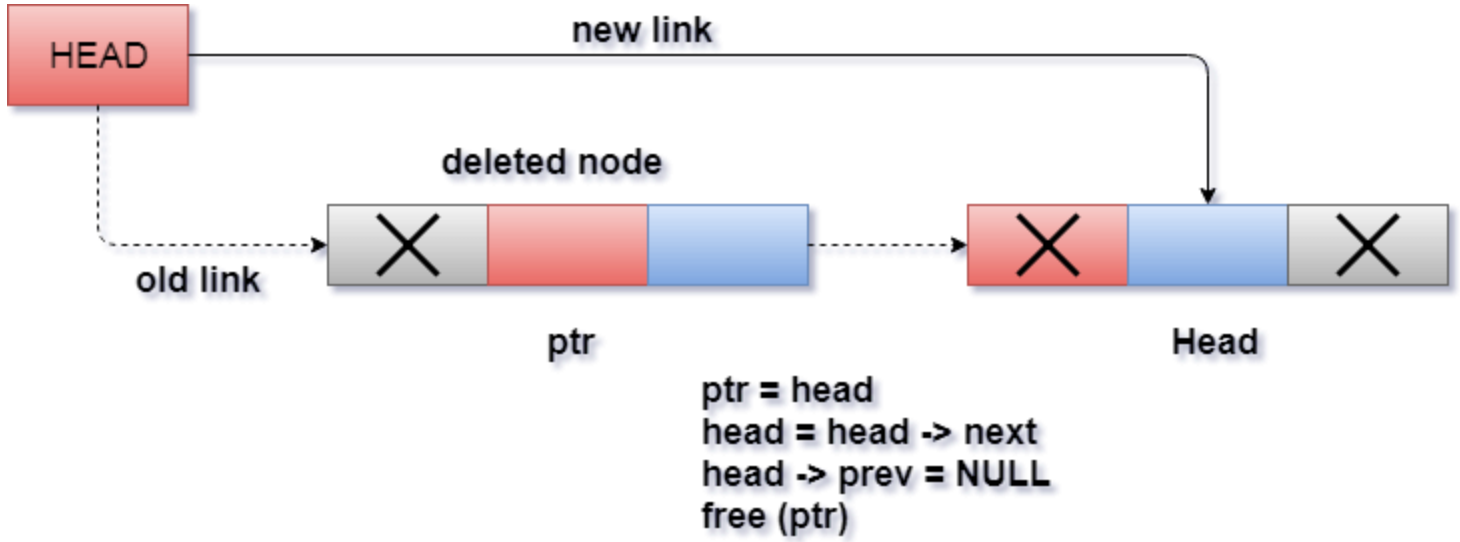
Algorithm

- **Step 1:** IF PTR = NULL
- Write OVERFLOW
  Go to Step 15
  [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 = START
- **Step 6:** SET I = 0
- **Step 7:** REPEAT 8 to 10 until I<="" li="">
- **Step 8:** SET TEMP = TEMP -> NEXT
- **STEP 9:** IF TEMP = NULL
- **STEP 10:** WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
- GOTO STEP 15
  [END OF IF]
  [END OF LOOP]
- **Step 11:** SET NEW_NODE -> NEXT = TEMP -> NEXT
- **Step 12:** SET NEW_NODE -> PREV = TEMP
- **Step 13 :** SET TEMP -> NEXT = NEW_NODE
- **Step 14:** SET TEMP -> NEXT -> PREV = NEW_NODE
- **Step 15:** EXIT

Deletion at beginning

- **STEP 1:** IF HEAD = NULL
  WRITE UNDERFLOW
  GOTO STEP 6

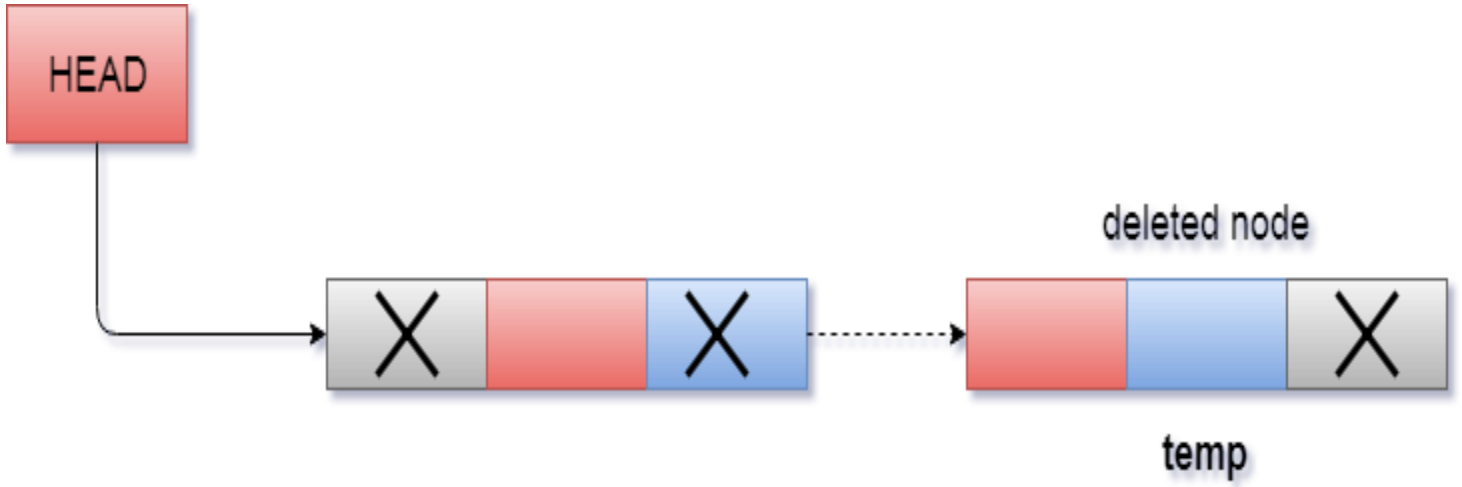
- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

Deletion in doubly linked list at the end

- **Step 1:** IF HEAD = NULL
- Write UNDERFLOW
  Go to Step 7
  [END OF IF]
- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT
- [END OF LOOP]
- **Step 5:** SET TEMP ->PREV-> NEXT = NULL
- **Step 6:** FREE TEMP
- **Step 7:** EXIT

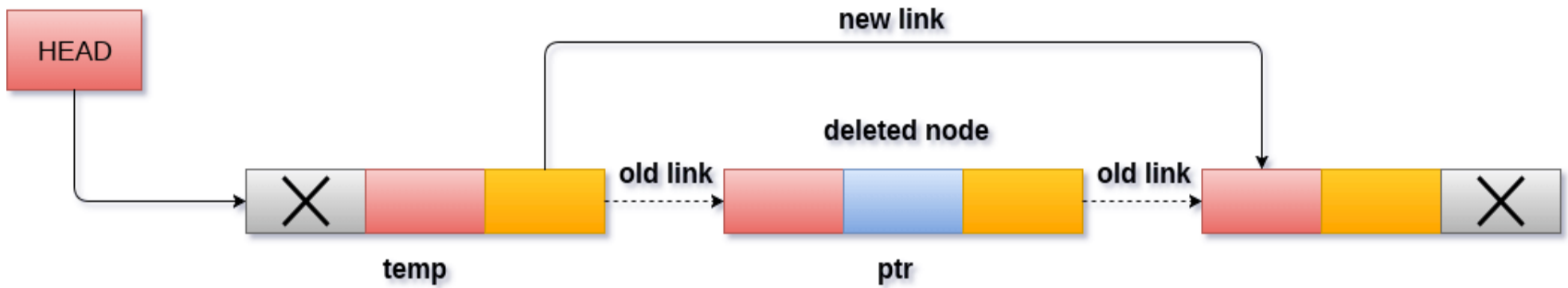


Deletion in doubly linked list at the end

Deletion in doubly linked list after the specified node

- **Step 1:** IF HEAD = NULL
  Write UNDERFLOW
  Go to Step 9
  [END OF IF]
- **Step 2:** SET TEMP = HEAD
- **Step 3:** Repeat Step 4 while TEMP -> DATA != ITEM
- **Step 4:** SET TEMP = TEMP -> NEXT
  [END OF LOOP]

- **Step 5:** SET PTR = TEMP -> NEXT
- **Step 6:** SET TEMP -> NEXT = PTR -> NEXT
- **Step 7:** SET PTR -> NEXT -> PREV = TEMP
- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

Traversing in doubly linked list

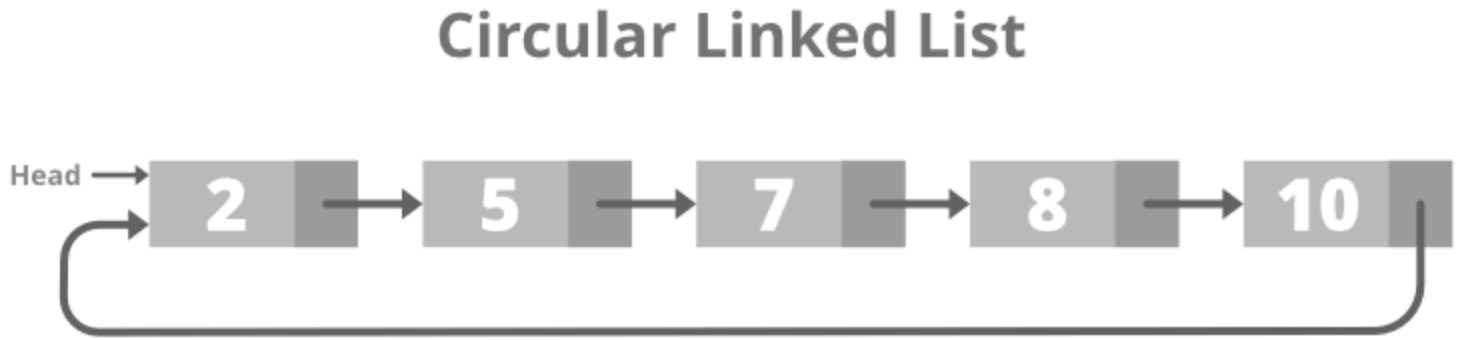
- **Step 1:** IF HEAD == NULL
- WRITE "UNDERFLOW"
  GOTO STEP 6
  [END OF IF]
- **Step 2:** Set PTR = HEAD
- **Step 3:** Repeat step 4 and 5 while PTR != NULL
- **Step 4:** Write PTR → data
- **Step 5:** PTR = PTR → next
- **Step 6:** Exit

Searching for a specific node in Doubly Linked List

- **Step 1:** IF HEAD == NULL
- WRITE "UNDERFLOW"
  GOTO STEP 8
  [END OF IF]
- **Step 2:** Set PTR = HEAD
- **Step 3:** Set i = 0
- **Step 4:** Repeat step 5 to 7 while PTR != NULL
- **Step 5:** IF PTR → data = item
- return i
  [END OF IF]
- **Step 6:** i = i + 1
- **Step 7:** PTR = PTR → next
- **Step 8:** Exit

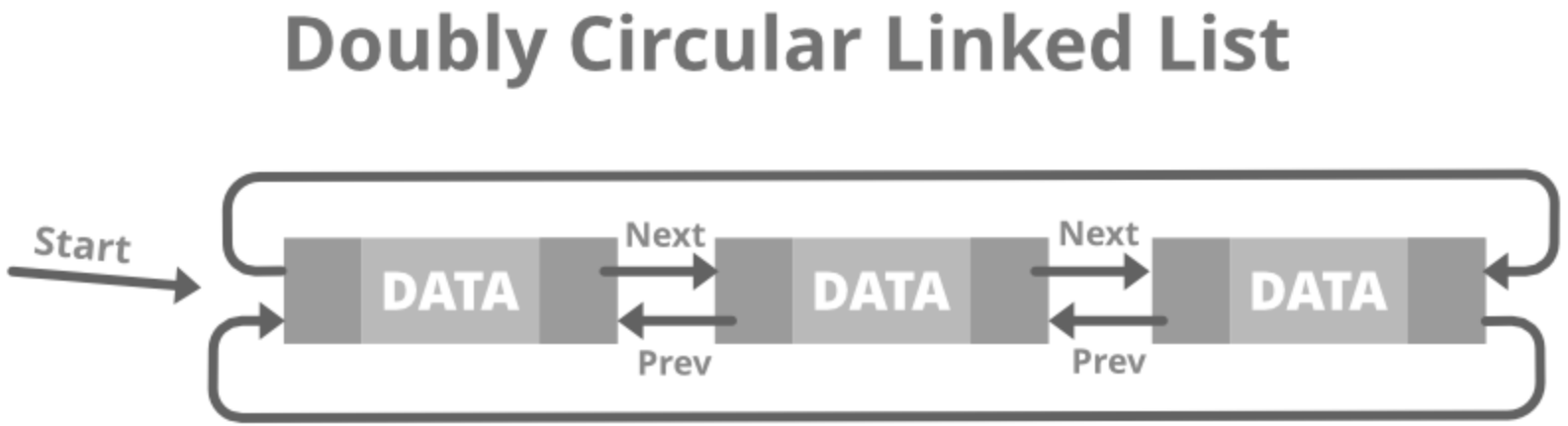
3. Circular Linked List

- A circular linked list is that in which the last node contains the pointer to the first node of the list.
- While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started.
- Thus, a circular linked list has no beginning and no end. Below is the image for the same:



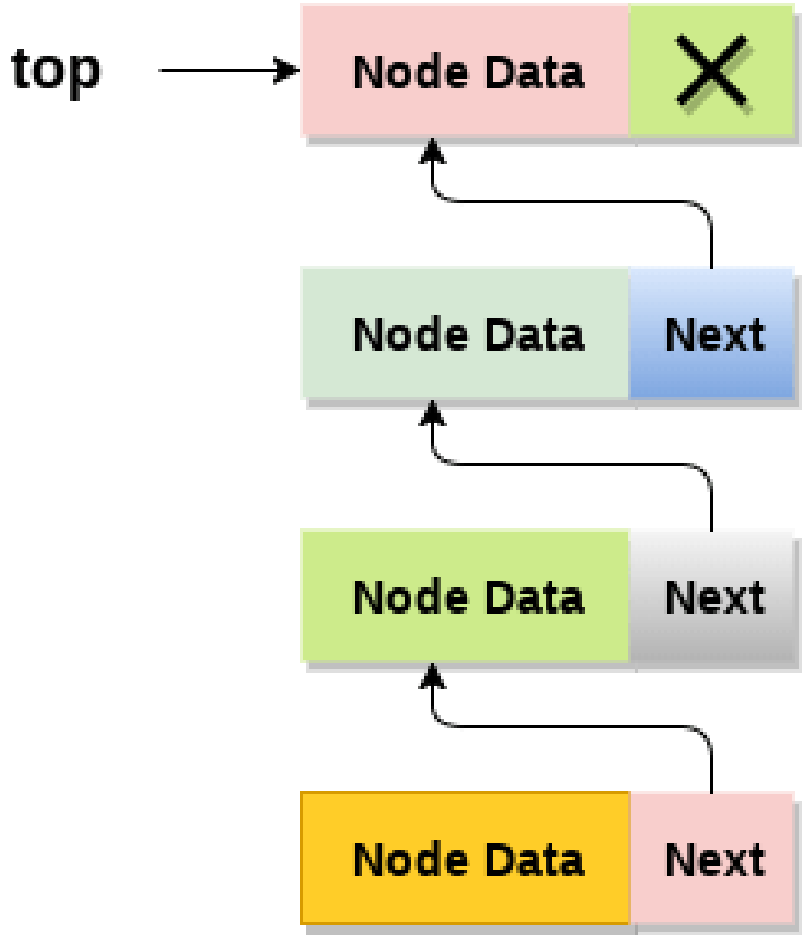
4. Doubly Circular linked list

- A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence.
- The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list.
- The circular doubly linked list does not contain null in the previous field of the first node. Below is the image for the same:



Linked list implementation of stack

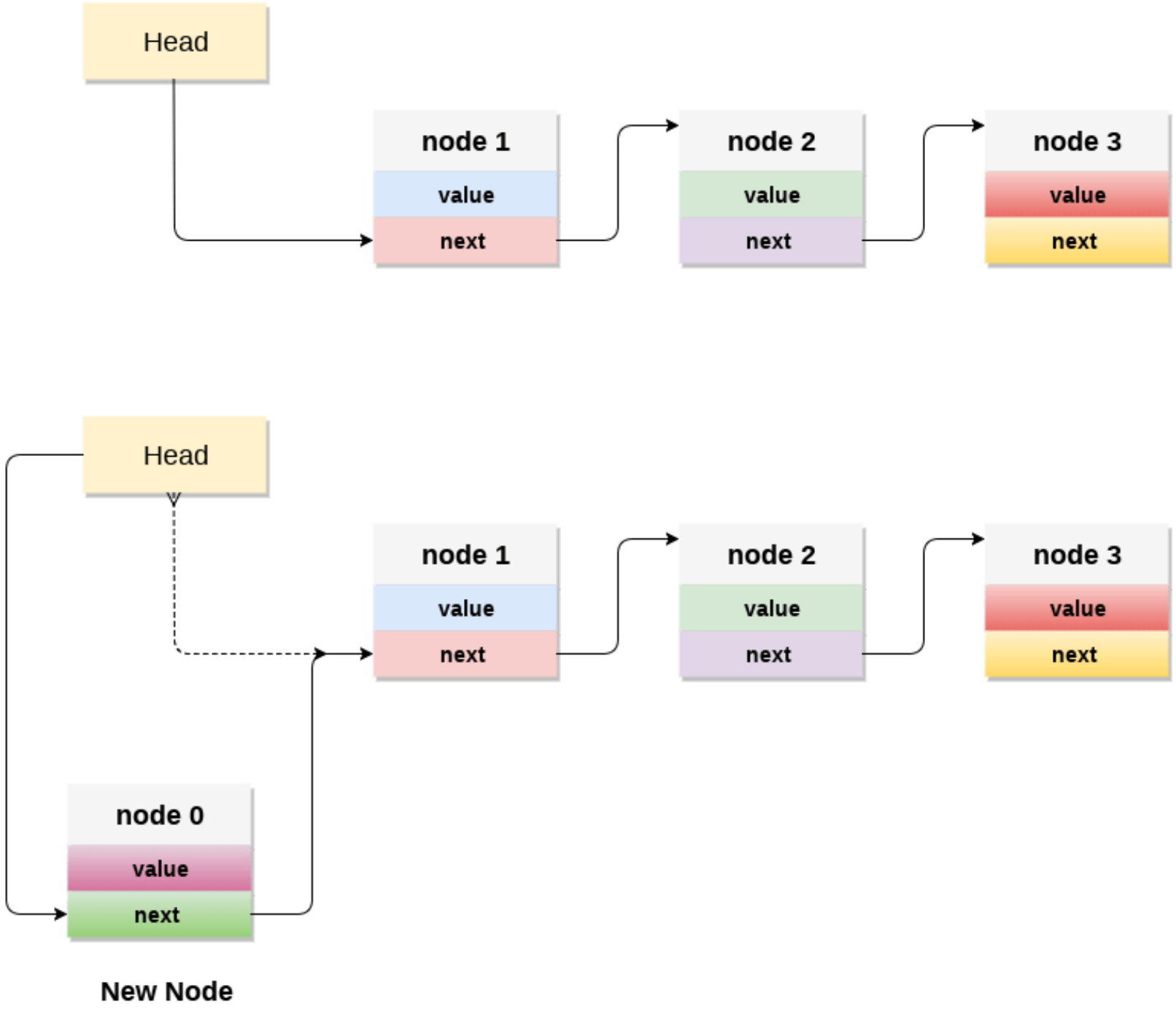
- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- Each node contains a pointer to its immediate successor node in the stack.
- Stack is said to be overflown if the space left in the memory heap is not enough to create a node.
- The top most node in the stack always contains null in its address field.



Stack

Adding a node to the stack (Push operation)

- Adding a node to the stack is referred to as **push** operation.
- Pushing an element to a stack in linked list implementation is different from that of an array implementation.
- In order to push an element onto the stack, the following steps are involved.
  1. Create a node first and allocate memory to it.
  2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
  3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.


Head
node 1
value
next
node 2
value
next
node 3
value
next
Head
node 1
value
next
node 2
value
next
node 3
value
next
node 0
value
next
New Node

```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)
      {
         ptr->val = val;
         ptr -> next = NULL;
         head=ptr;
      }
      else
      {
         ptr->val = val;
         ptr->next = head;
         head=ptr;
       }
      printf("Item pushed");
      }
}
```

Deleting a node from the stack (POP operation)

- Deleting a node from the top of stack is referred to as **pop** operation.
- Deleting a node from the linked list implementation of stack is different from that in the array implementation.
- In order to pop an element from the stack, we need to follow the following steps :
  1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
  2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
    else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

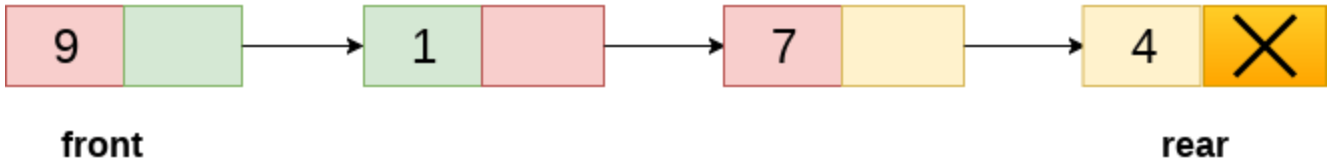
Display the nodes (Traversing)

- Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.
  - Copy the head pointer into a temporary pointer.
  - Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

```
void display()
{
   int i;
   struct node *ptr;
   ptr=head;
   if(ptr == NULL)
   {
      printf("Stack is empty\n");
   }
   else
   {
      printf("Printing Stack elements \n");
      while(ptr!=NULL)
      {
         printf("%d\n",ptr->val);
         ptr = ptr->next;
      }
   } }
```

Linked List implementation of Queue

- In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.
- In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
- Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.
- The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

- **The two main operations performed on the linked queue are:**
  - Insertion
  - Deletion

Insertion

- Insert operation or insertion on a linked queue adds an element to the end of the queue. The new element which is added becomes the last element of the queue.
- **Algorithm to perform Insertion on a linked queue:**
  - Create a new node pointer.
    ptr = (struct node *) malloc (sizeof(struct node));
  - Now, two conditions arise, i.e., either the queue is empty, or the queue contains at least one element.
  - If the queue is empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.
  - If the queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point the rear pointer to the newly created node ptr

- Create a new node pointer.
  ptr = (struct node *) malloc (sizeof(struct node));
- Now, two conditions arise, i.e., either the queue is empty, or the queue contains at least one element.
- If the queue is empty, then the new node added will be both front and rear, and the next pointer of front and rear will point to NULL.

```
*ptr - > data = val;

if (front == NULL) {
    front = ptr;
    rear = ptr;
    front - > next = NULL;
    rear - > next = NULL;
}
```

- If the queue contains at least one element, then the condition front == NULL becomes false. So, make the next pointer of rear point to new node ptr and point the rear pointer to the newly created node ptr

```
rear -> next = ptr;
rear = ptr;
```