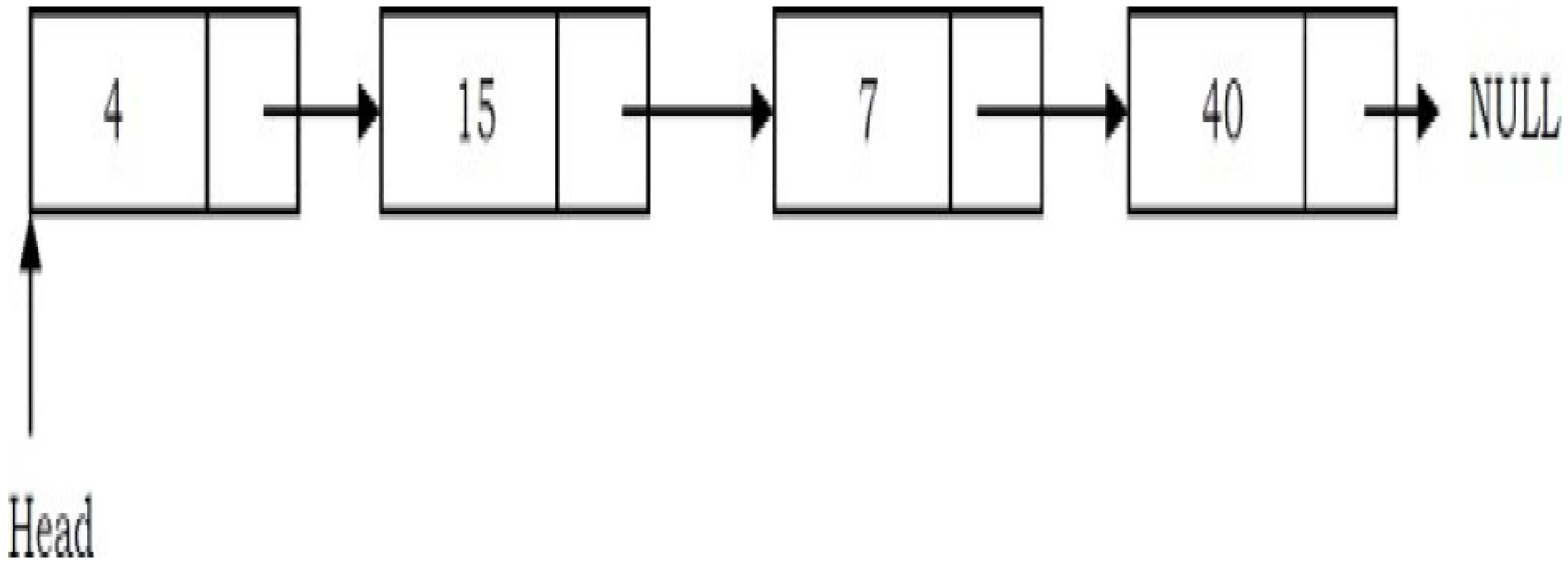
Linked Lists

What is a Linked List?

- A linked list is a data structure used for storing collections of data.
- A linked list has the following properties.
  - Successive **elements** are **connected by pointers**
  - The **last** element points to **NULL**
  - Can **grow or shrink in size** during execution of a program
  - Can be made just as long as required (**until systems memory exhausts**)
  - Does not **waste memory space** (but takes some extra memory for pointers). It allocates memory as list grows.

Example of Single Linked List



Linked Lists ADT

- The following operations make linked lists an ADT:
- **Main Linked Lists Operations**
  - **Insert**: **inserts** an element **into** the list
  - **Delete**: **removes** and **returns** the specified **position** element from the list

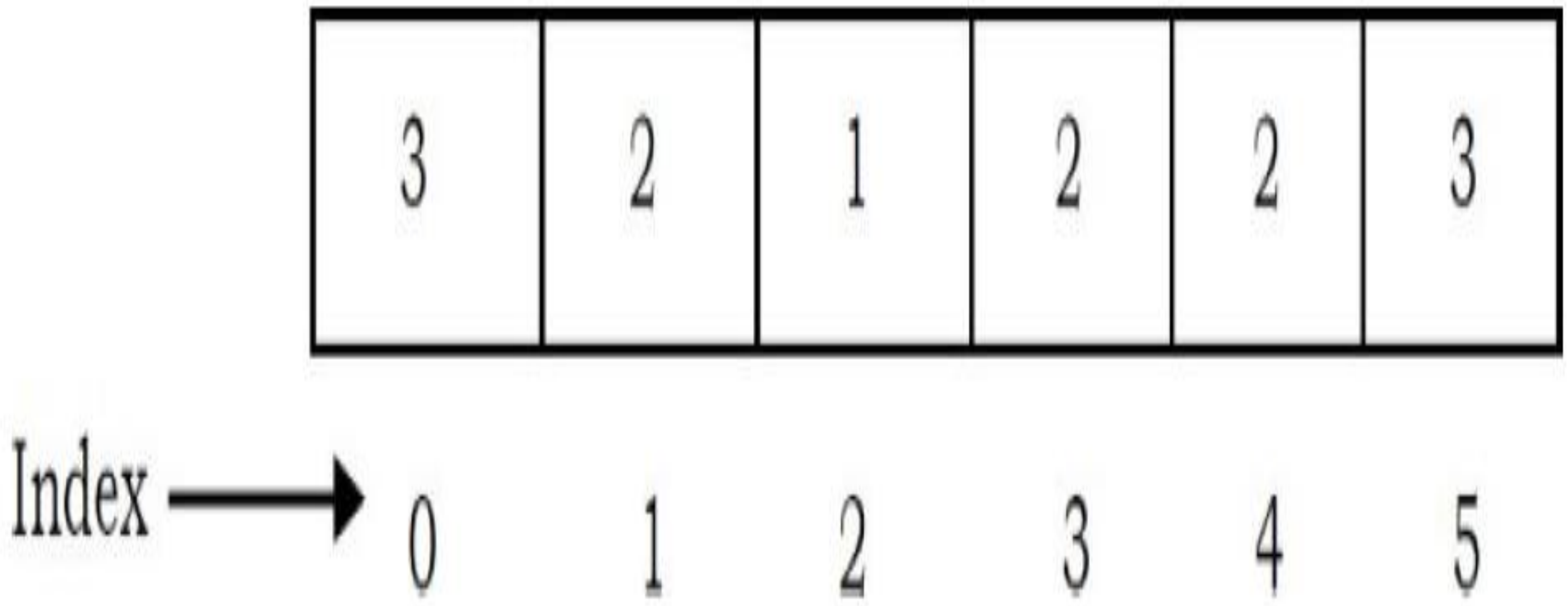
- **Auxiliary Linked Lists Operations**
  - **Delete List**: **removes all elements** of the list (disposes the list)
  - **Count**: returns the **number of elements** in the list
  - Find ***nth*** **node** from the end of the list

Why Linked Lists?

- Both linked lists and arrays are used to store collections of data, and since both are used for the same purpose, we need to differentiate their usage.
- That means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

Arrays Overview

- One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.



3
2
1
2
2
3
Index
0
1
2
3
4
5

Why Constant Time for Accessing Array Elements?

- To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.
- This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

First Disadvantage of Arrays

- Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.
  - **Fixed size:** The **size** of the array is **static** (specify the array size before using it).

Second Disadvantage of Arrays

- – **One block allocation:** To **allocate** the **array** itself at the **beginning**, sometimes it may not be possible to get the memory for the complete array (**if the array size is big**).

- Eg: A[10000]

Third Disadvantage of Arrays

- **Complex position-based insertion:** To **insert an element** at a **given position**, we may **need to shift the existing elements**.
- This will **create a position for us to insert** the **new element at the desired position**.
- If the position at which we **want to add an element is at the beginning**, then the shifting operation is **more expensive**.
- A[20]

Dynamic Arrays

- Dynamic array (also called as ***growable array, resizable array, dynamic table*, or *array list***) is a random access, variable-size list data structure that allows elements to be added or removed.
- One simple way of implementing dynamic arrays is to initially start with some fixed size array.
- As soon as that array becomes full, create the new array double the size of the original array.
- Similarly, reduce the array size to half if the elements in the array are less than half.

Advantages of Linked Lists

- Linked lists have both advantages and disadvantages.
- **Issues in arrays:**
  - To create an array, we must allocate memory for a certain number of elements.
  - To add more elements to the array when full, we must create a new array and copy the old array into the new array.
  - This can take a lot of time.
  - We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory.

Advantages of Linked Lists

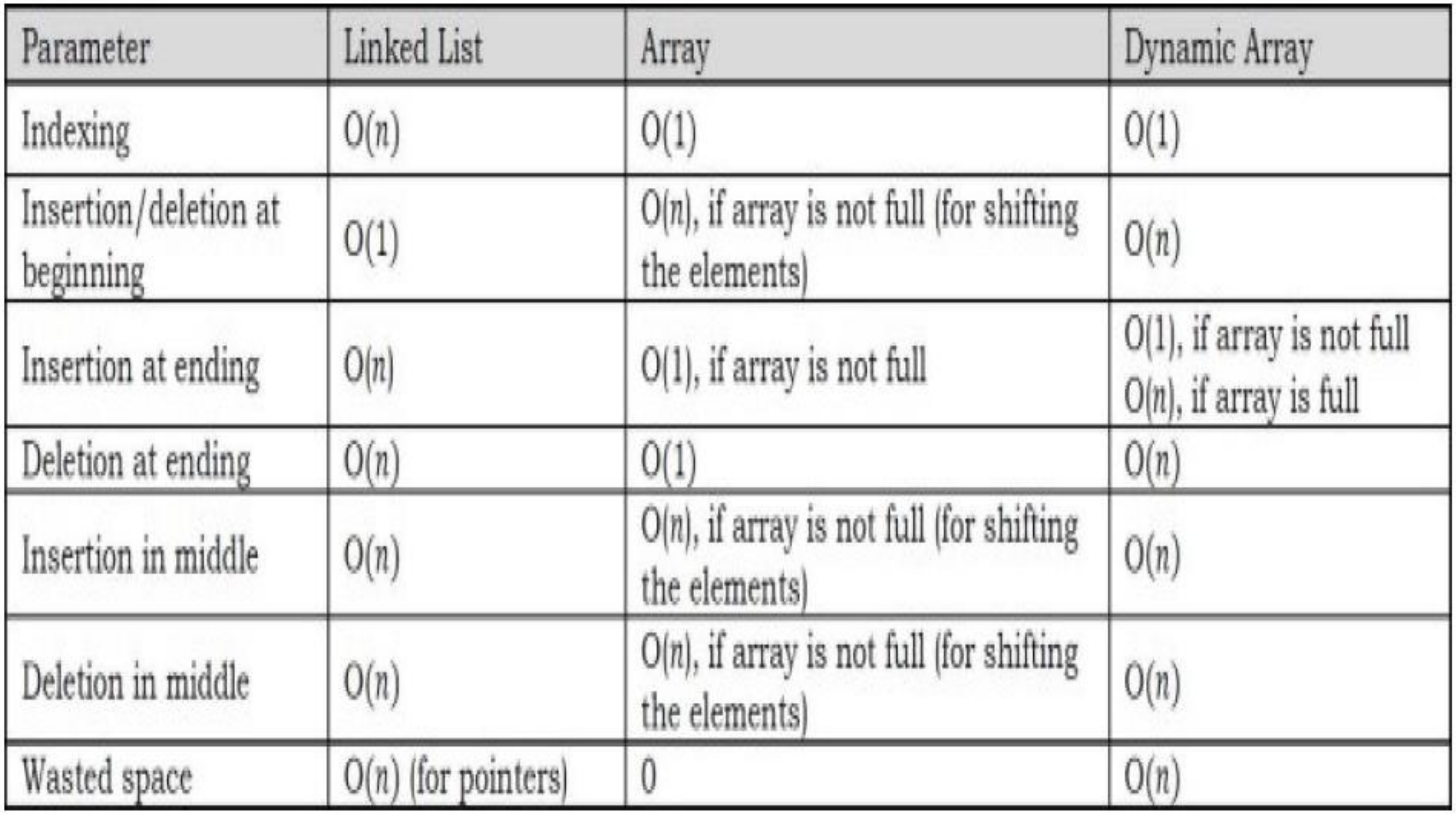
- **The advantage of linked lists is that they can be *expanded* in constant time.**
  - With a linked list, we can **start with space for just one allocated element** and ***add* on new elements** easily without the need to do any copying and reallocating.

Issues with Linked Lists (Disadvantages)

- There are a number of issues with linked lists.
  - The main disadvantage of linked lists is ***access time*** **to individual elements**.
    - Array is random-access, which means it takes O(1) to access any element in the array.
    - **Linked lists take O(*n*) for access to an element in the list in the worst case**.
  - Another advantage of arrays in **access time is *spacial locality* in memory**. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

- Although the dynamic allocation of storage is a great advantage, the ***overhead*** **with storing and retrieving data** can make a big difference. Sometimes linked lists are *hard to manipulate*.
  - ➢ If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference.
  - ➢ This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference.
- Finally, linked lists **waste memory in terms of extra reference points.**

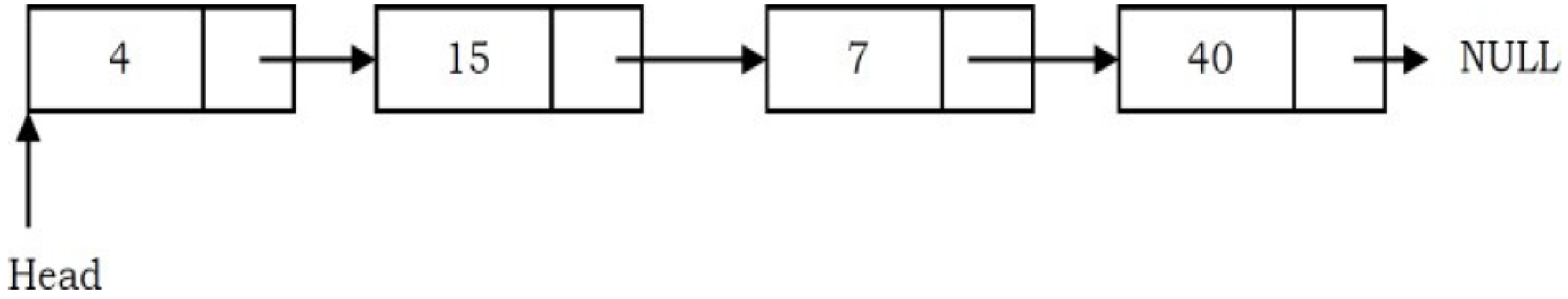
Comparison of Linked Lists with Arrays & Dynamic Arrays



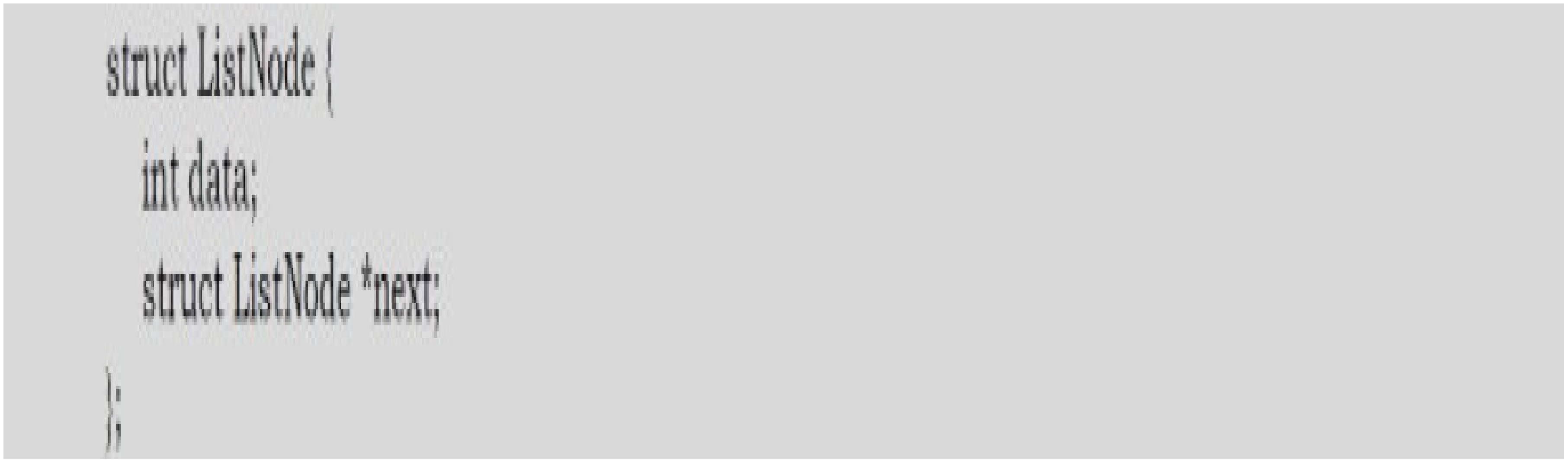
Parameter	Linked List	Array	Dynamic Array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$ (for pointers)	0	$O(n)$

Singly Linked Lists

- Generally "linked list" means a singly linked list.
- This **list consists of a number of nodes**
- **Each node has a *next* pointer to the following element**.
- The link of the **last node in the list is NULL**, which indicates the **end of the list**.



- Following is a type declaration for a linked list of integers:



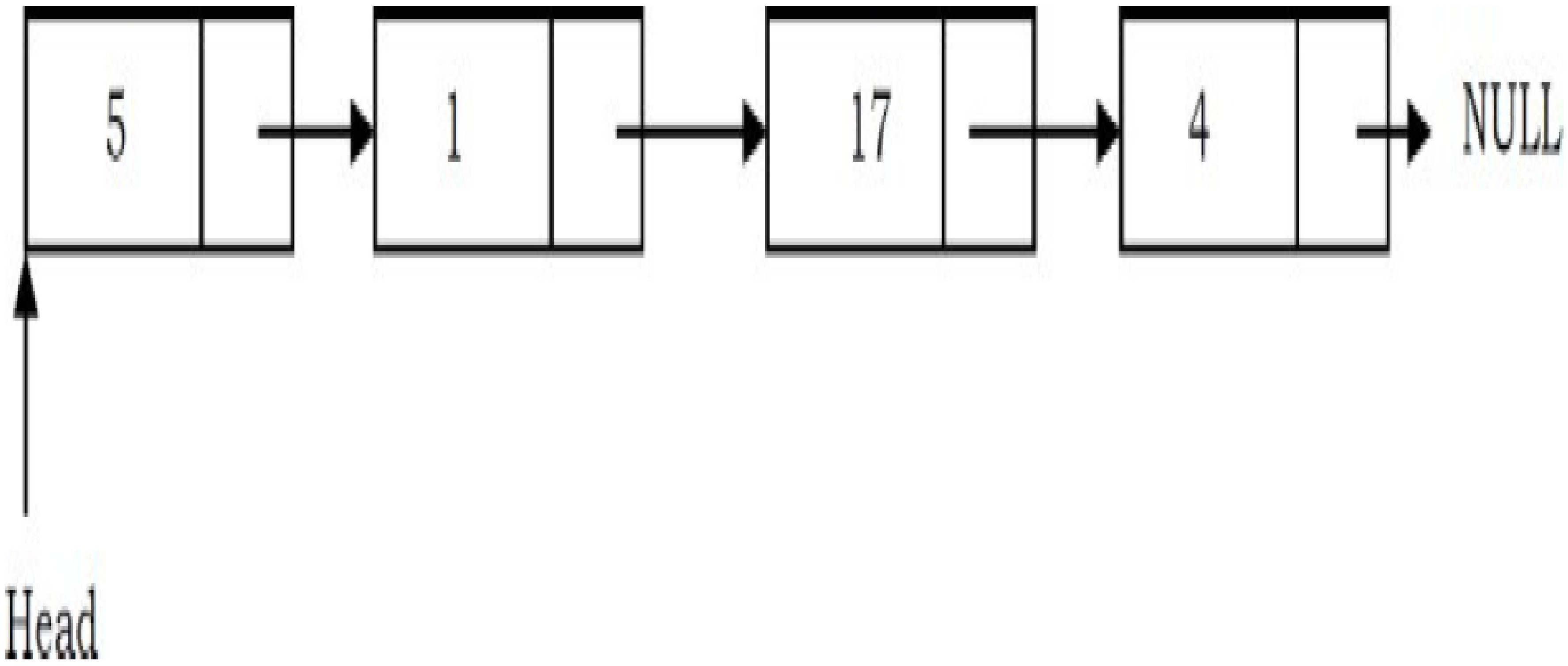
```
struct ListNode {
   int data;
   struct ListNode *next;
};
```

Basic Operations on a List

- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

Traversing the Linked List

- Let us assume that the *head* points to the first node of the list.
- To traverse the list we do the following
  - ❖ Follow the pointers.
  - ❖ Display the contents of the nodes (or count) as they are traversed.
  - ❖ Stop when the next pointer points to NULL.


5
1
17
4
NULL
Head

- The ListLength() function takes a linked list as input and counts the number of nodes in the list.
- The function given below can be used for printing the list data with extra print function.

```
int ListLength(struct ListNode *head) {
   struct ListNode *current = head;
   int count = 0;

   while (current != NULL) {
      count++;
      current = current→next;
   }
   return count;
}
```

- Time Complexity: O(*n*), for scanning the list of size *n*.
- Space Complexity: O(1), for creating a temporary variable.

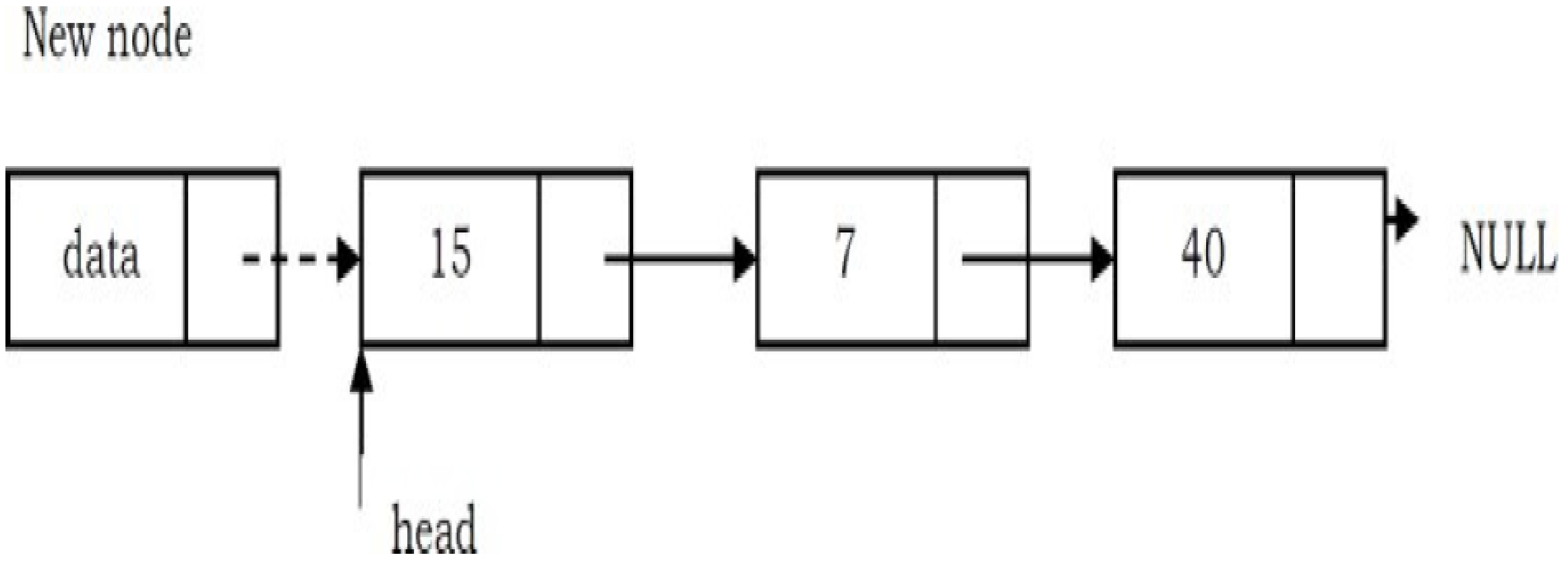
Singly Linked List Insertion

- Insertion into a singly-linked list has three cases:
- Inserting a new node before the head (<u>**at the beginning**</u>)
- Inserting a new node after the tail (at the <u>**end of the list**</u>)
- Inserting a new node at the <u>**middle of the list**</u> (random location)
- **Note:** To insert an element in the linked list at some position *p*, assume that after inserting the element the position of this new node is *p*.

Inserting a Node in Singly Linked List at the Beginning

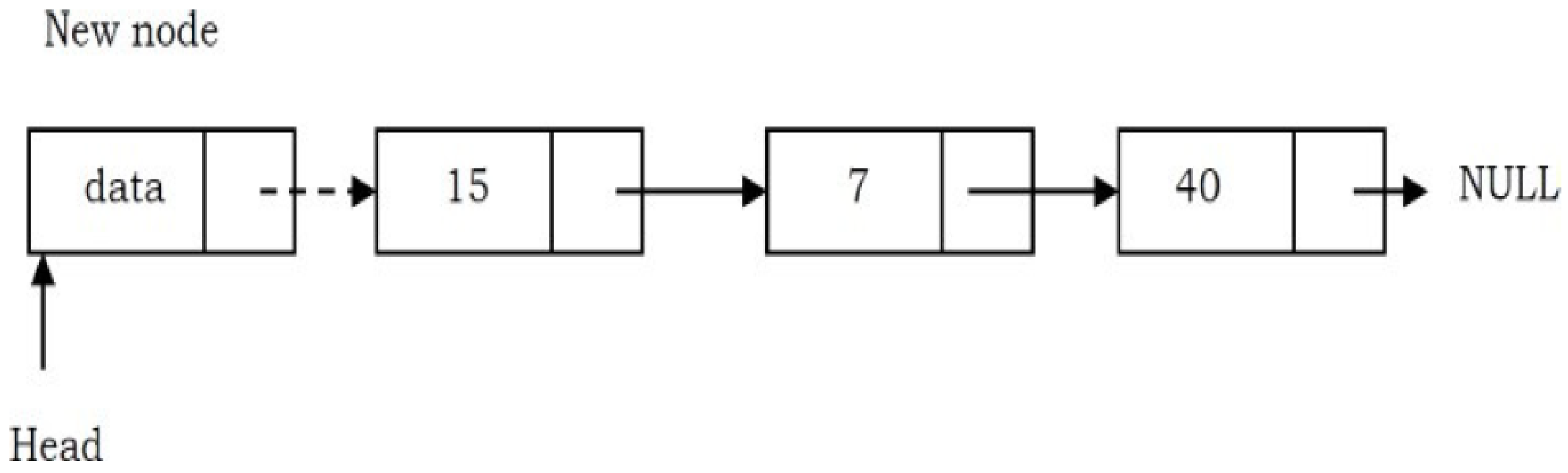
- In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (*n*ew node’s next pointer) and it can be done in two steps:
  - Update the next pointer of new node, to point to the current head.

Inserting a Node in Singly Linked List at the Beginning(Diagram)



- Update head pointer to point to the new node.

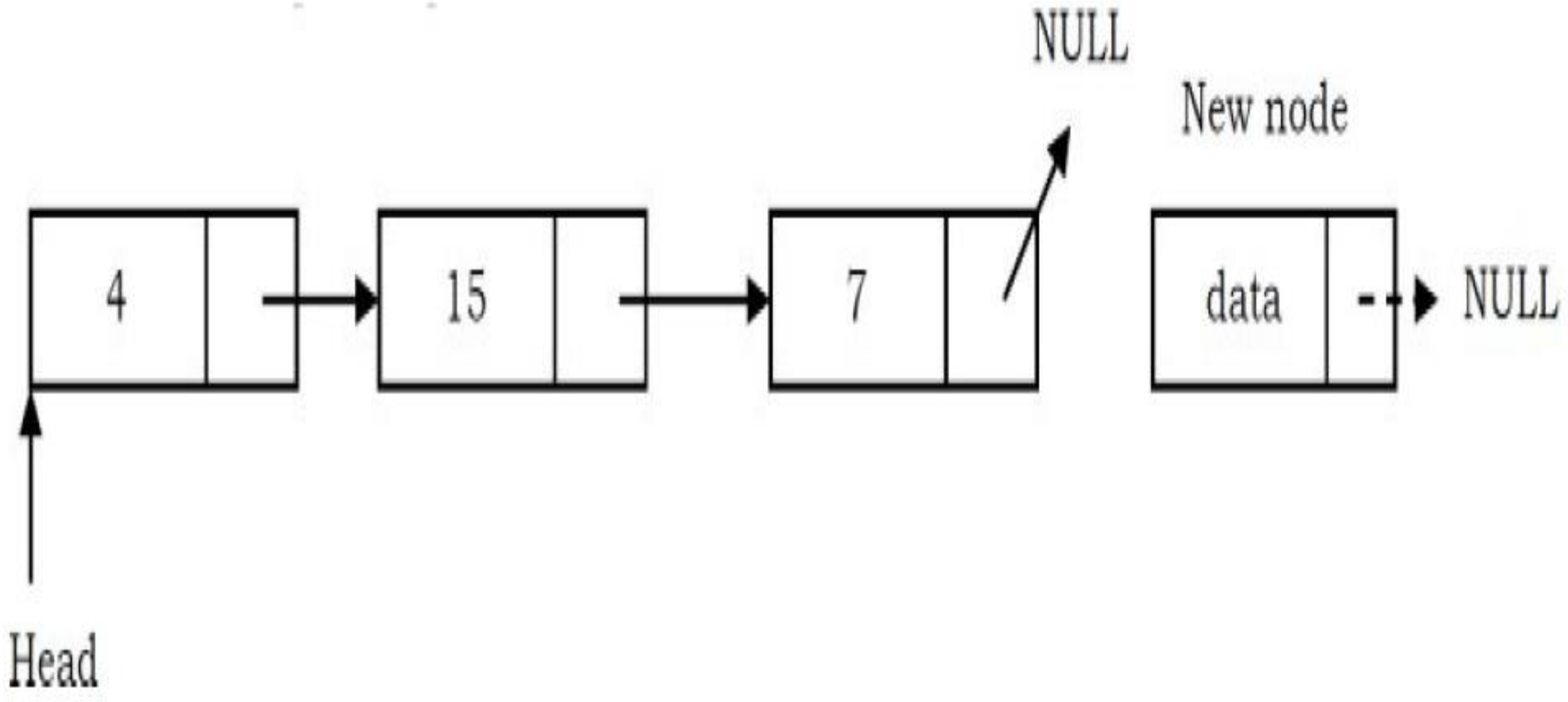
Inserting a Node in Singly Linked List at the Beginning(Diagram)



Inserting a Node in Singly Linked List at the Ending

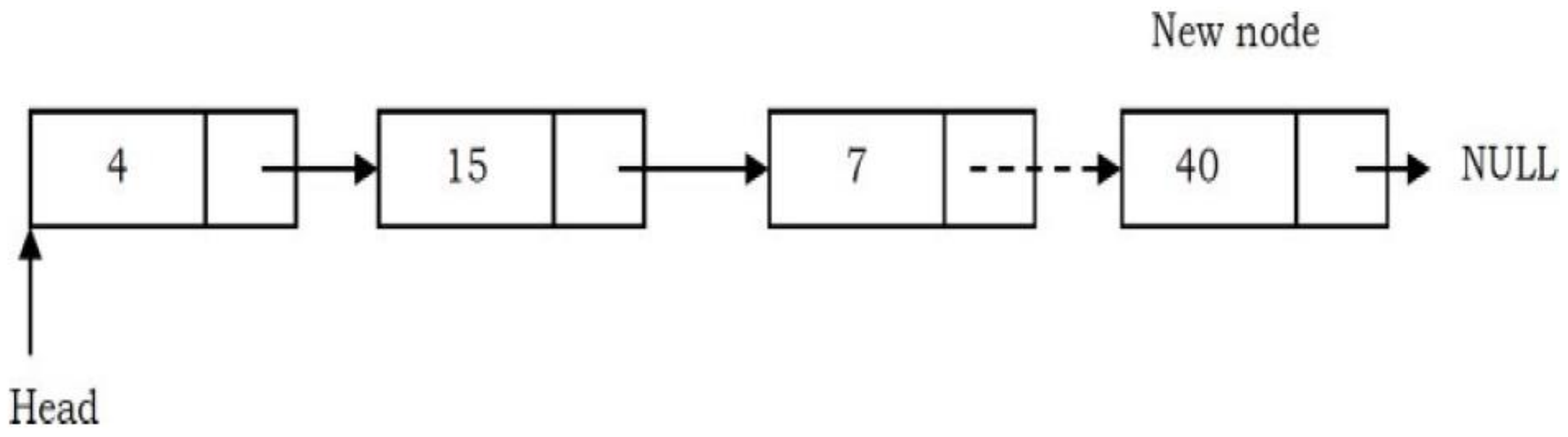
- In this case, we need to modify *two next pointers*
  - Last nodes next pointer and
  - new nodes next pointer.
    - New node’s next pointer points to **NULL**.

Inserting a Node in Singly Linked List at the Ending (Diagram)



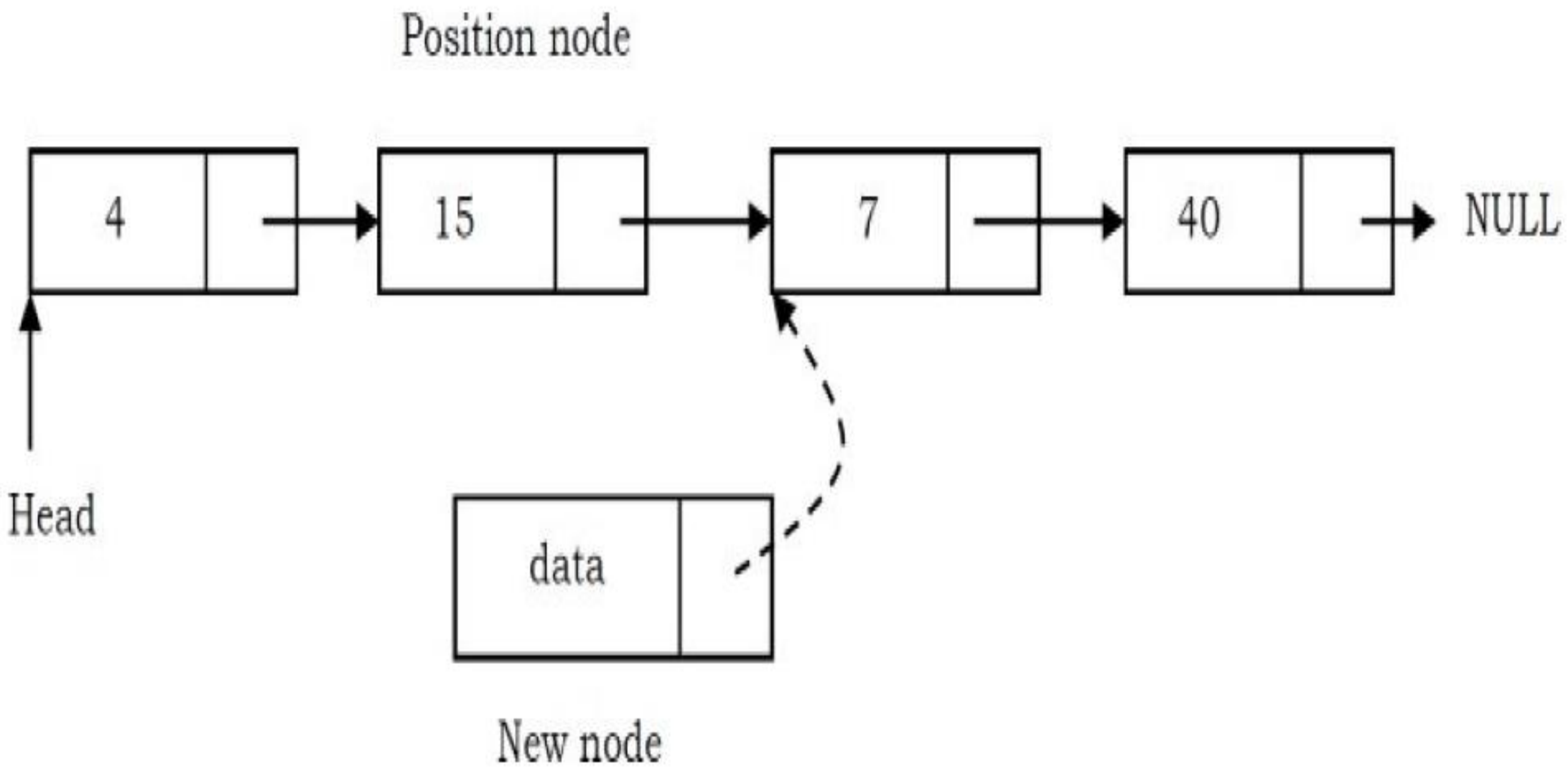
- Last nodes next pointer points to the new node.

Inserting a Node in Singly Linked List at the Beginning(Diagram)

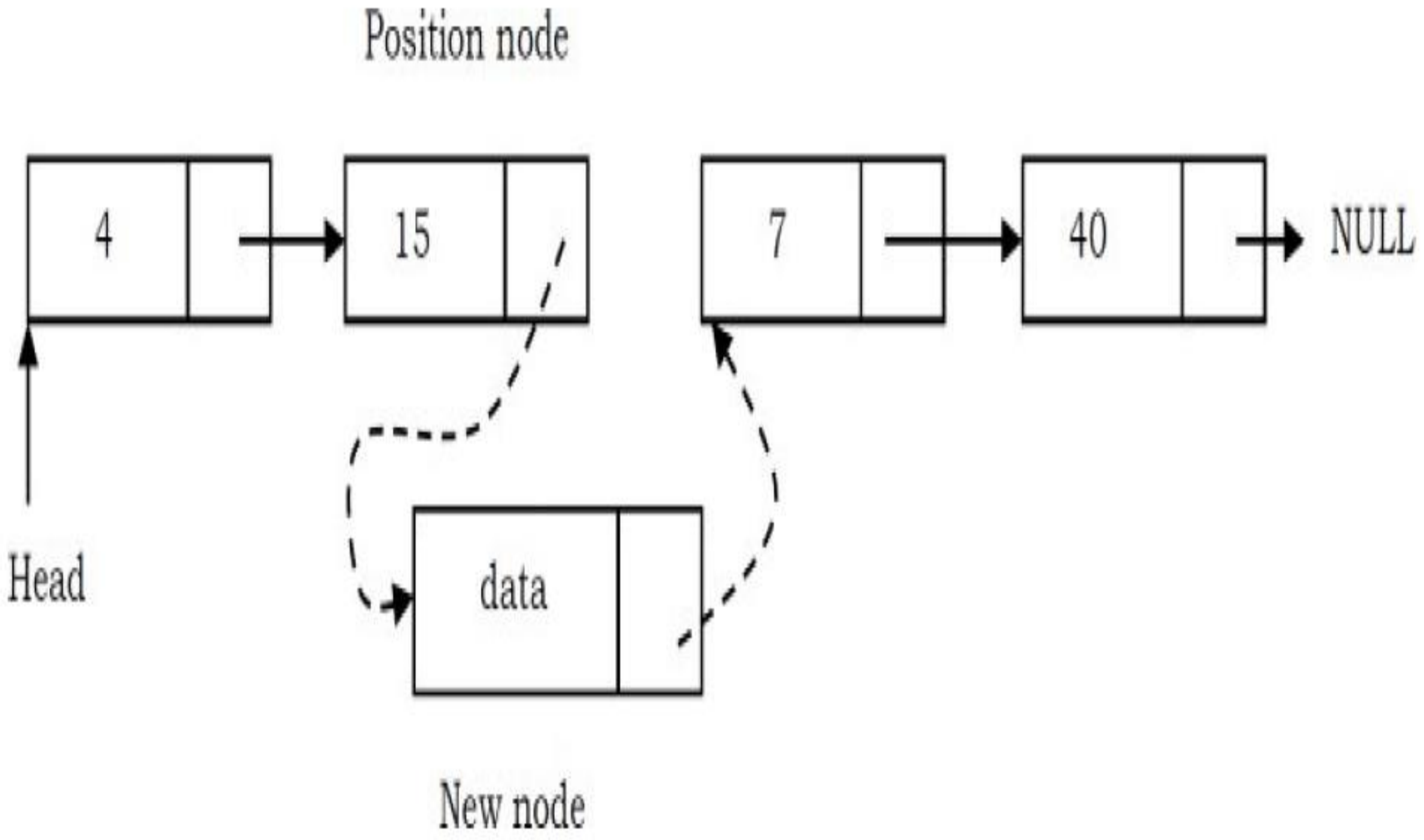


Inserting a Node in Singly Linked List at the Middle

- Let us assume that we are given a position where we want to insert the new node.
- In this case also, we need to modify two next pointers.
- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node.
- For simplicity let us assume that the second node is called *position* node.
- The new node points to the next node of the position where we want to add this node.

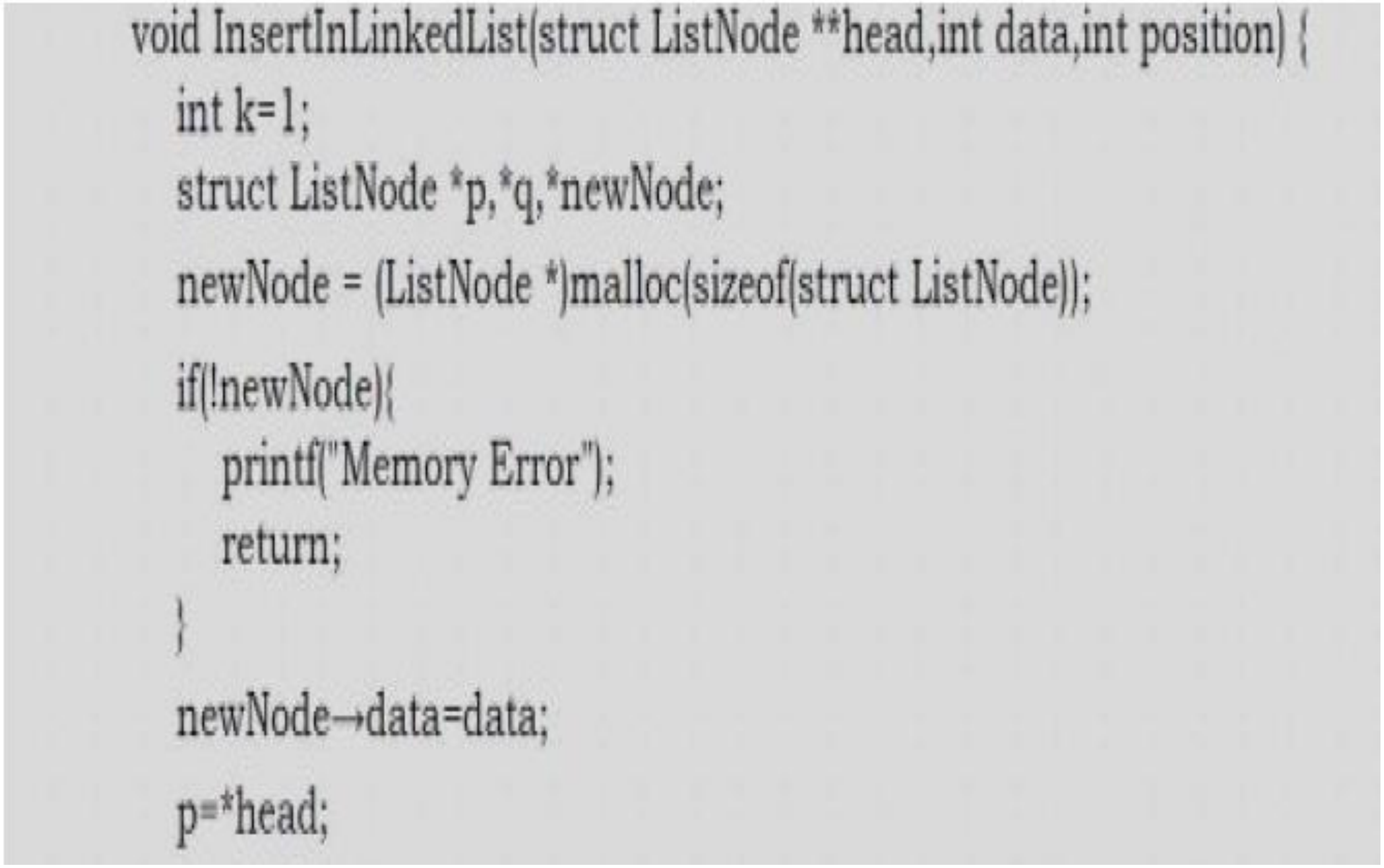


- Position node's next pointer now points to the new node.


Position node
4
15
7
40
NULL
Head
data
New node

- Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function.
- For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

Code for Inserting a Node



```
void InsertInLinkedList(struct ListNode **head,int data,int position) {
  int k=1;
  struct ListNode *p,*q,*newNode;
  newNode = (ListNode *)malloc(sizeof(struct ListNode));
  if(!newNode){
    printf("Memory Error");
    return;
  }
  newNode→data=data;
  p=*head;
```

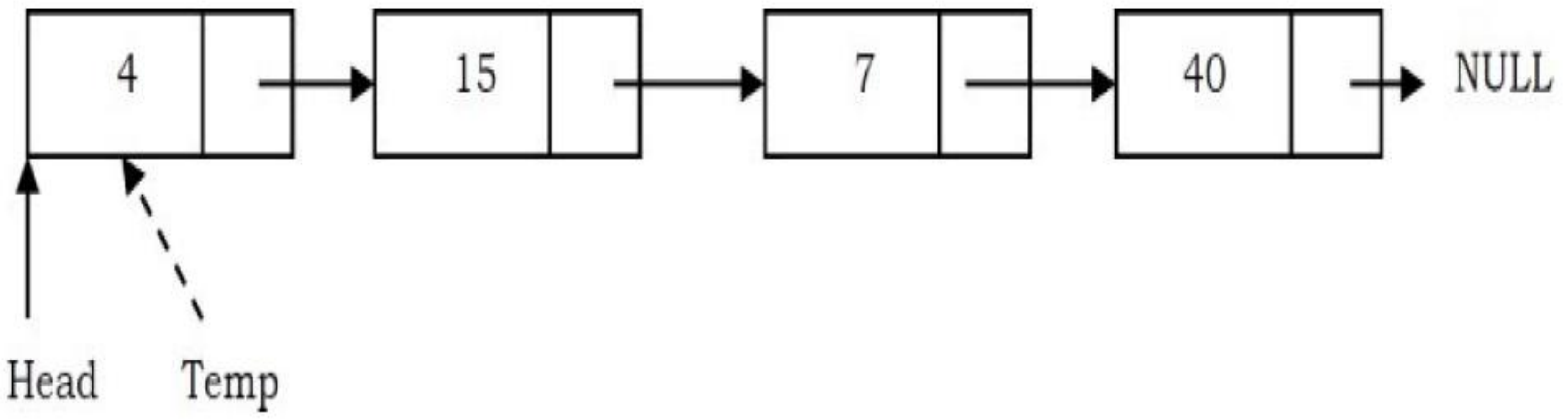
```
    //Inserting at the beginning
    if(position == 1){
        newNode→next=p;
        *head=newNode;
    }
    else{
        //Traverse the list until the position where we want to insert
        while((p!=NULL) && (k<position)){
            k++;
            q=p;
            p=p→next;
        }
        q→next=newNode;   //more optimum way to do this
        newNode→next=p;
    }
}
```

- **Note:** We can implement the three variations of the *insert* operation separately.
- Time Complexity: O(*n*), since, in the worst case, we may need to insert the node at the end of the list.
- Space Complexity: O(1), for creating one temporary variable.

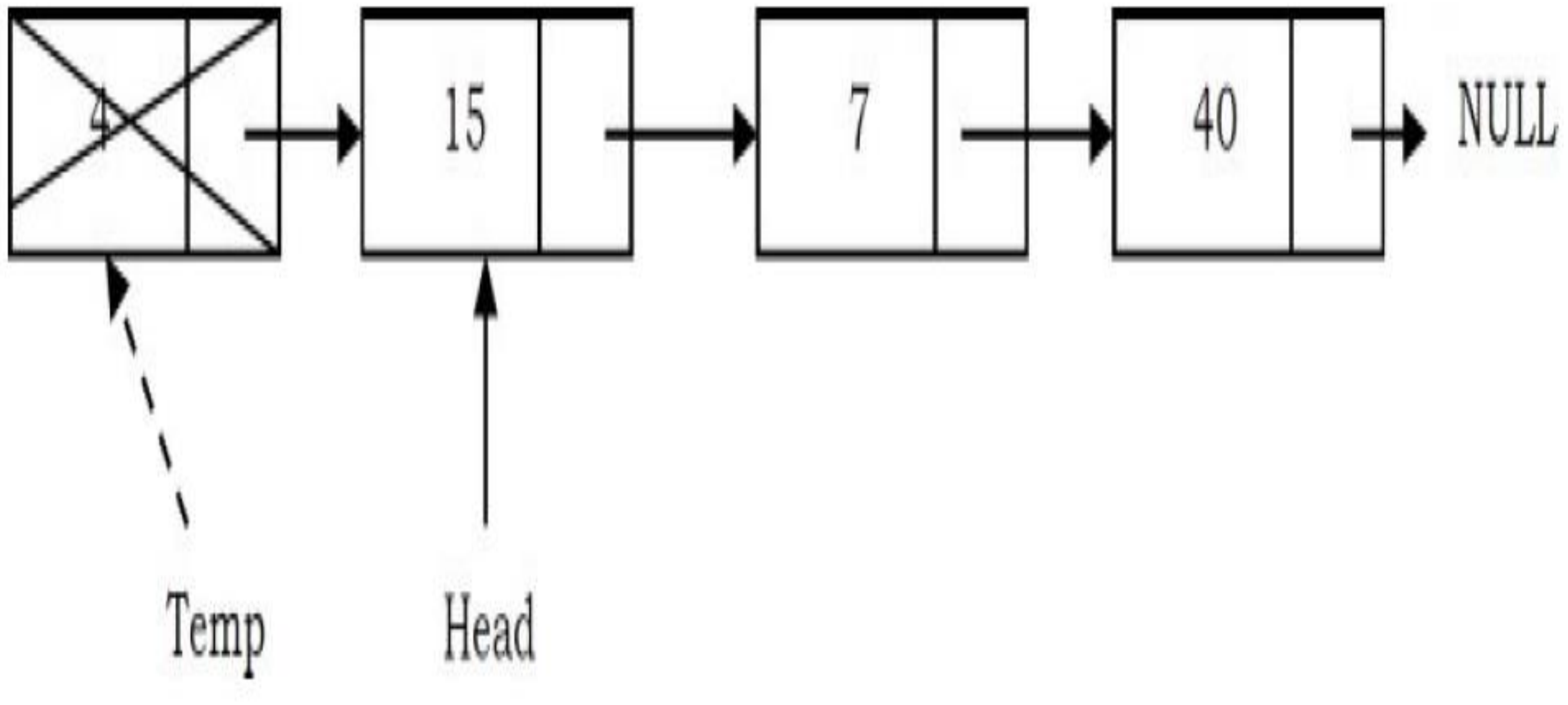
- **Singly Linked List Deletion**
- Similar to insertion, here we also have three cases.
  - Deleting the first node
  - Deleting the last node
  - Deleting an intermediate node.

Deleting the First Node in Singly Linked List

- First node (current head node) is removed from the list. It can be done in two steps:
  - Create a temporary node which will point to the same node as that of head.

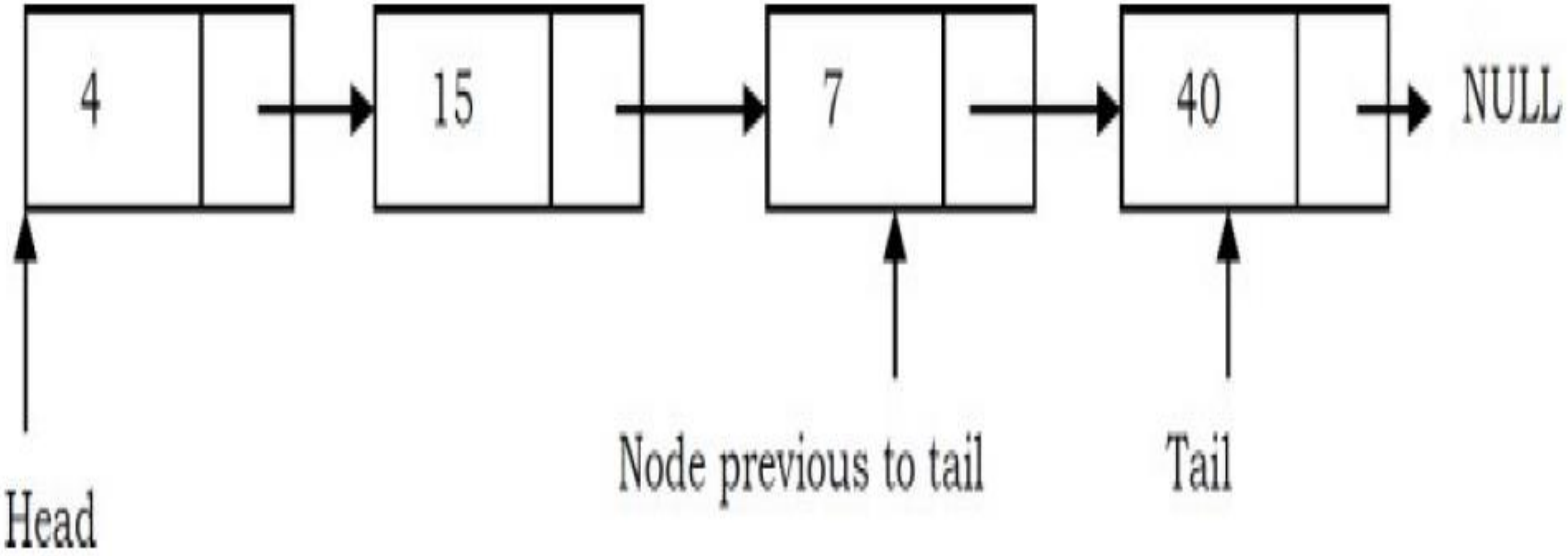


- Now, move the head nodes pointer to the next node and dispose of the temporary node.

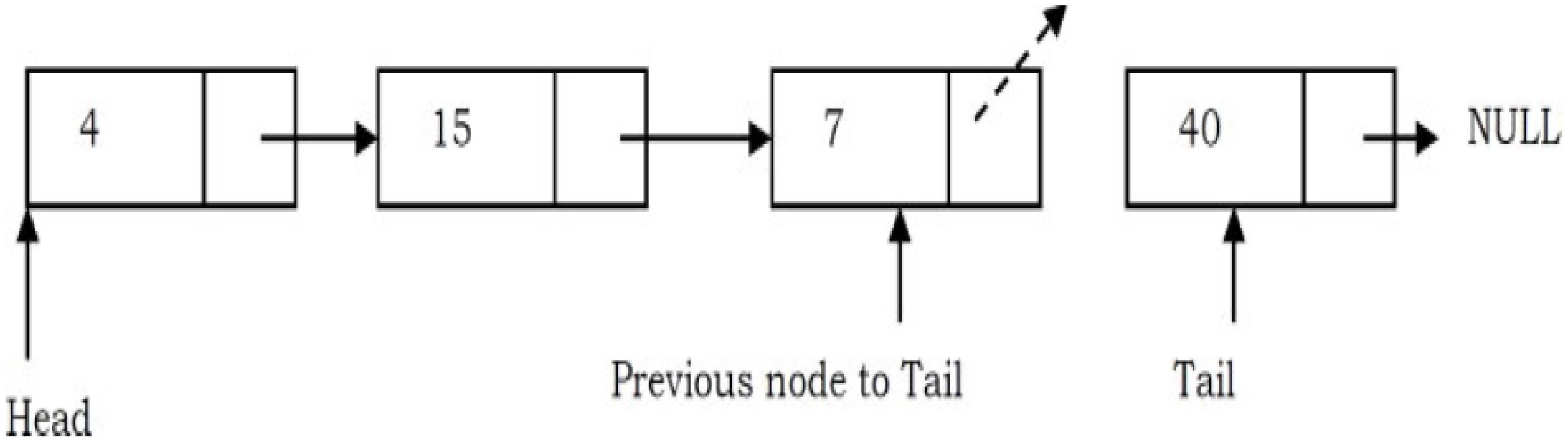


Deleting the Last Node in Singly Linked List

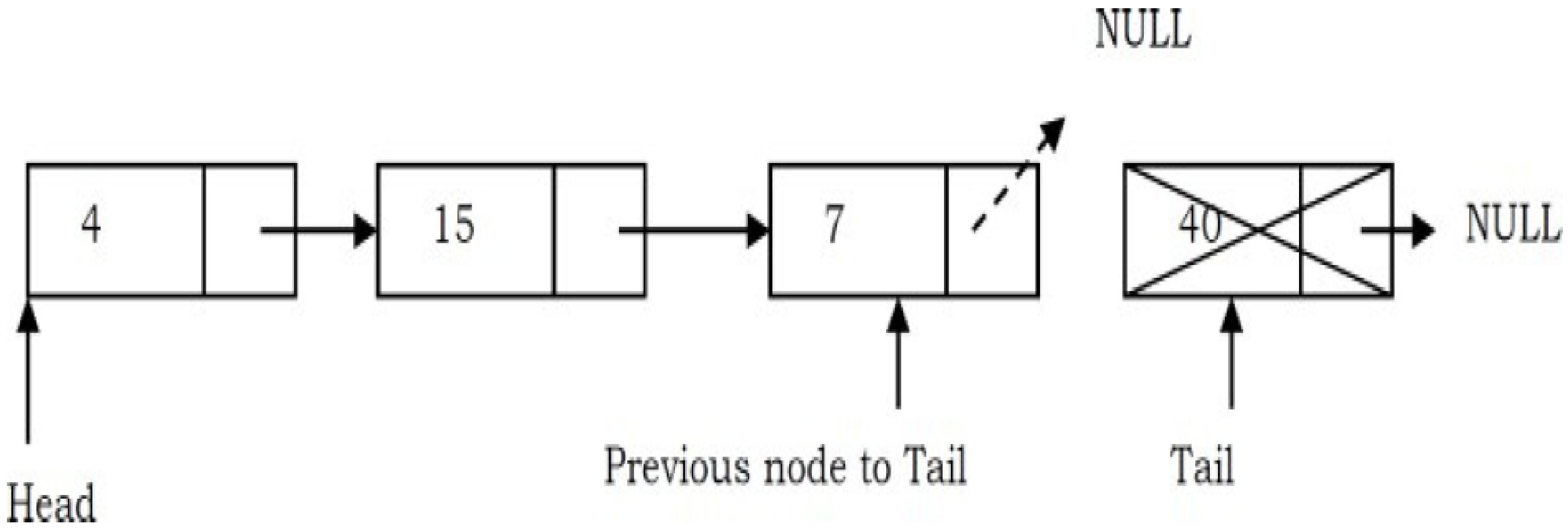
- In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail. It can be done in three steps:
  - Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.


4
15
7
40
NULL
Head
Node previous to tail
Tail

- Update previous node’s next pointer with NULL.

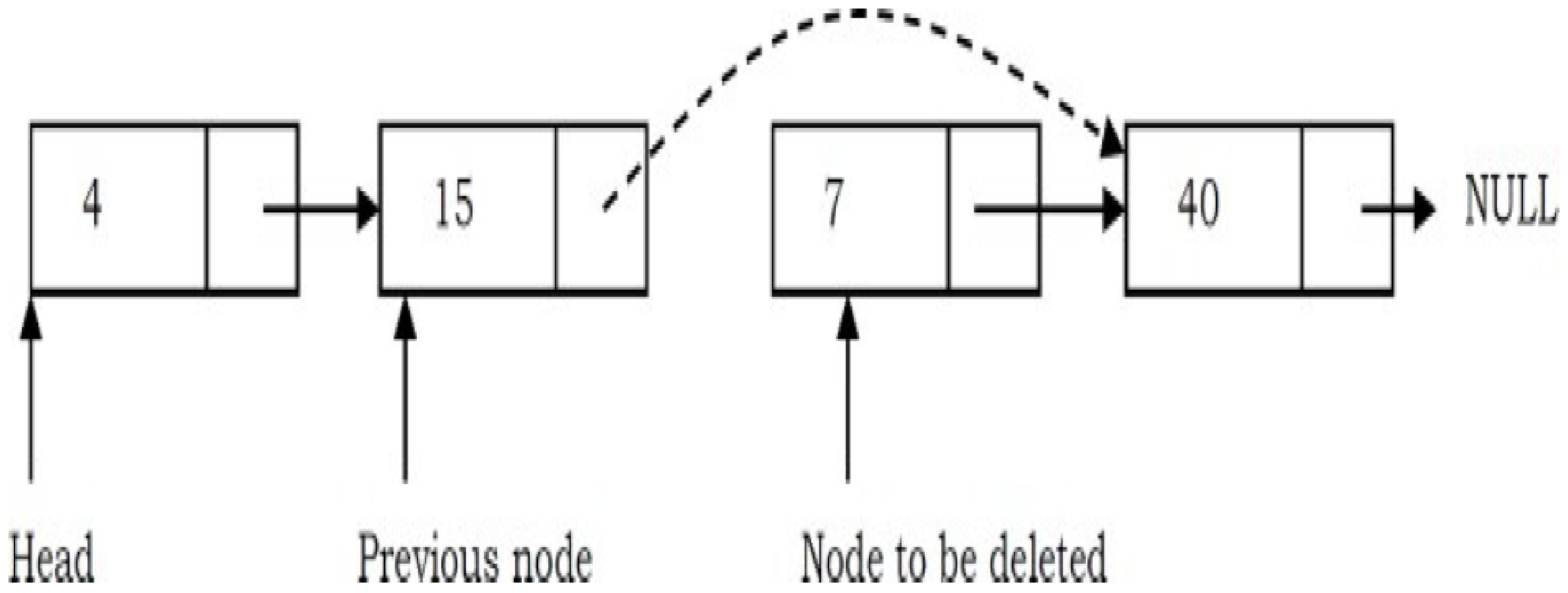


– Dispose of the tail node.

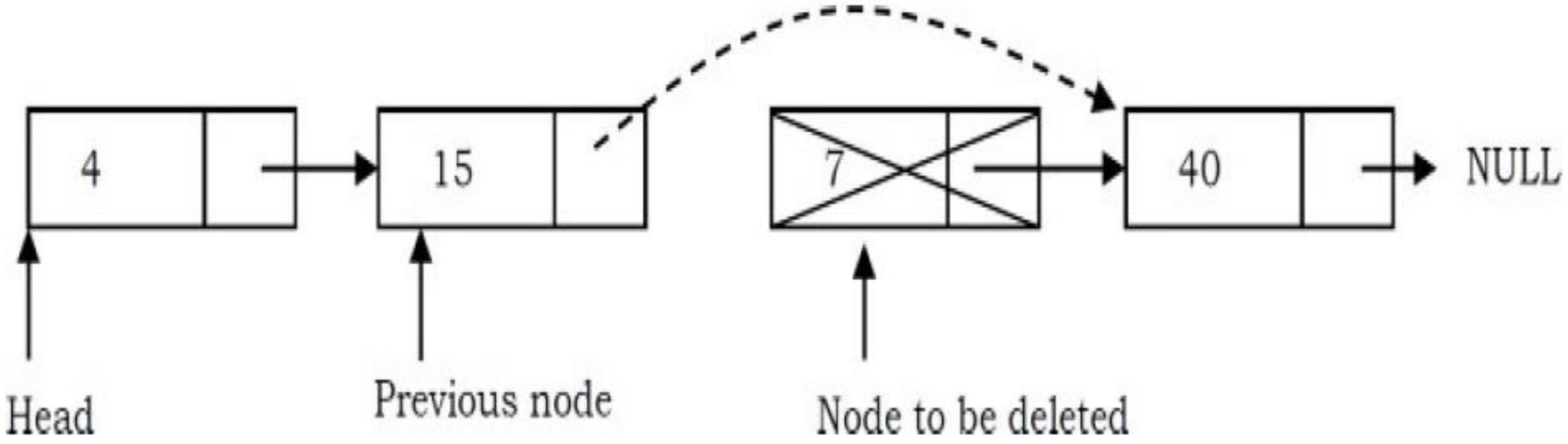


Deleting an Intermediate Node in Singly Linked List

- In this case, the node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:
  - Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.


4
15
7
40
NULL
Head
Previous node
Node to be deleted

– Dispose of the current node to be deleted.



```
void DeleteNodeFromLinkedList (struct ListNode **head, int position) {
    int k = 1;
    struct ListNode *p, *q;
    if(*head == NULL) {
       printf ("List Empty");
       return;
    }
    p = *head;
    /* from the beginning */
    if(position == 1) {
       *head = (*head)→next;
       free (p);
       return;
    }
    else {
        //Traverse the list until arriving at the position from which we want to delete
        while ((p != NULL) && (k < position )) {
            k++;
            q = p;
            p = p→next;
        }
        if(p == NULL)                          /* At the end */
            printf ("Position does not exist.");
        else {                                 /* From the middle */
            q→next = p→next;
            free(p);
        }
    }
}
```

- Time Complexity: O(*n*). In the worst case, we may need to delete the node at the end of the list.
- Space Complexity: O(1), for one temporary variable.

Deleting Singly Linked List

- This works by storing the current node in some temporary variable and freeing the current node.
- After freeing the current node, go to the next node with a temporary variable and repeat this process for all nodes.

```
void DeleteLinkedList(struct ListNode **head) {
    struct ListNode *auxilaryNode, *iterator;
    iterator = *head;
    while (iterator) {
        auxilaryNode = iterator→next;
        free(iterator);
        iterator = auxilaryNode;
    }
    *head = NULL;                    // to affect the real head back in the caller.
}
```

Time & Space Complexity

- Time Complexity: O(*n*), for scanning the complete list of size n.
- Space Complexity: O(1), for creating one temporary variable.

Data Structure	Time Complexity								Space Complexity
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

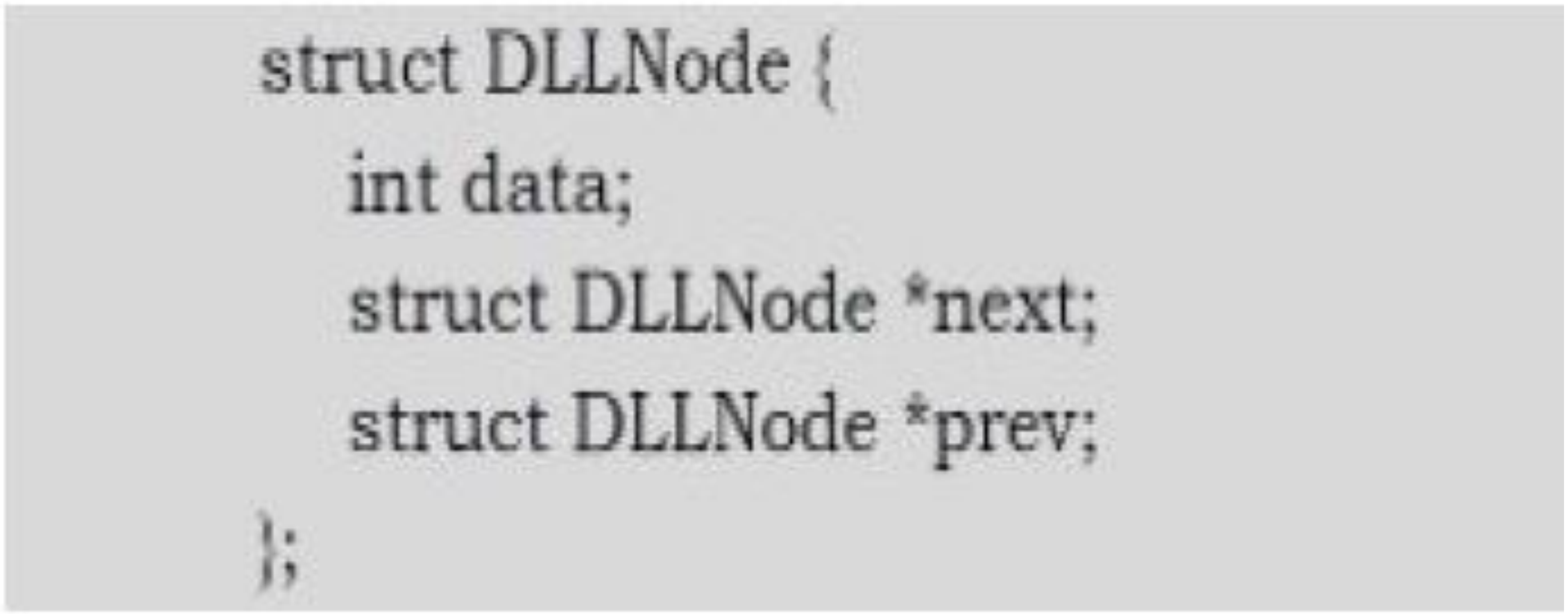
Doubly Linked Lists

- The *advantage* of a doubly linked list (also called *two – way linked list*) is that given a node in the list, we **can navigate in both directions**.
- A node in a singly linked list cannot be removed unless we have the pointer to its predecessor.
  - But in a doubly linked list, **we can delete a node even if we don’t have the previous node’s address (since each node has a left pointer pointing to the previous node and can move backward).**

Disadvantages

- The primary *disadvantages* of doubly linked lists are:
  - **Each node requires an extra pointer, requiring more space.**
  - **The insertion or deletion of a node takes a bit longer (more pointer operations).**

- Following is a type declaration for a doubly linked list of integers:



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

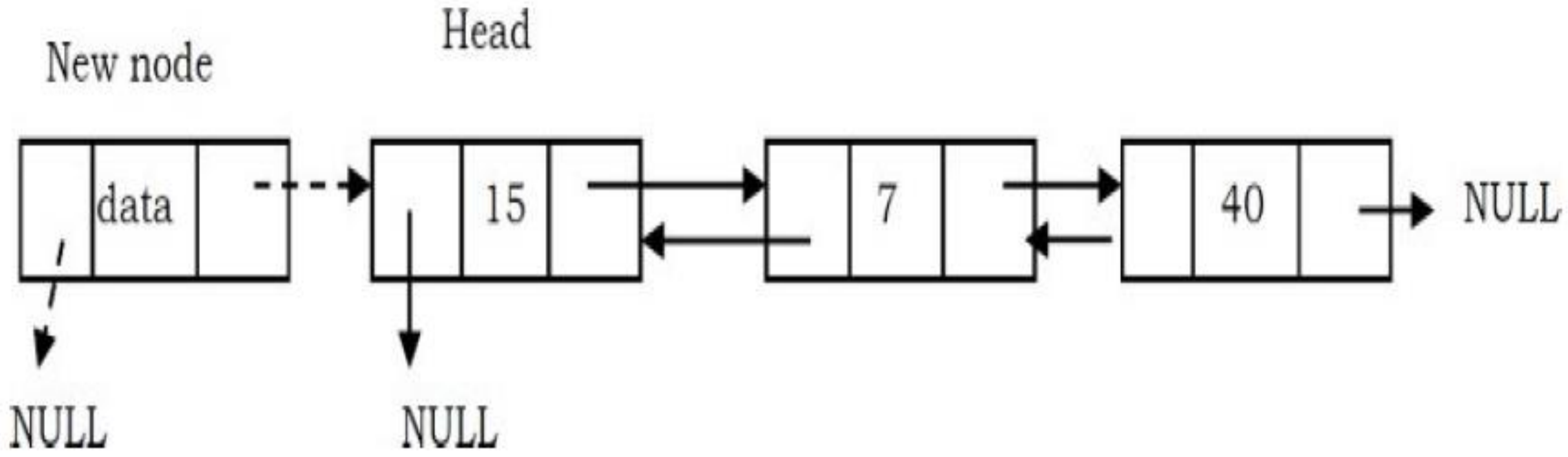
Doubly Linked List Insertion

- Insertion into a doubly-linked list has three cases (same as singly linked list):
  - Inserting a **new node before the head**.
  - Inserting a **new node after the tail** (at the **end of the list**).
  - Inserting a **new node at the middle of the list**.

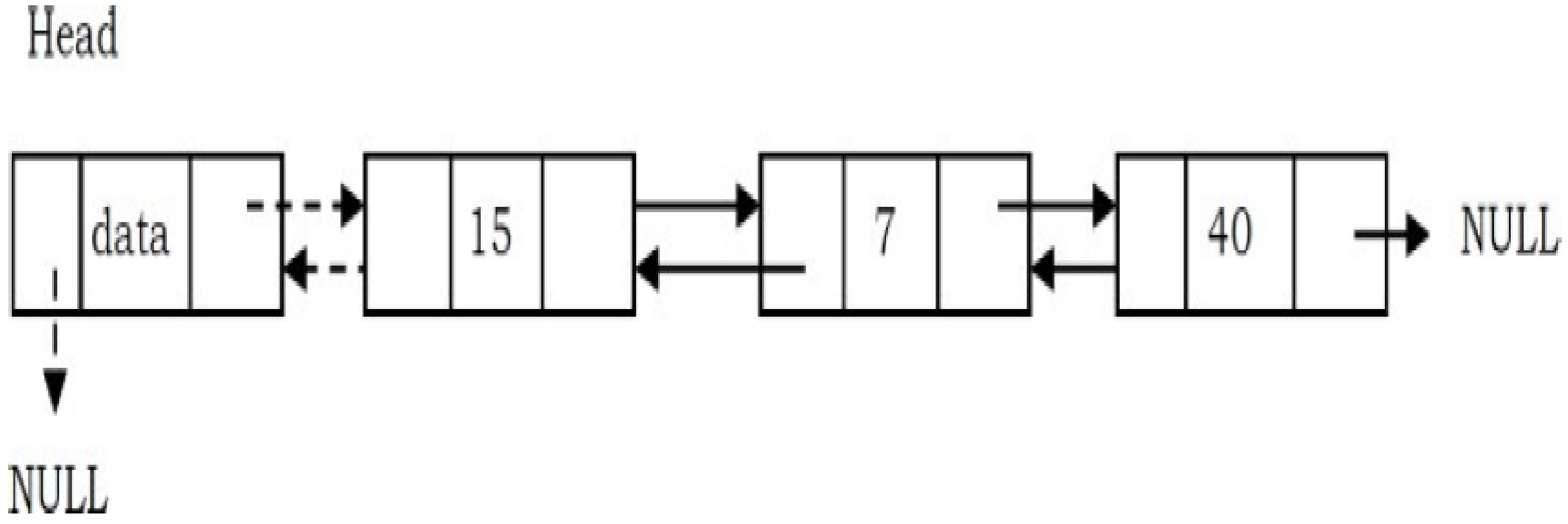
Inserting a Node in Doubly Linked List at the Beginning

- In this case, new node is inserted before the head node.
  - Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.

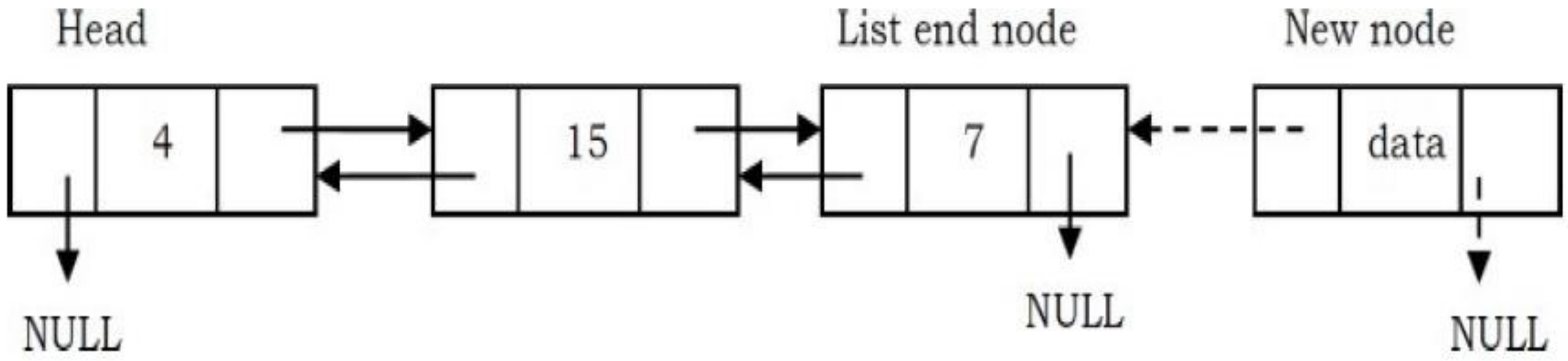


- Update head node's left pointer to point to the new node and make new node as head.

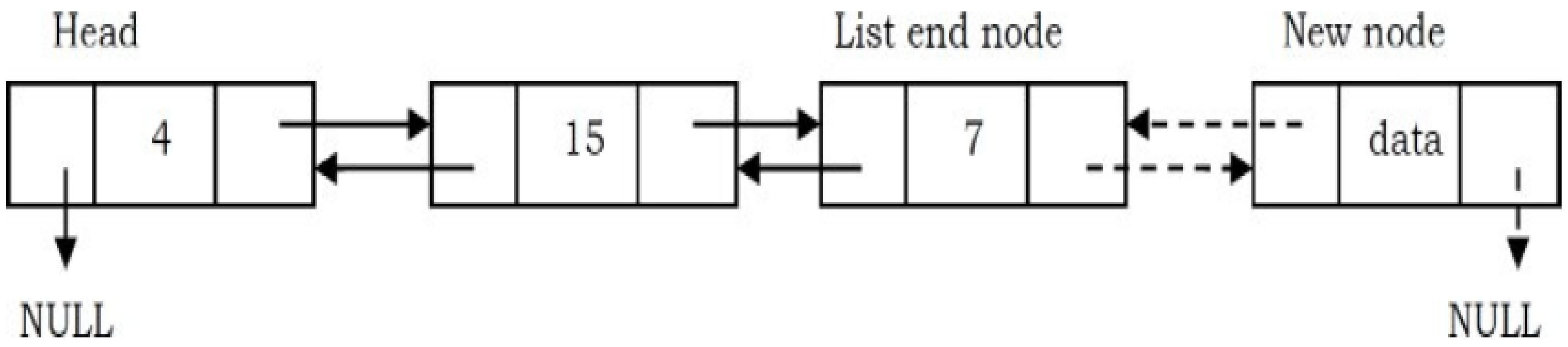


Inserting a Node in Doubly Linked List at the Ending

- In this case, traverse the list till the end and insert the new node.
  - New node **right pointer points to NULL**
  - **left pointer points** to the **end of the list.**



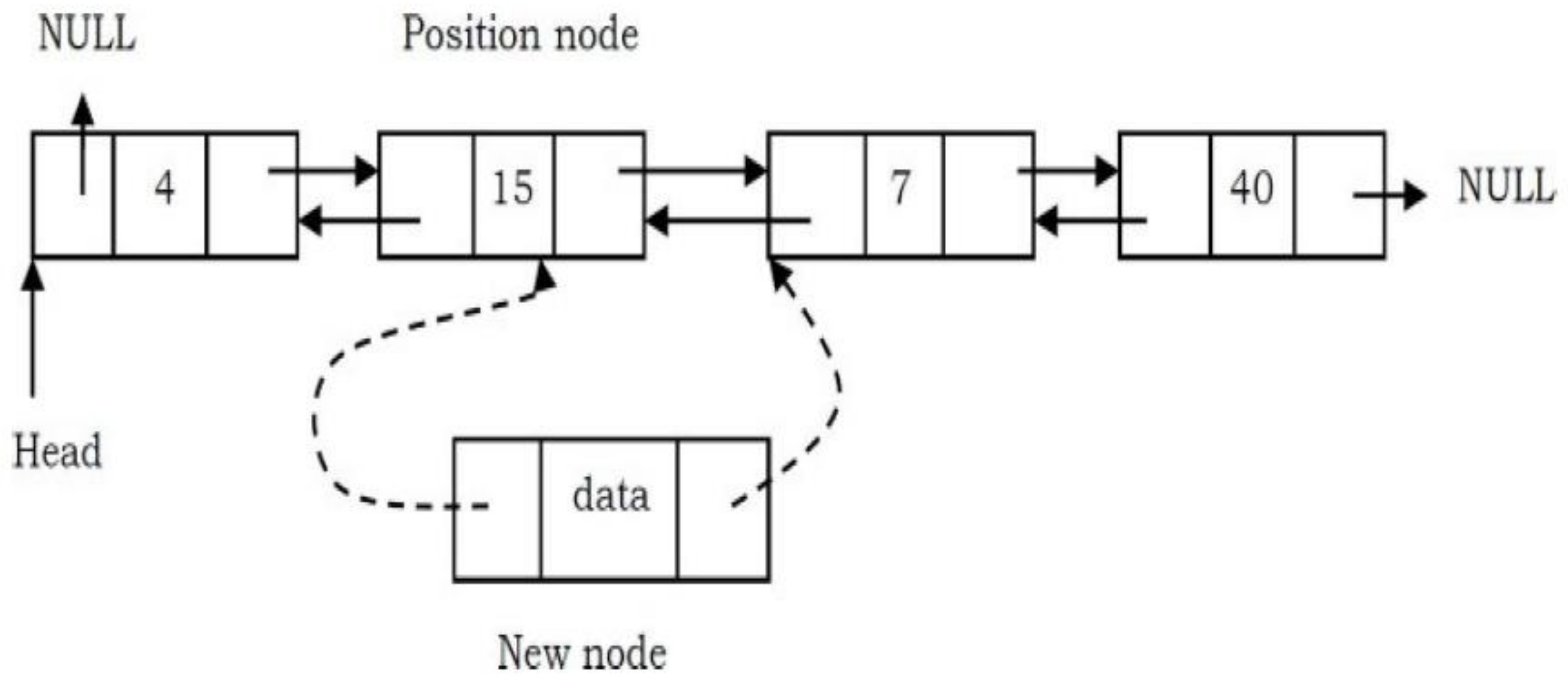
– **Update right pointer of last node to point to new node.**



Inserting a Node in Doubly Linked List at the Middle

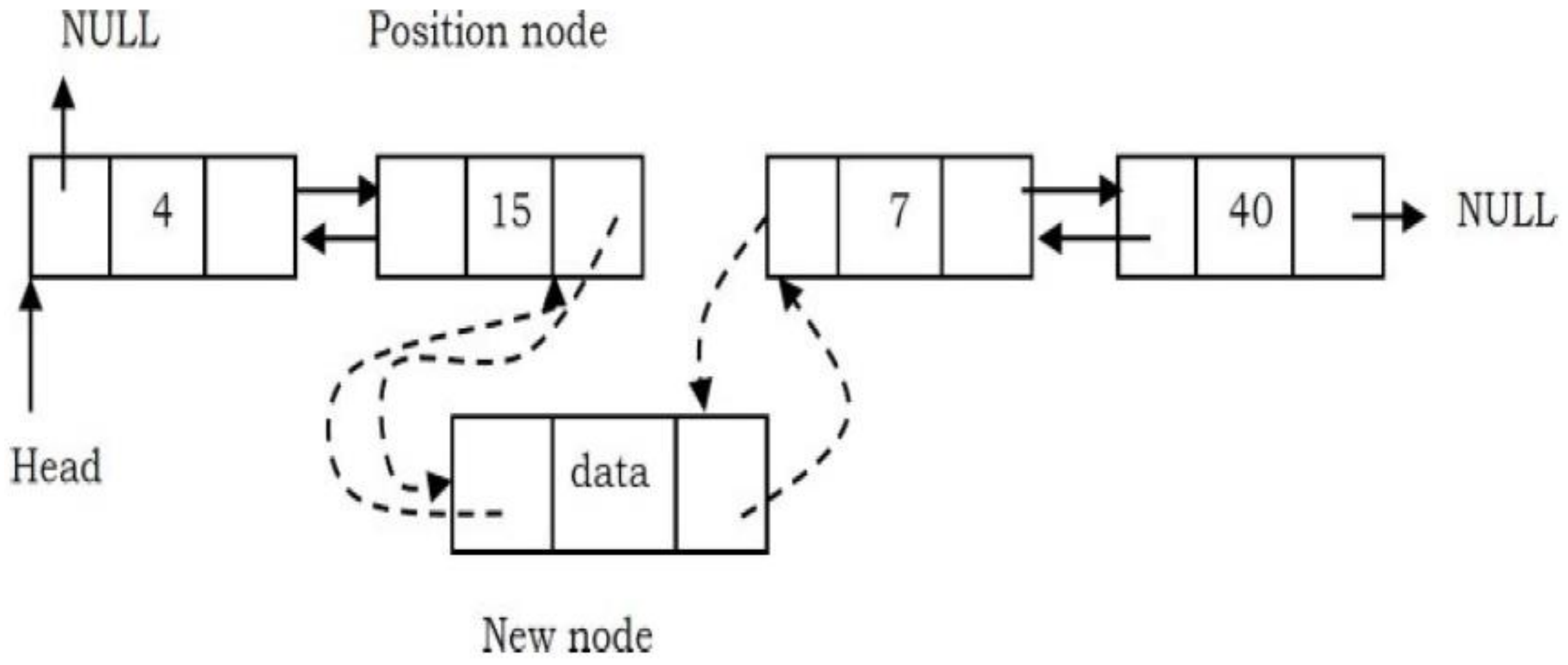
- Traverse the list to the position node and insert the new node.
  - ***New node* right pointer points to the next node of the *position node*** where we want to insert the new node.
  - ***New node* left pointer points to the *position node.***

I Step



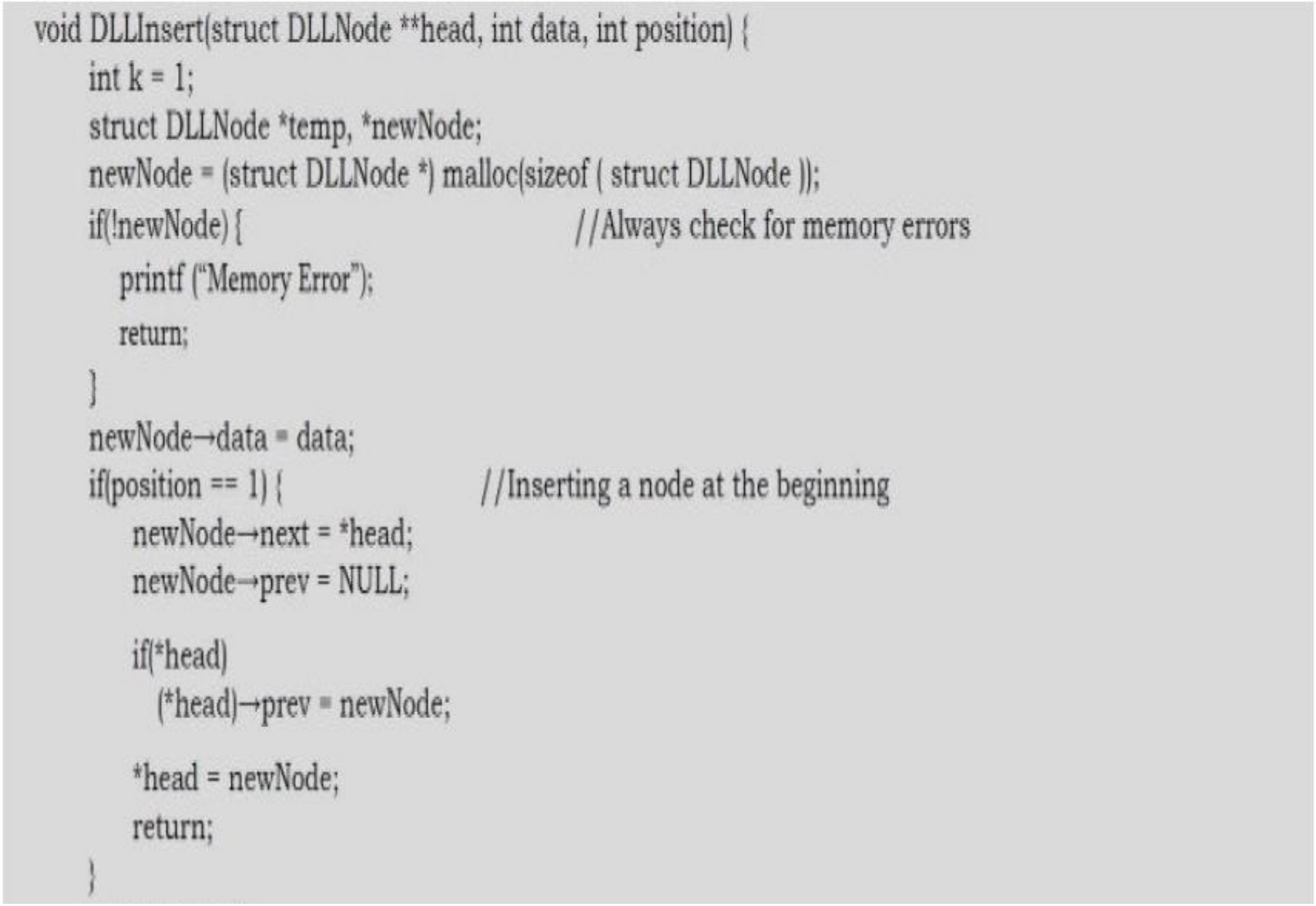
II Step

- **Position node right pointer points to the new node**
- **The *next node* of position node left pointer points to new node.**



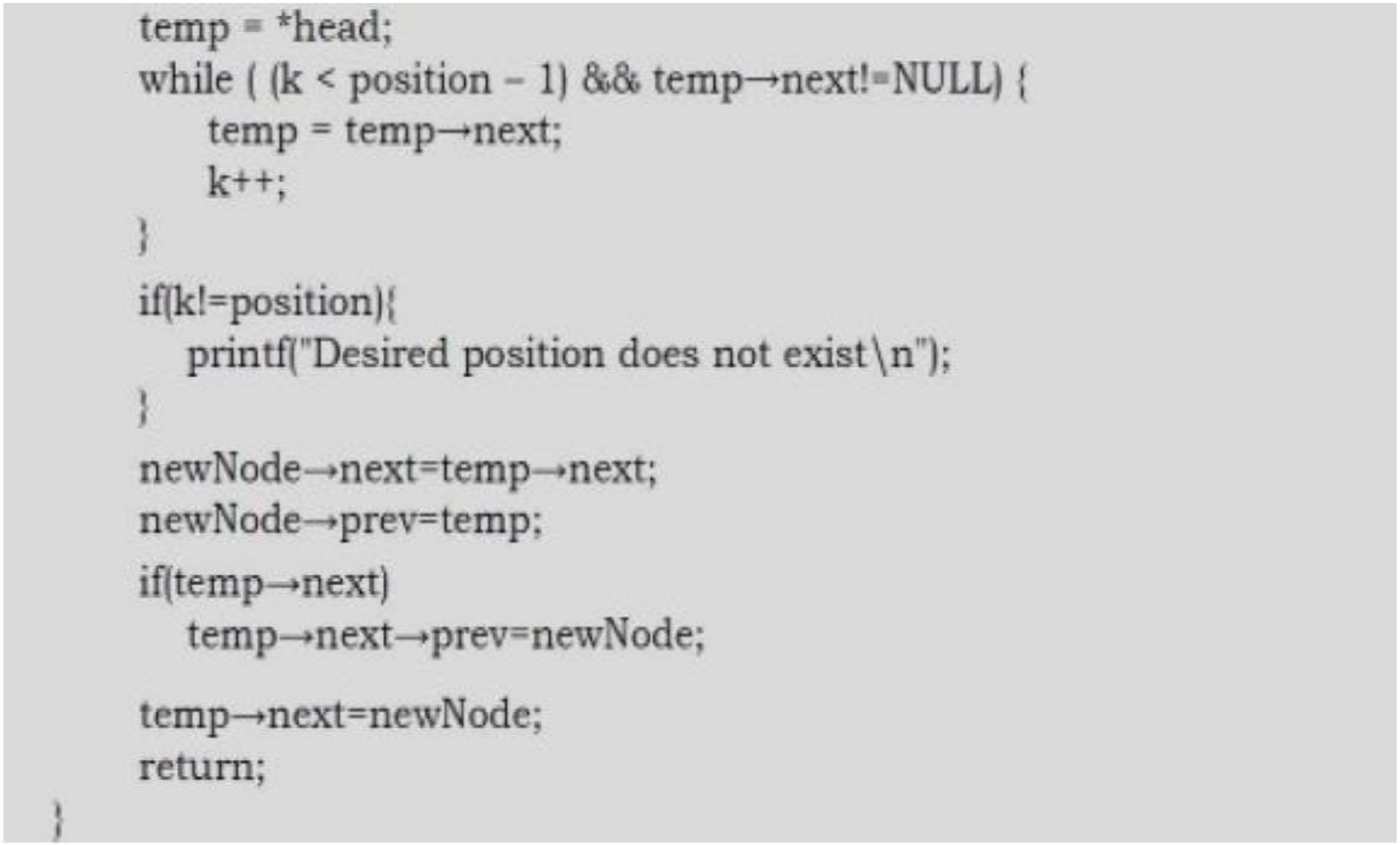
- Now, let us write the code for all of these three cases.
  - Update the first element pointer in the calling function, not just in the called function.
  - For this reason we need to send a double pointer.
  - The following code inserts a node in the doubly linked list

Code



```
void DLLInsert(struct DLLNode **head, int data, int position) {
    int k = 1;
    struct DLLNode *temp, *newNode;
    newNode = (struct DLLNode *) malloc(sizeof ( struct DLLNode ));
    if(!newNode) {                                    //Always check for memory errors
      printf ("Memory Error");
      return;
    }
    newNode→data = data;
    if(position == 1) {                    //Inserting a node at the beginning
        newNode→next = *head;
        newNode→prev = NULL;

        if(*head)
          (*head)→prev = newNode;

        *head = newNode;
        return;
    }
```

Code



```
    temp = *head;
    while ( (k < position - 1) && temp→next!=NULL) {
        temp = temp→next;
        k++;
    }
    if(k!=position){
      printf("Desired position does not exist\n");
    }
    newNode→next=temp→next;
    newNode→prev=temp;
    if(temp→next)
      temp→next→prev=newNode;
    temp→next=newNode;
    return;
}
```

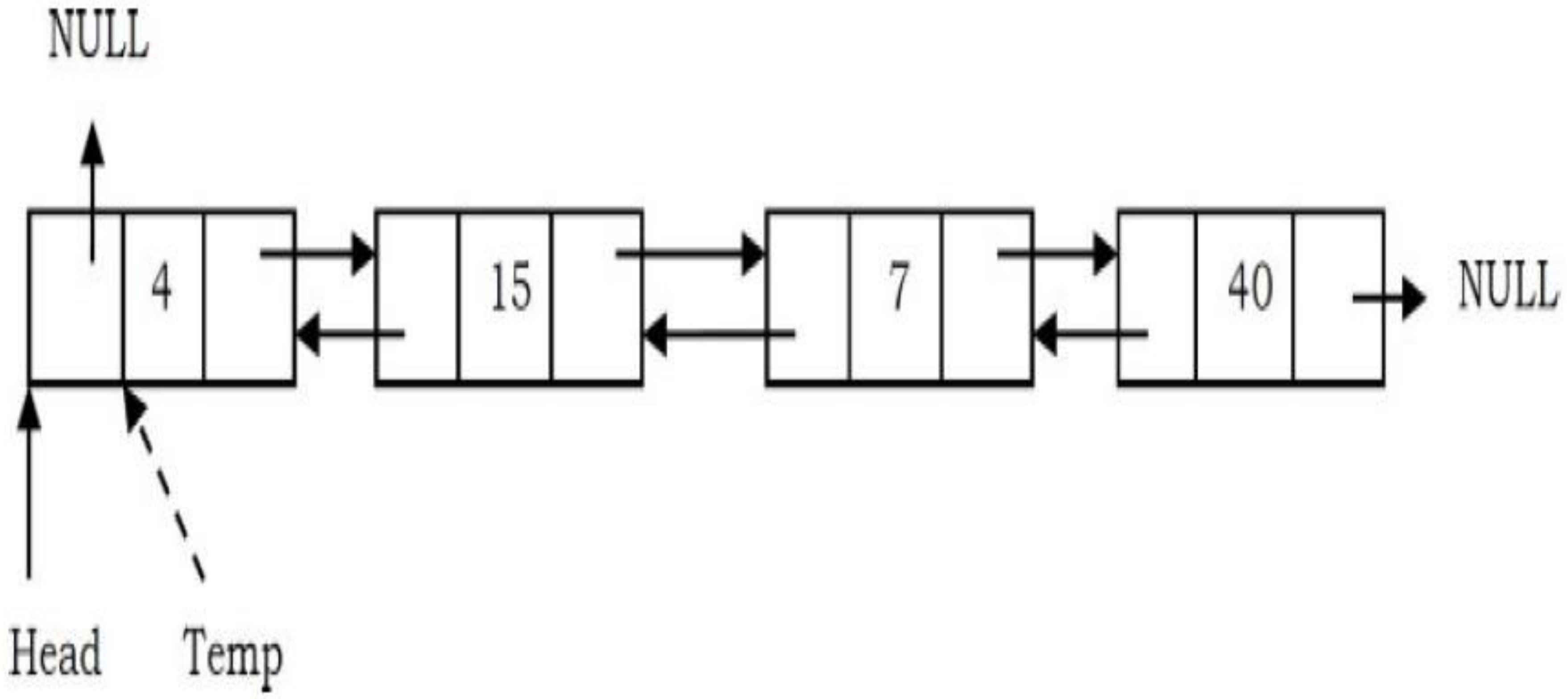
- Time Complexity: O(*n*). In the worst case, we may need to insert the node at the end of the list.
- Space Complexity: O(1), for creating one temporary variable.

Doubly Linked List Deletion

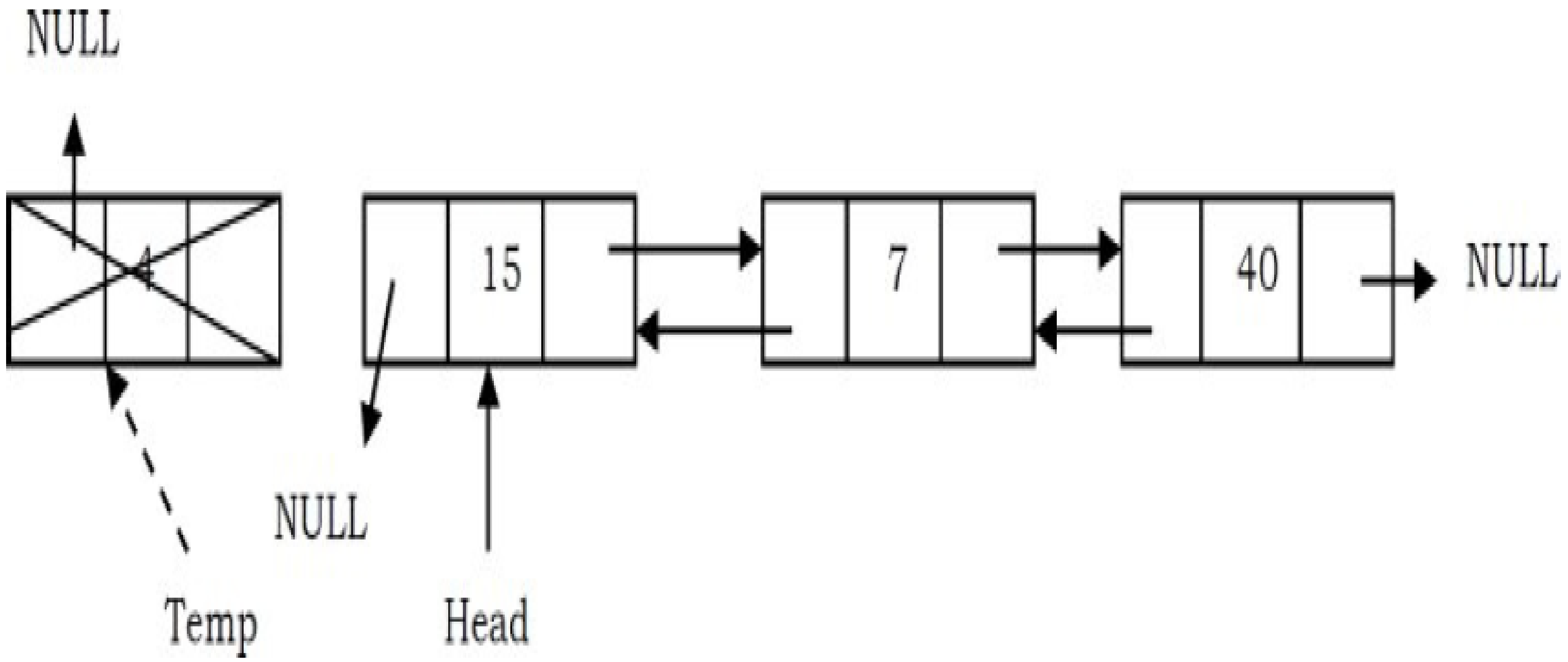
- Similar to singly linked list deletion, here we have three cases:
  - Deleting the first node
  - Deleting the last node
  - Deleting an intermediate node

Deleting the First Node in Doubly Linked List

- In this case, the first node (current head node) is removed from the list. It can be done in two steps:
  - Create a temporary node which will point to the same node as that of head.

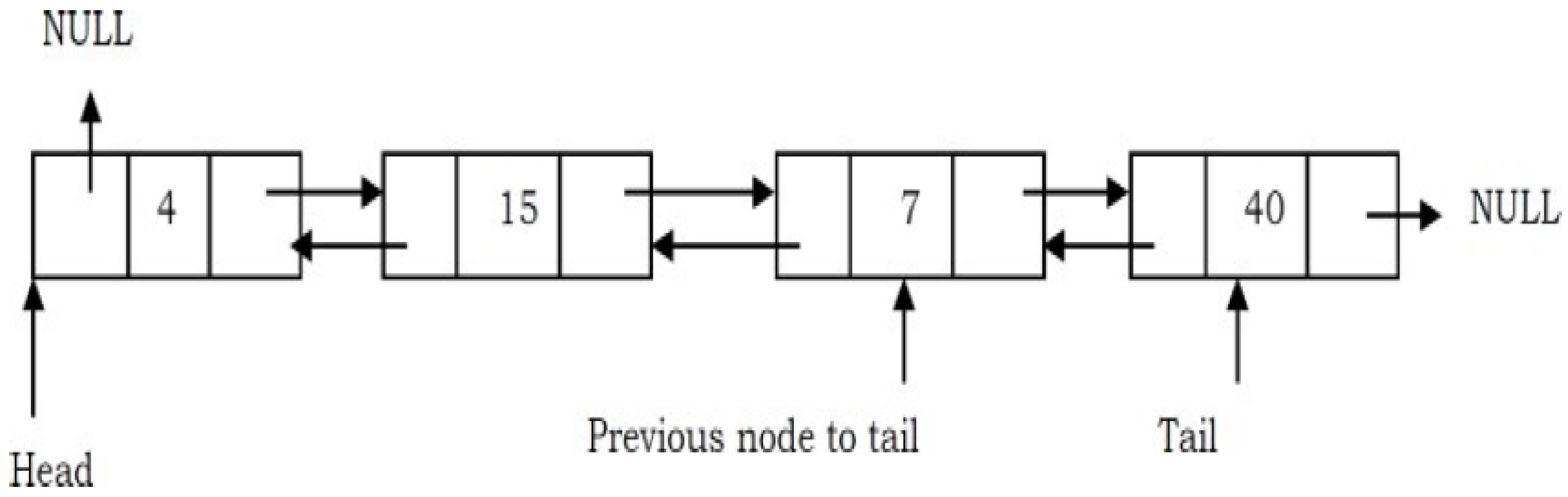

NULL
4
15
7
40
NULL
Head
Temp

– Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.

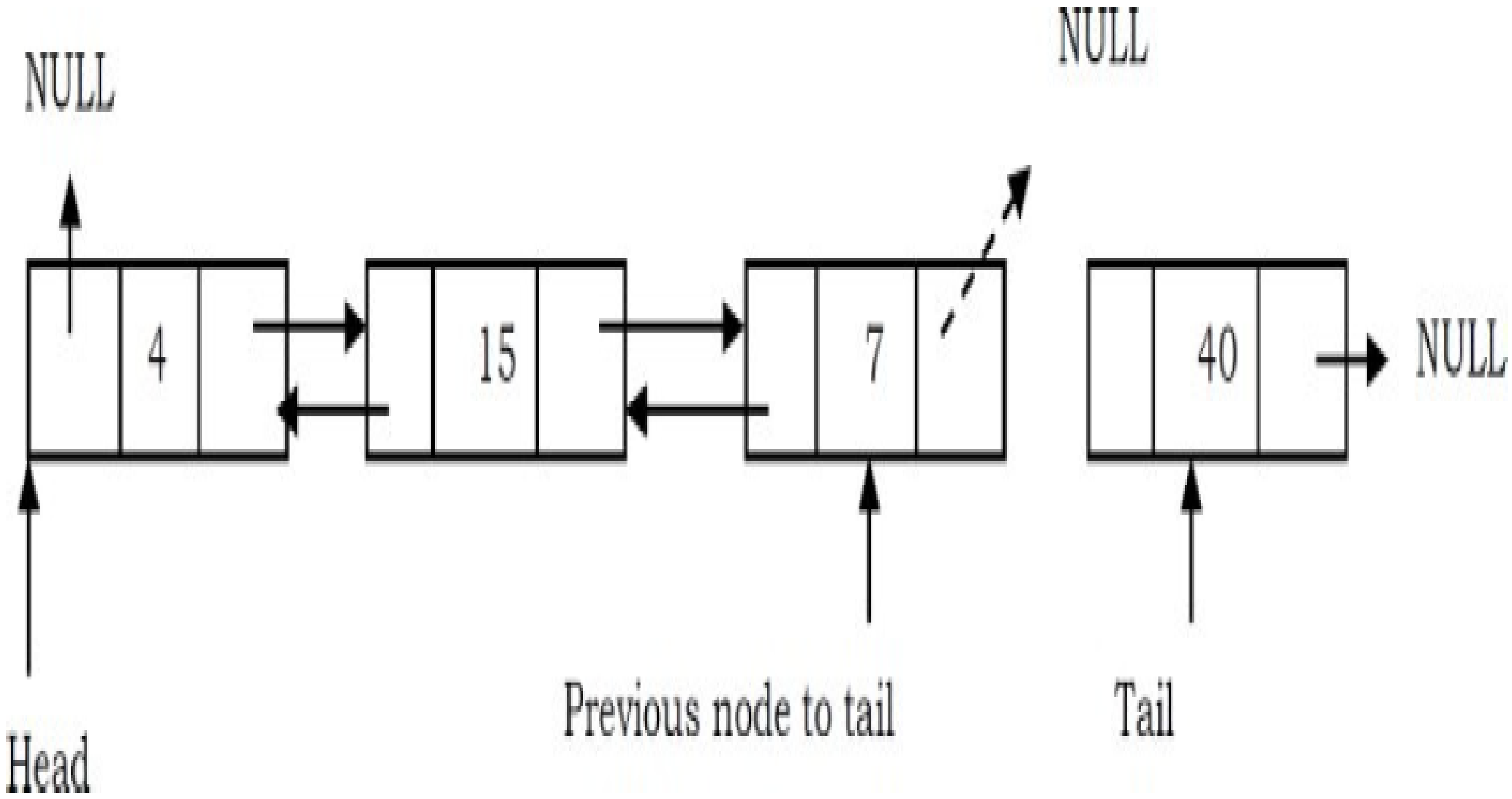

NULL
4
15
7
40
NULL
NULL
Temp
Head

Deleting the Last Node in Doubly Linked List

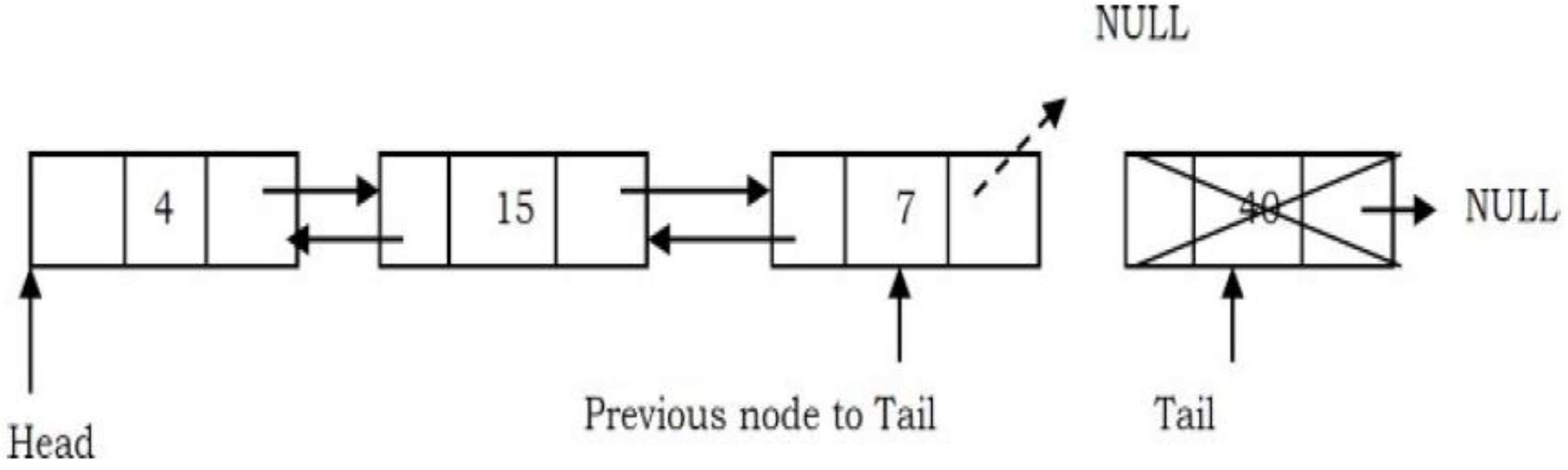
- This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail first. This can be done in three steps:
  - Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.


NULL
4
15
7
40
NULL
Head
Previous node to tail
Tail

  – Update the next pointer of previous node to the tail node with NULL.

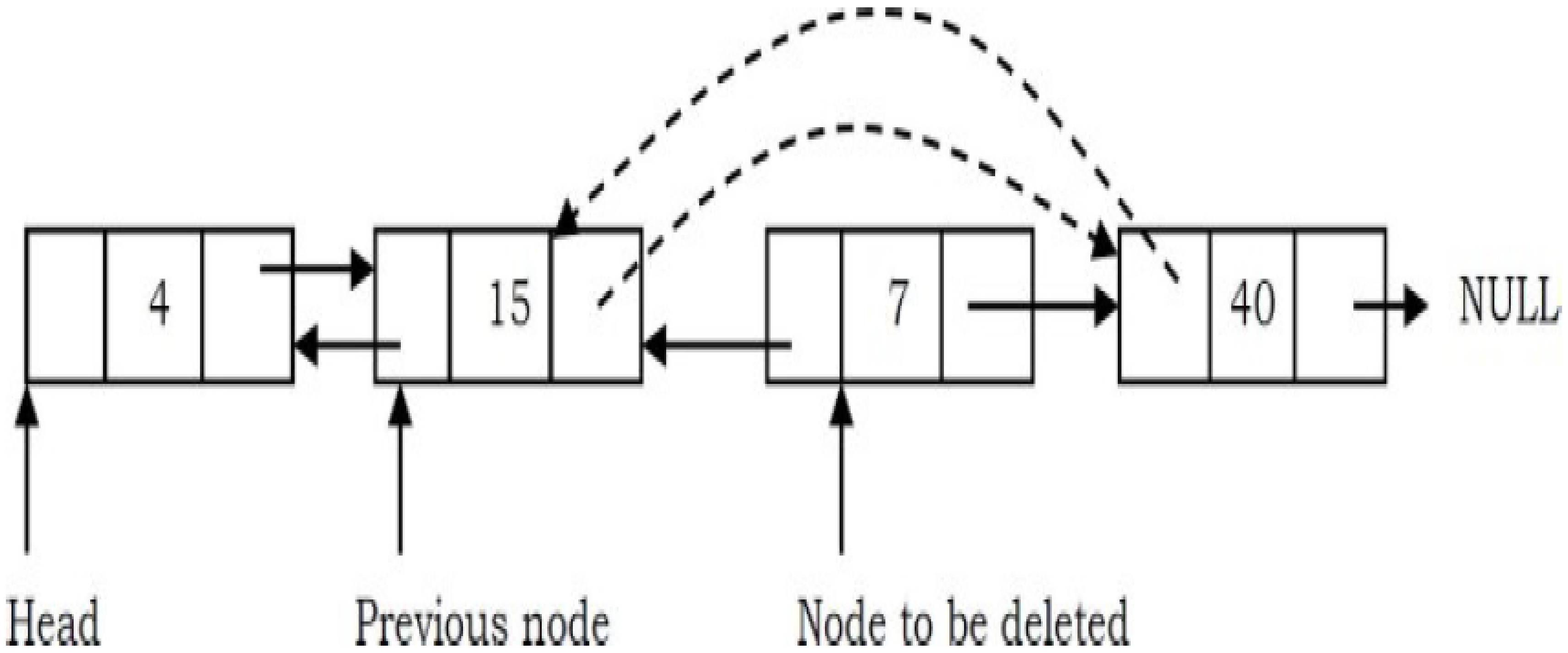

NULL
NULL
4
15
7
40
NULL
Previous node to tail
Tail
Head

– Dispose the tail node.

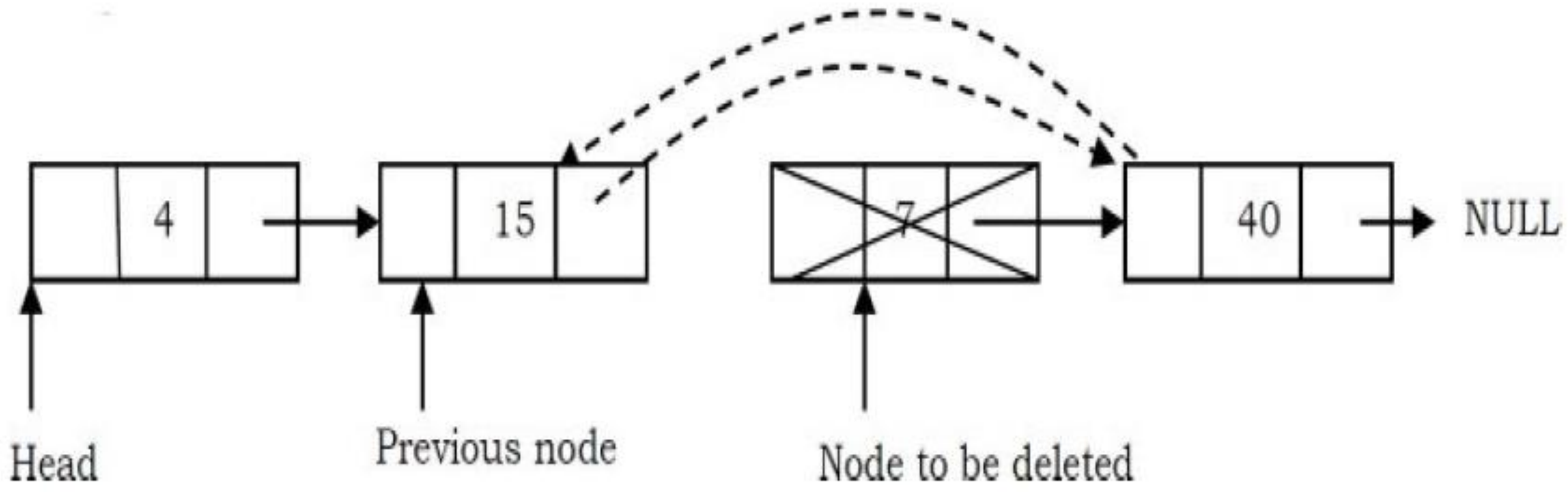


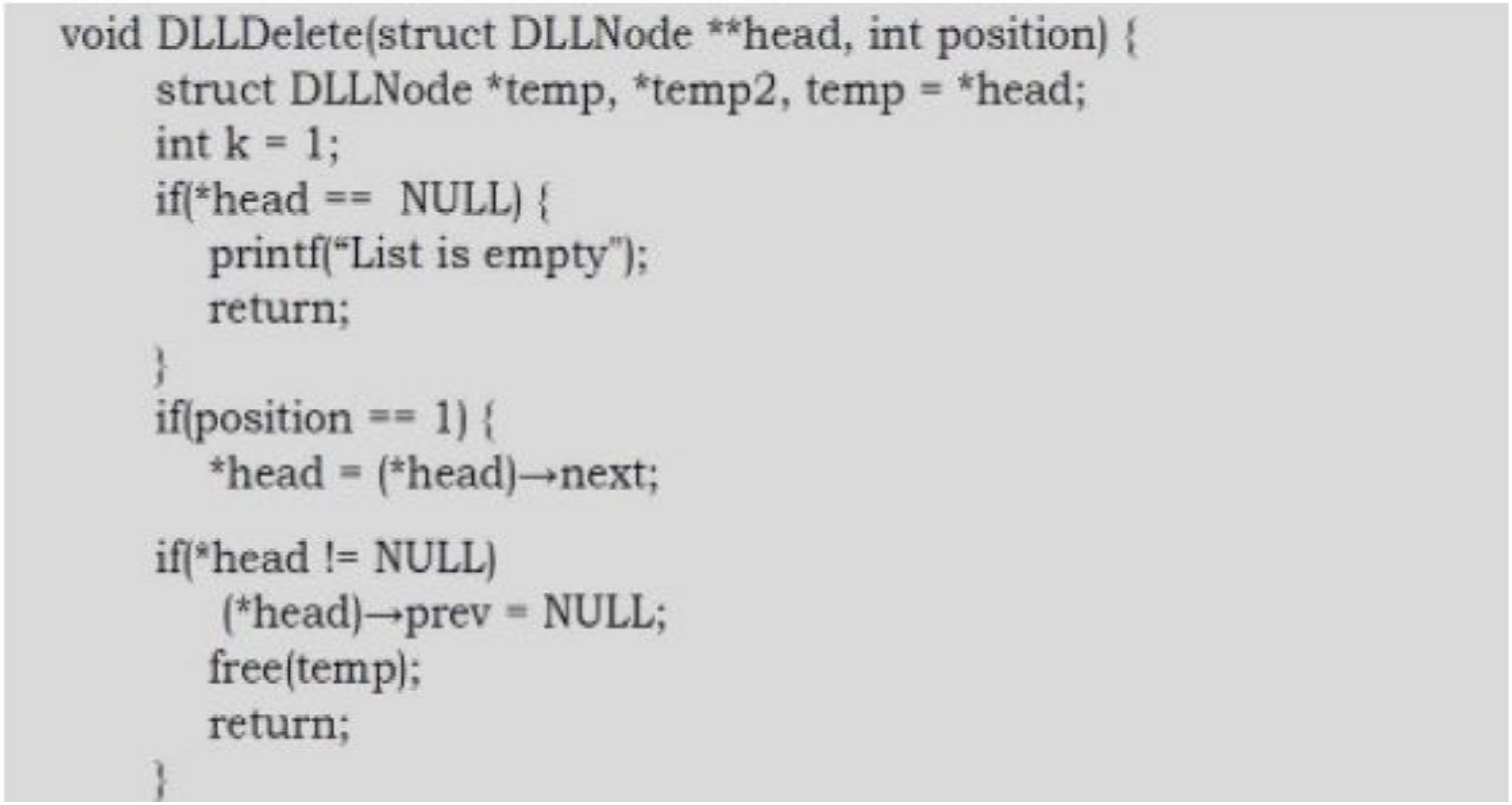
Deleting an Intermediate Node in Doubly Linked List

- In this case, the node to be removed is *always located between* two nodes, and the head and tail links are not updated. The removal can be done in two steps:
  - Similar to the previous case, maintain the previous node while also traversing the list. Upon locating the node to be deleted, change the previous node's next pointer to the next node of the node to be deleted.

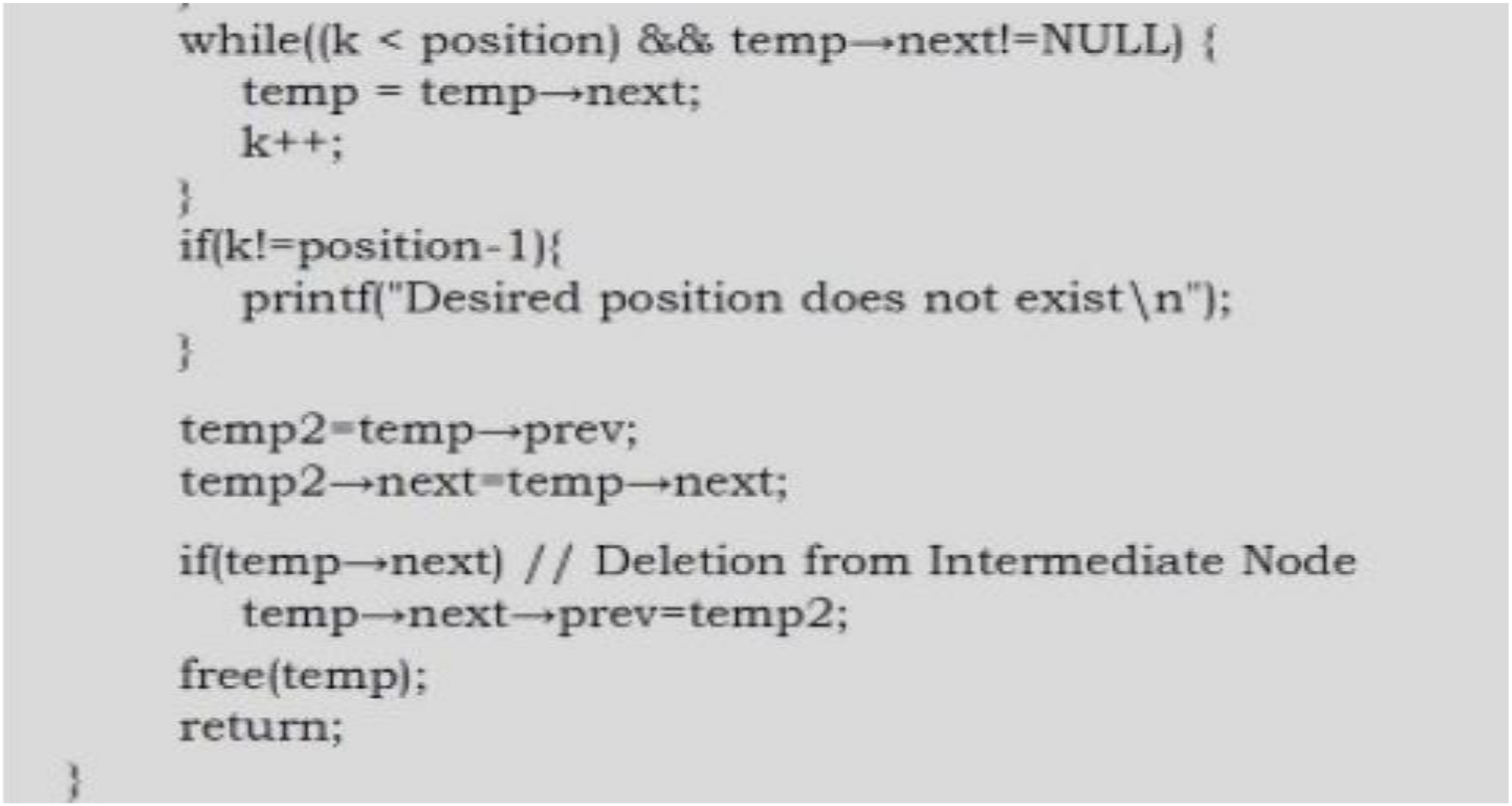

4
15
7
40
NULL
Head
Previous node
Node to be deleted

– Dispose of the current node to be deleted.





```
void DLLDelete(struct DLLNode **head, int position) {
      struct DLLNode *temp, *temp2, temp = *head;
      int k = 1;
      if(*head ==  NULL) {
         printf("List is empty");
         return;
      }
      if(position == 1) {
         *head = (*head)→next;

      if(*head != NULL)
          (*head)→prev = NULL;
         free(temp);
         return;
      }
```



```
    while((k < position) && temp→next!=NULL) {
        temp = temp→next;
        k++;
    }
    if(k!=position-1){
        printf("Desired position does not exist\n");
    }

    temp2=temp→prev;
    temp2→next=temp→next;

    if(temp→next) // Deletion from Intermediate Node
        temp→next→prev=temp2;
    free(temp);
    return;
}
```

- Time Complexity: O(*n*), for scanning the complete list of size *n*.
- Space Complexity: O(1), for creating one temporary variable.

Circular Linked Lists

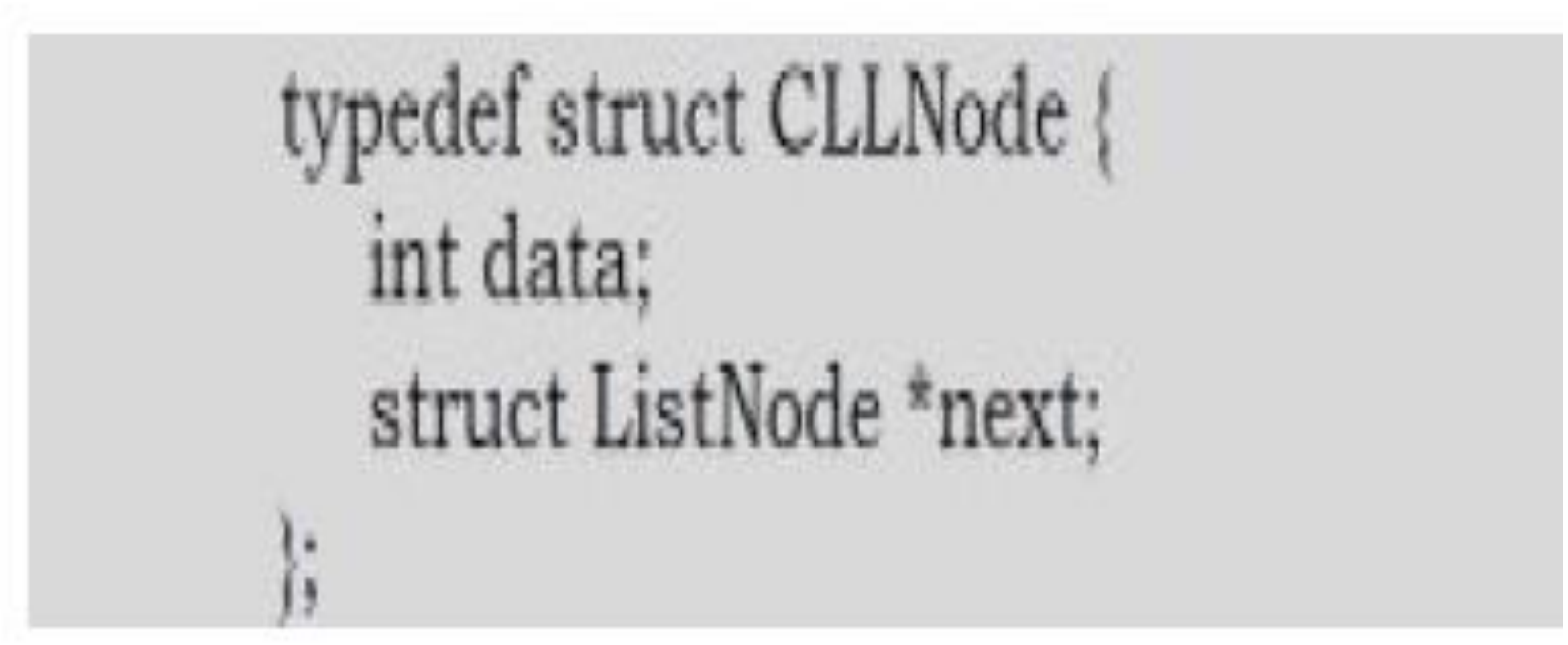
- In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value.
- Circular linked lists do not have ends.
- In circular linked lists, each node has a successor.
- In singly linked lists, there is no node with NULL pointer in a circularly linked list.

Usage of circular linked lists

- When several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes do (round robin algorithm).

Declaration of node

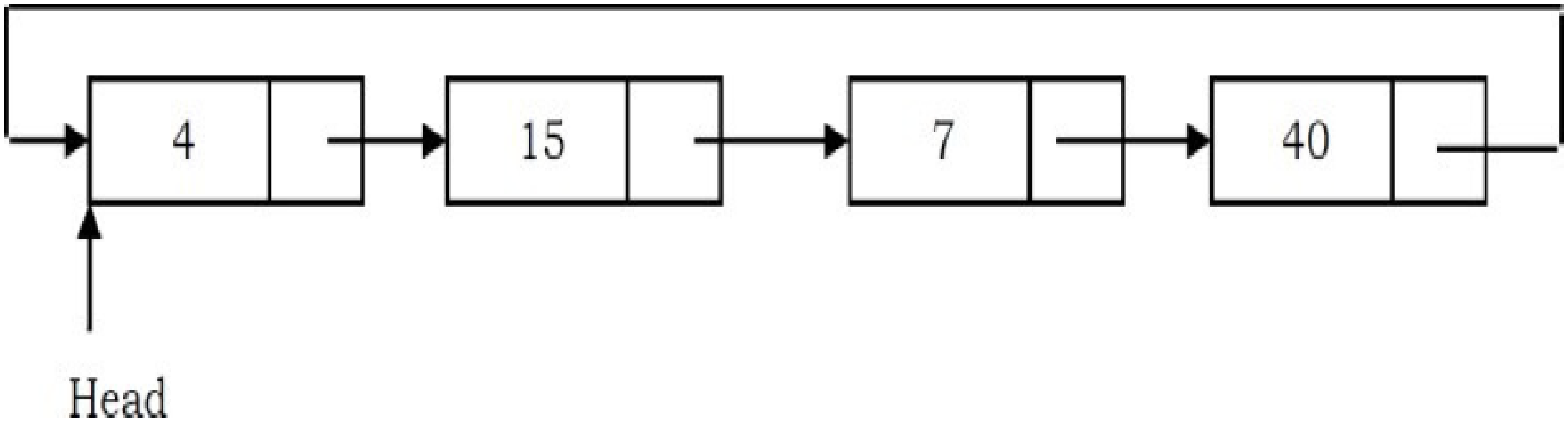
- The following is a type declaration for a circular linked list of integers:



```
typedef struct CLLNode {
    int data;
    struct ListNode *next;
};
```

- In a circular linked list, we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists).

Counting Nodes in a Circular Linked List



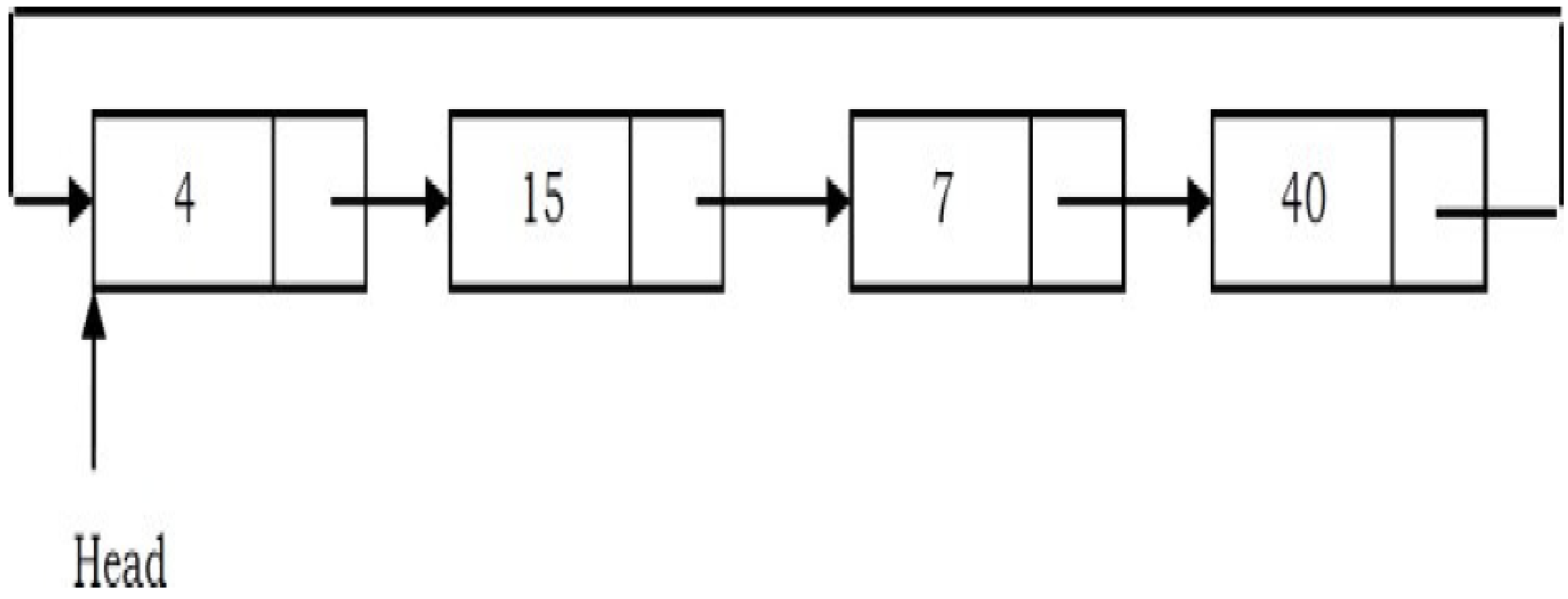
- The circular list is accessible through the node marked *head*.
- To count the nodes, the list has to be traversed from the node marked *head*, with the help of a dummy node *current*, and stop the counting when *current* reaches the starting node *head.*
- If the list is empty, *head* will be NULL, and in that case set *count* = 0.
- Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

- Time Complexity: O(*n*), for scanning the complete list of size *n*.
- Space Complexity: O(1), for creating one temporary variable.

```
int CircularListLength(struct CLLNode *head) {
      struct CLLNode *current = head;
      int count = 0;
      if(head == NULL)
         return 0;

      do {
         current = current→next;
         count++;
      } while (current != head);

      return count;
}
```

Printing the Contents of a Circular Linked List

- We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node previous to the *head* node.
- Let us assume we want to print the contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.


4
15
7
40
Head

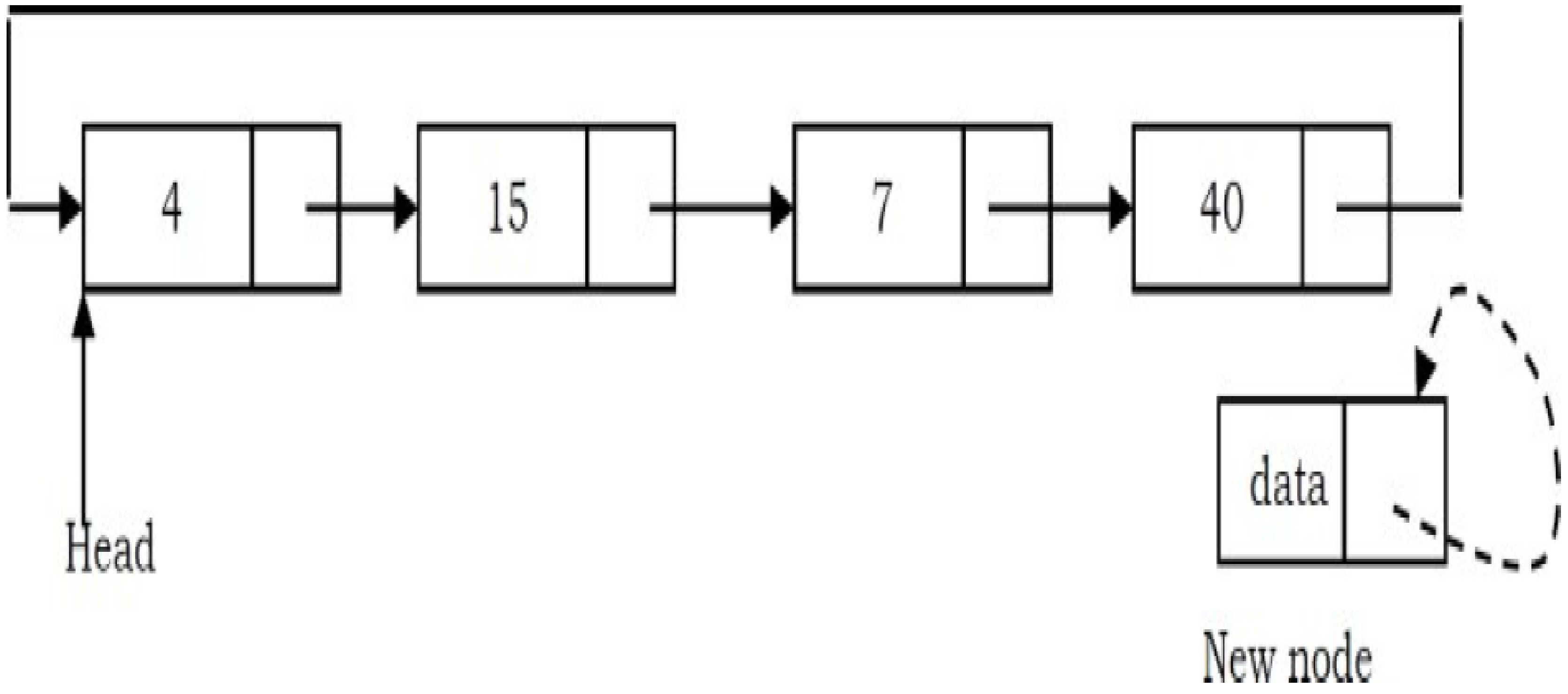
```
void PrintCircularListData(struct CLLNode *head) {
      struct CLLNode *current = head;
      if(head == NULL)
         return;

      do {
         printf ("%d", current→data);
         current = current→next;
      } while (current != head);
}
```

- Time Complexity: O(*n*), for scanning the complete list of size *n.*
- Space Complexity: O(1), for temporary variable.

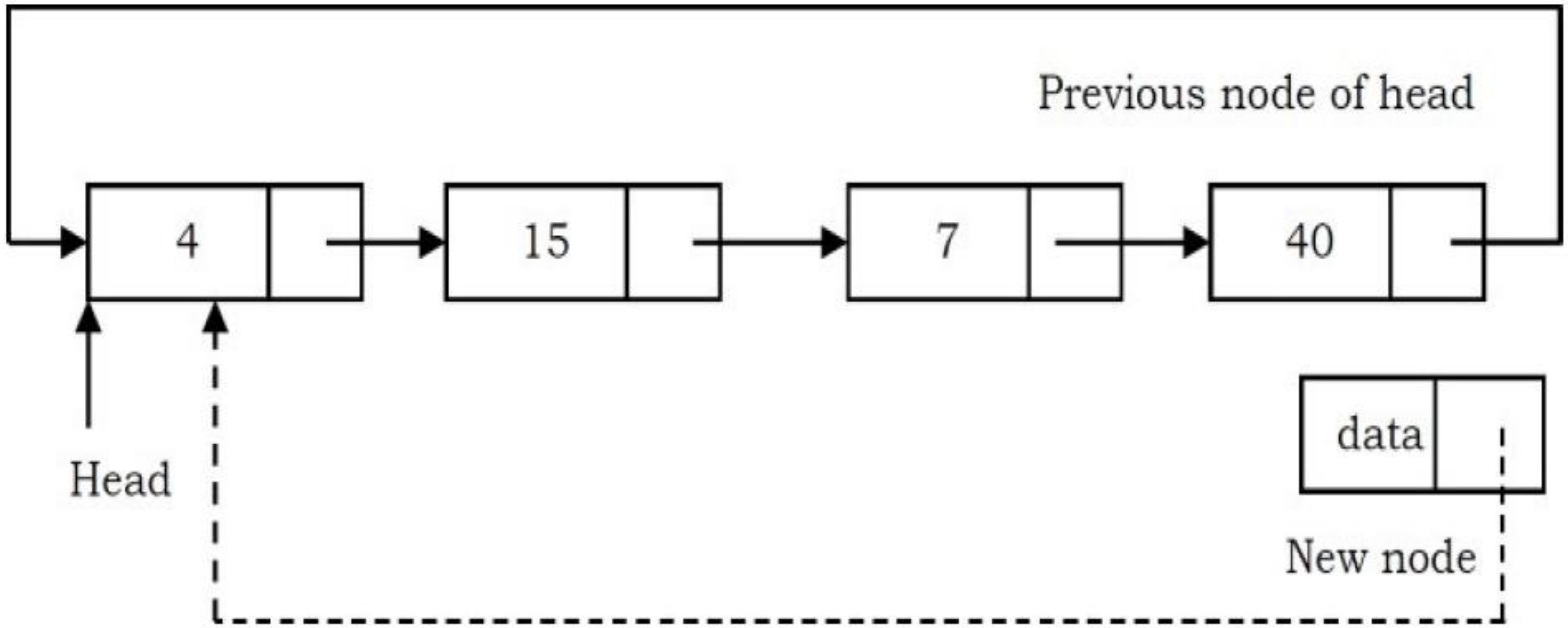
Inserting a Node at the End of a Circular Linked List

- Let us add a node containing *data*, at the end of a list (circular list) headed by *head*.
- The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

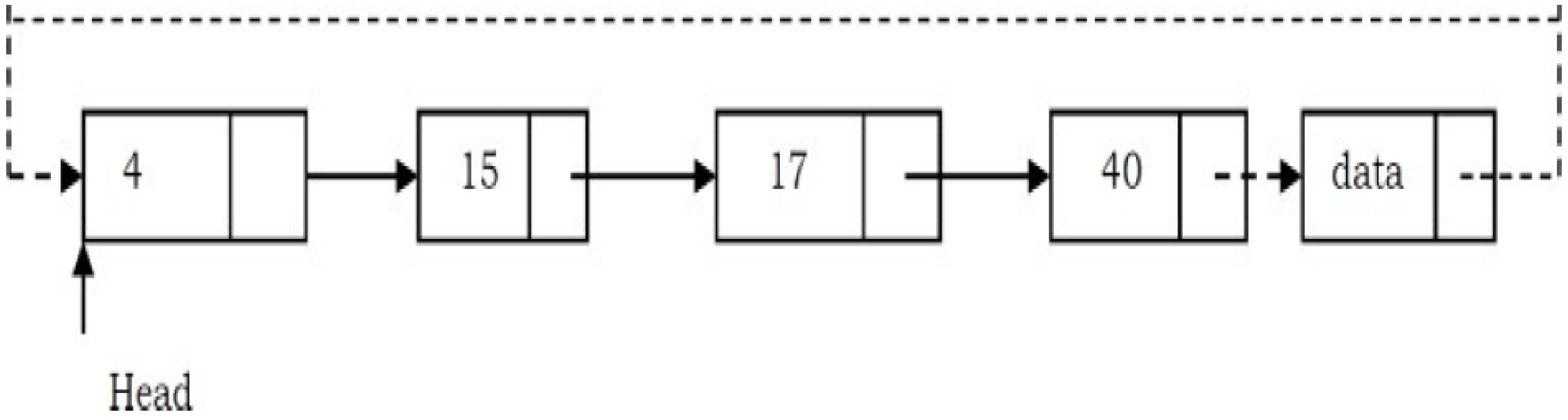
– Create a new node and initially keep its next pointer pointing to itself.



- Update the next pointer of the new node with the head node and also traverse the list to the tail. That means in a circular list we should stop at the node whose next node is head.


Previous node of head
4
15
7
40
Head
data
New node

- Update the next pointer of the previous node to point to the new node and we get the list as shown below.

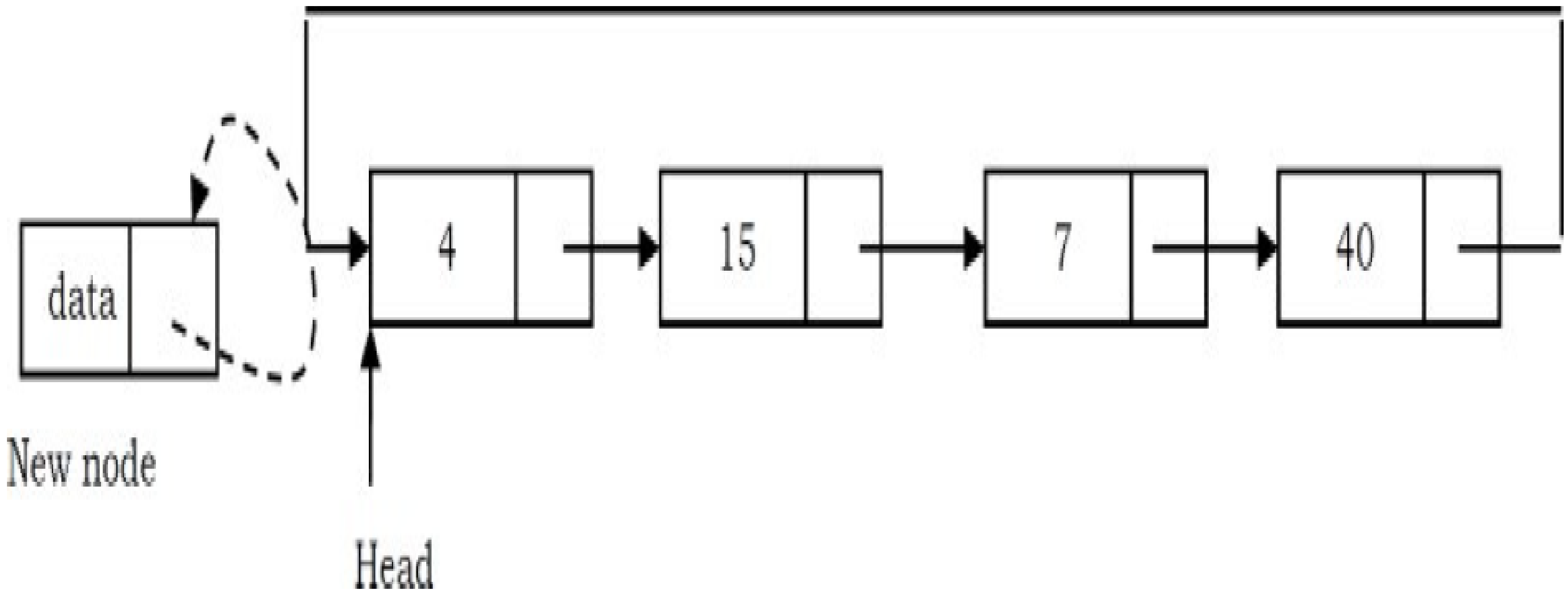


```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
      struct CLLNode *current = *head;
      struct CLLNode *newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
      if(!newNode) {
         printf("Memory Error");
         return;
      }
      newNode→data = data;
      while (current→next != *head)
         current = current→next;

      newNode→next = newNode;

      if(*head ==NULL)
         *head = newNode;
      else {
         newNode→next = *head;
         current→next = newNode;
      }
}
```

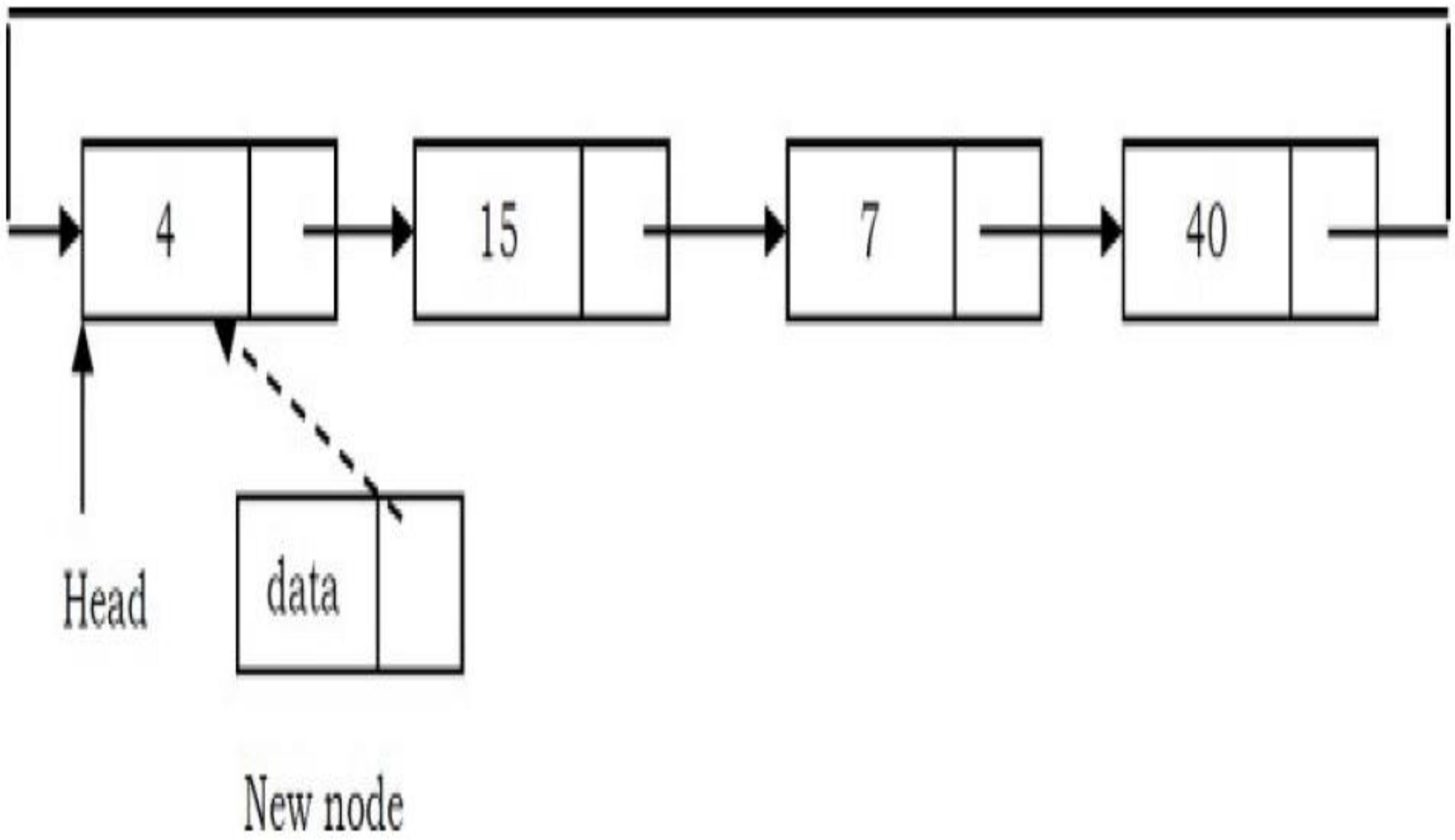
Inserting a Node at the Front of a Circular Linked List

- The only difference between inserting a node at the beginning and at the end is that, after inserting the new node, we just need to update the pointer. The steps for doing this are given below:

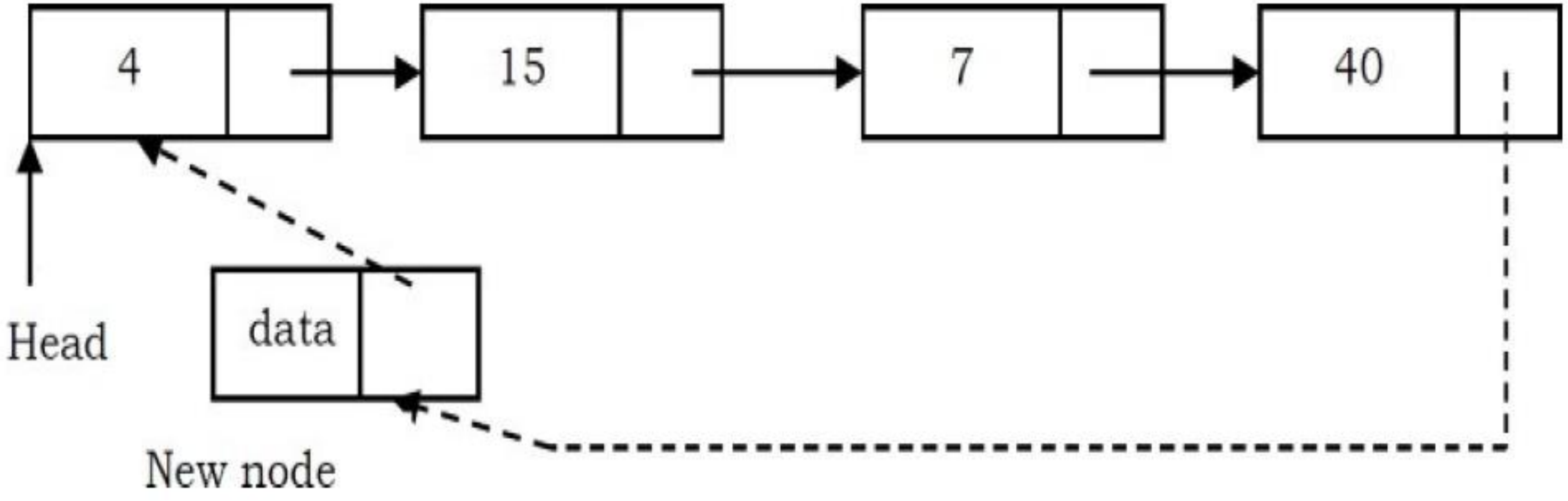
- Create a new node and initially keep its next pointer pointing to itself.



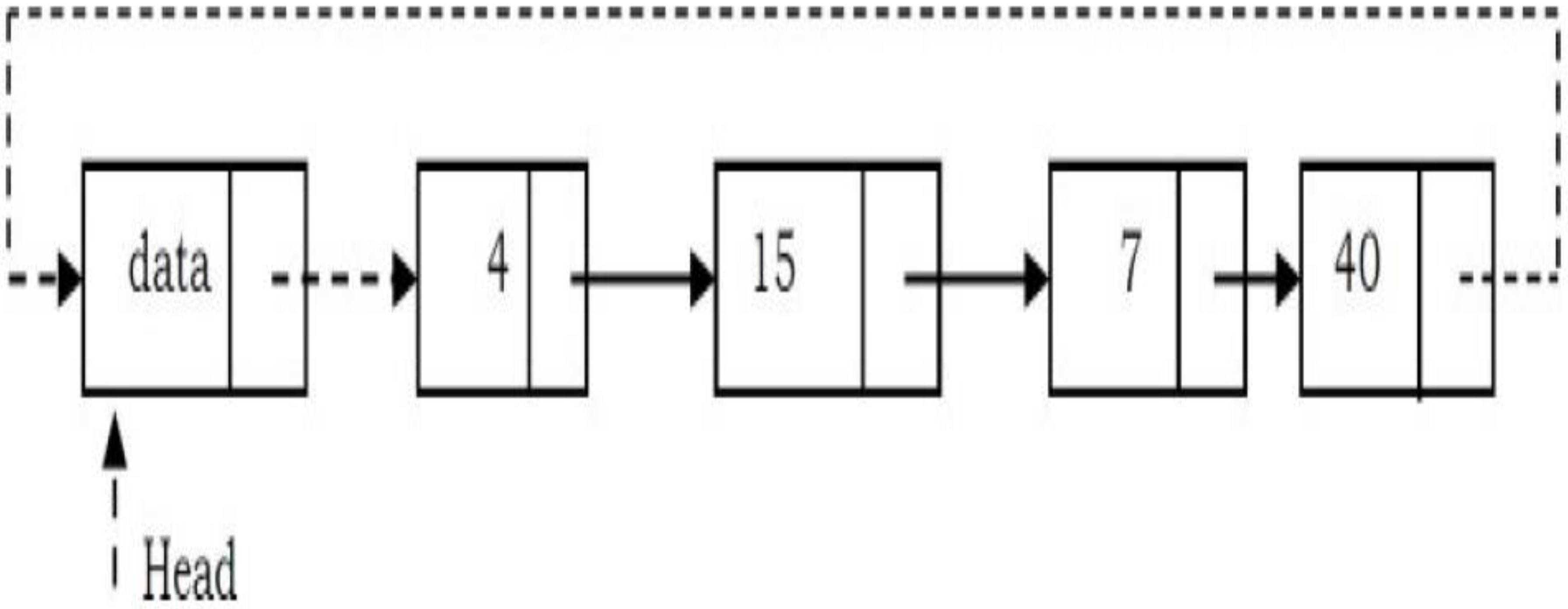
- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.


4
15
7
40
Head
data
New node

- Update the previous head node in the list to point to the new node.



- Make the new node as the head.



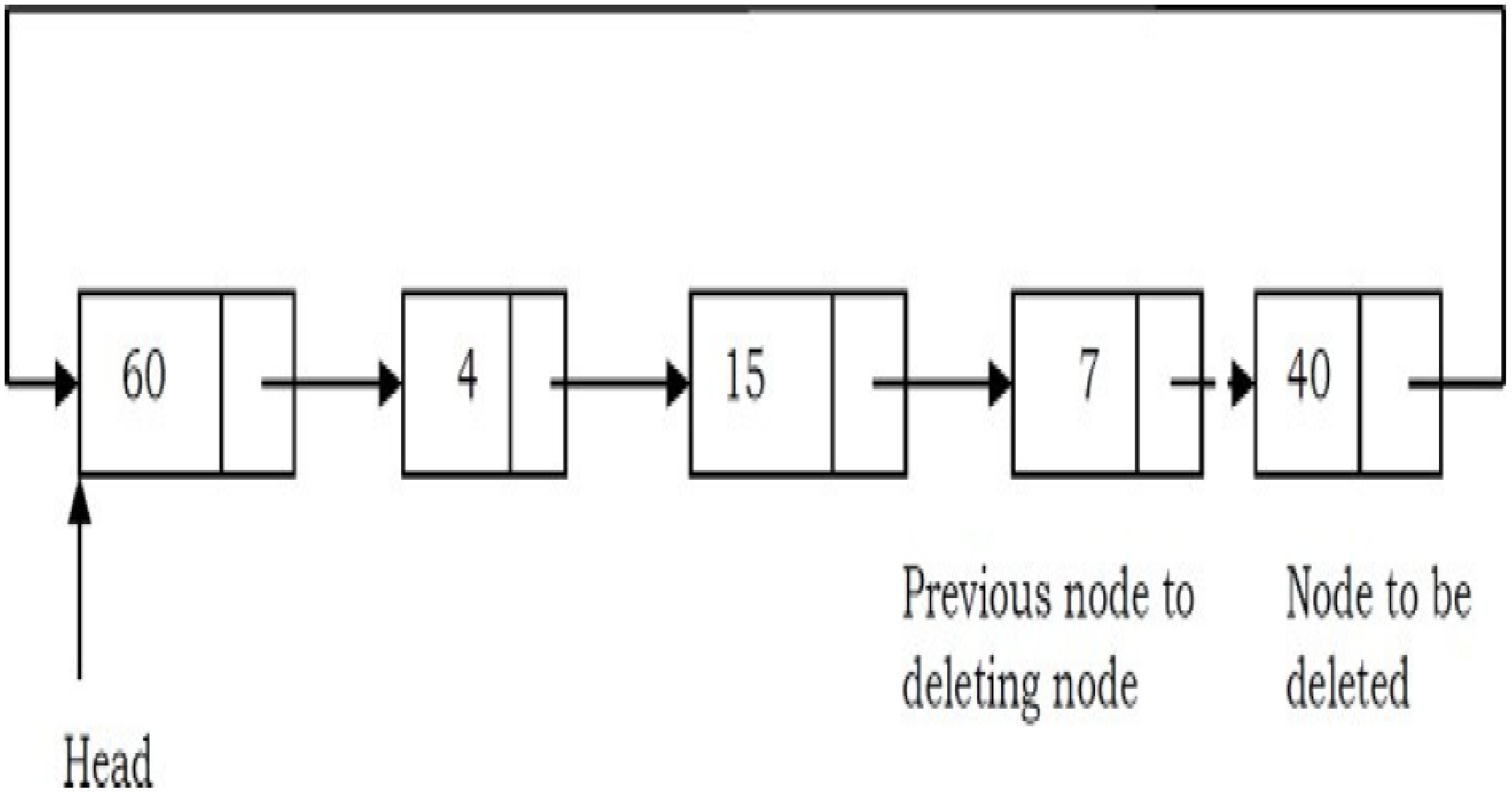
```
void InsertAtBeginInCLL (struct CLLNode **head, int data) {
	struct CLLNode *current = *head;
	struct CLLNode * newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
	if(!newNode) {
	   printf("Memory Error");
	   return;
	}
	newNode→data = data;
	while (current→next != *head)
	   current = current→next;
	newNode→next = newNode;
	if(*head ==NULL)
	   *head = newNode;
	else {
	   newNode→next = *head;
	   current→next = newNode;
	   *head = newNode;
	}
	return;
}
```

- Time Complexity: O(*n*), for scanning the complete list of size *n.*
- Space Complexity: O(1), for temporary variable.

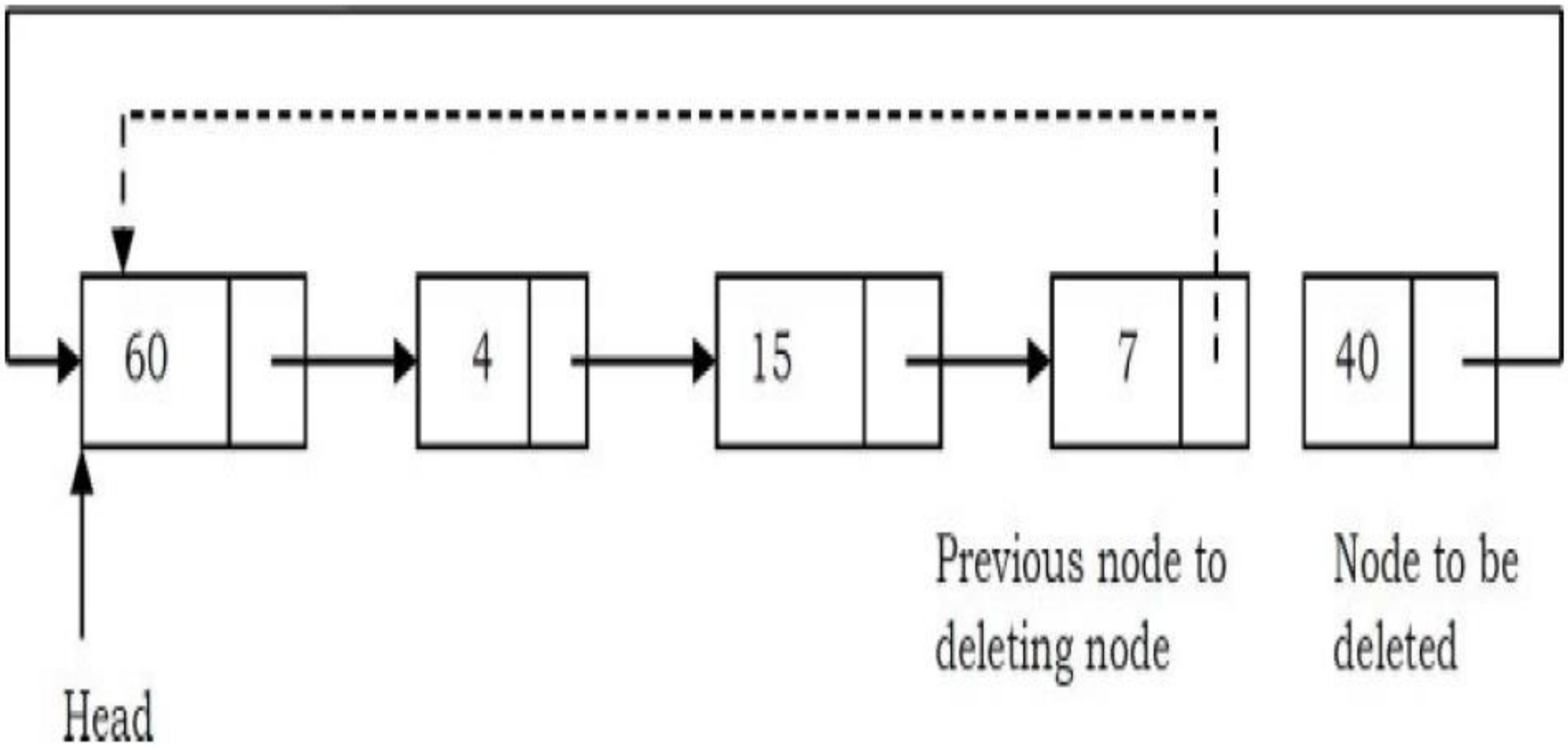
Deleting the Last Node in a Circular Linked List

- The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list.
- To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed *pTail*

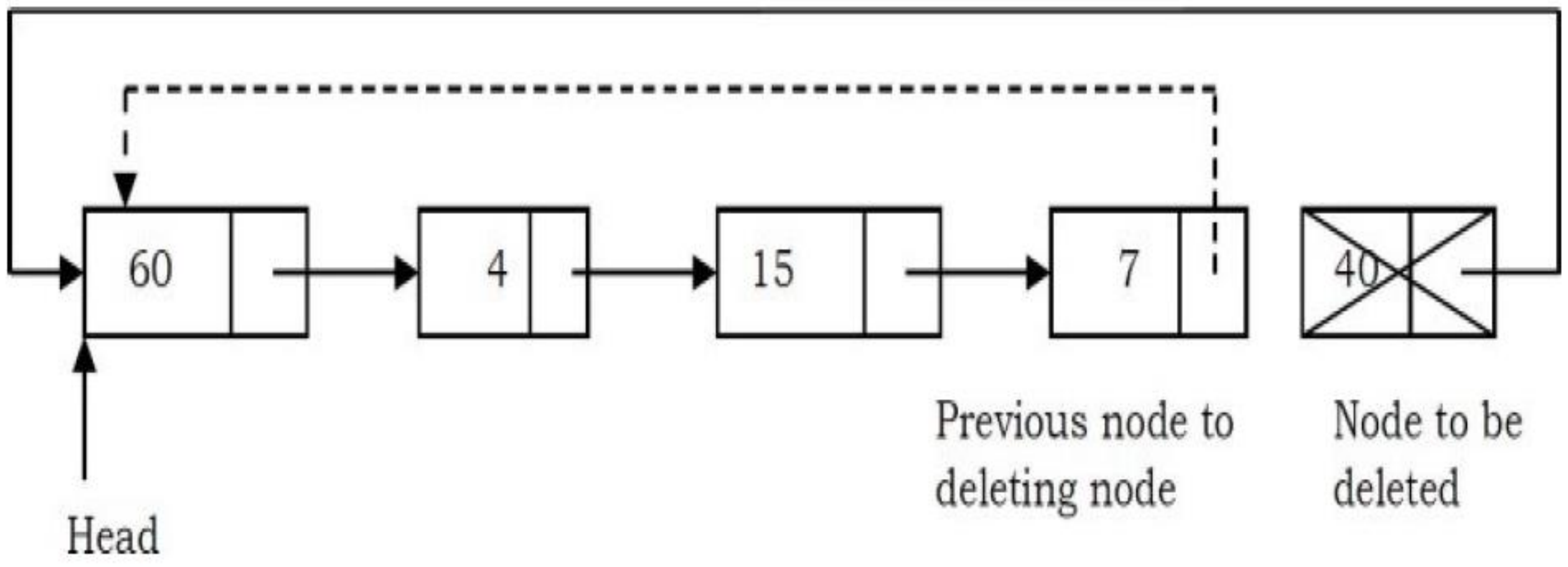
- Traverse the list and find the tail node and its previous node.



- Update the next pointer of tail node's previous node to point to head.



- Dispose of the tail node.



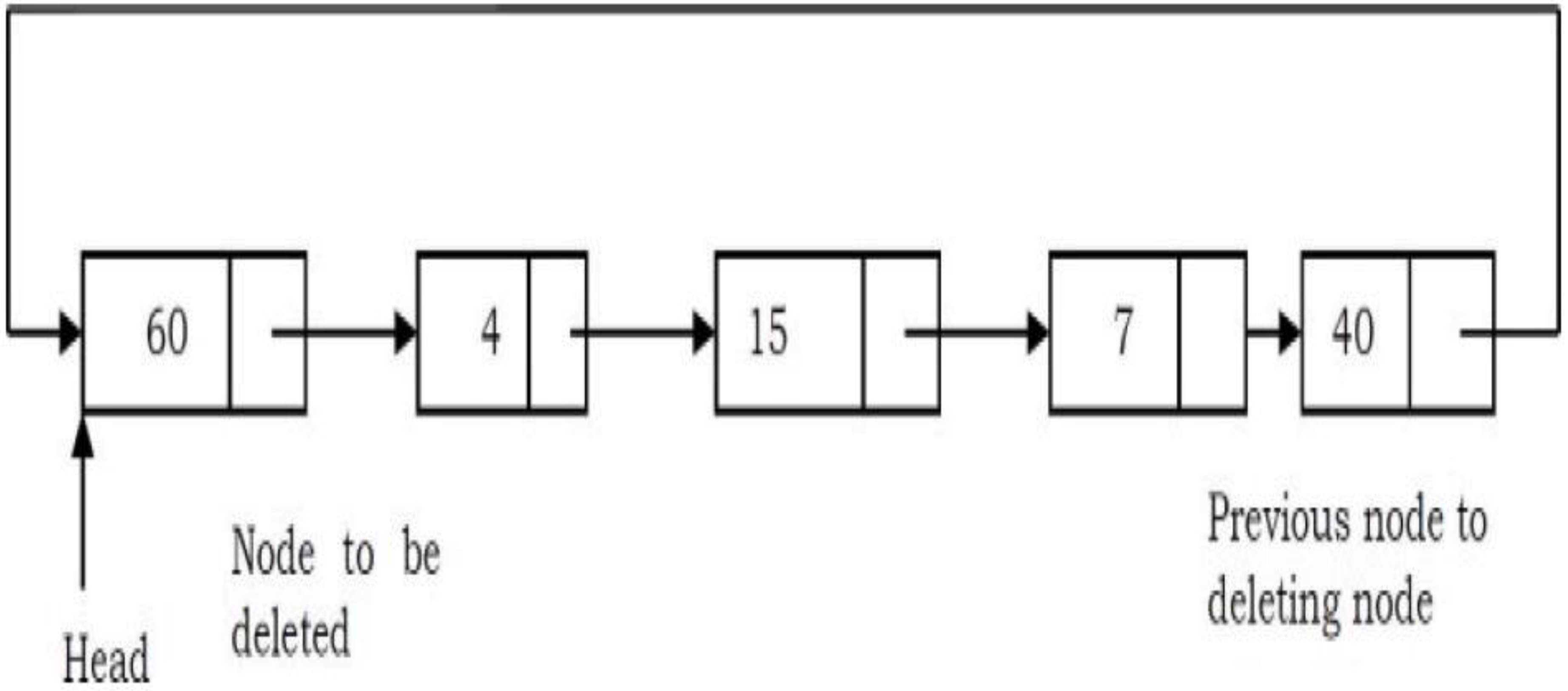
```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
    struct CLLNode *temp = *head, *current = *head;
    if(*head == NULL) {
        printf( "List Empty"); return;
    }
    while (current→next != *head) {
        temp = current;
        current = current→next;
    }
    temp→next = current→next;
    free(current);
    return;
}
```

- Time Complexity: O(*n*), for scanning the complete list of size *n*.
- Space Complexity: O(1), for a temporary variable.

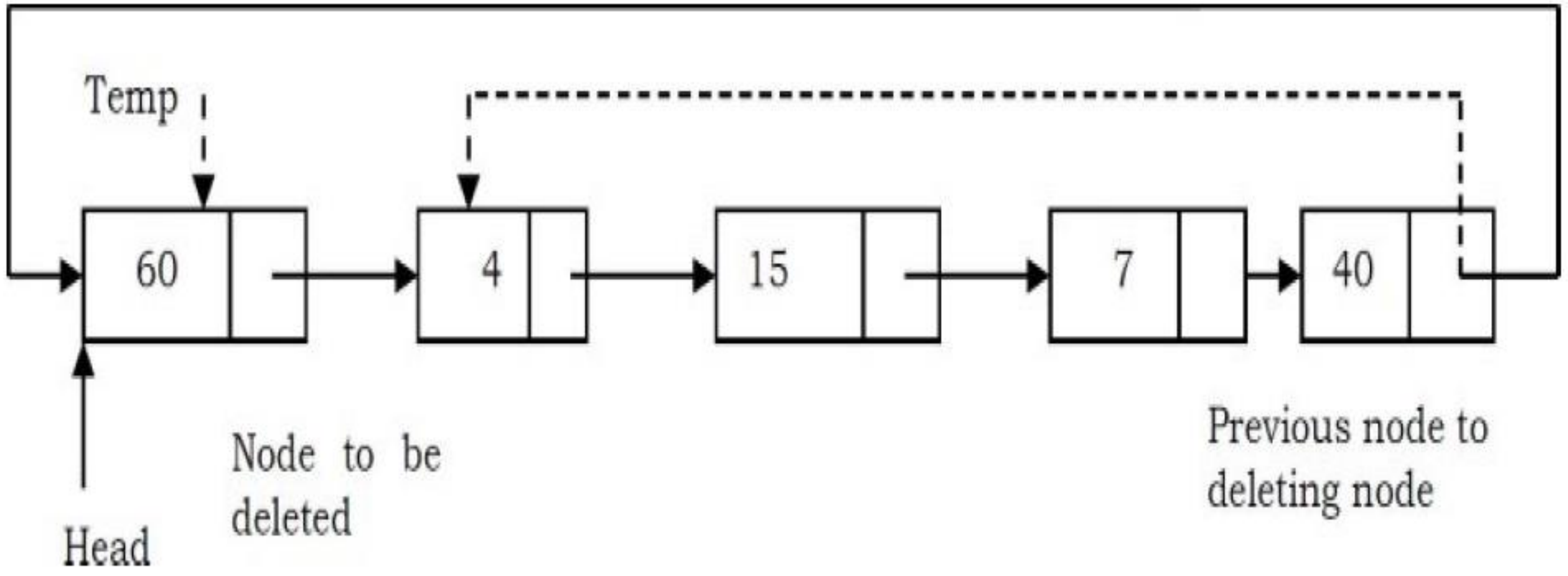
Deleting the First Node in a Circular List

- The first node can be deleted by simply replacing the next field of the tail node with the next field of the first node.

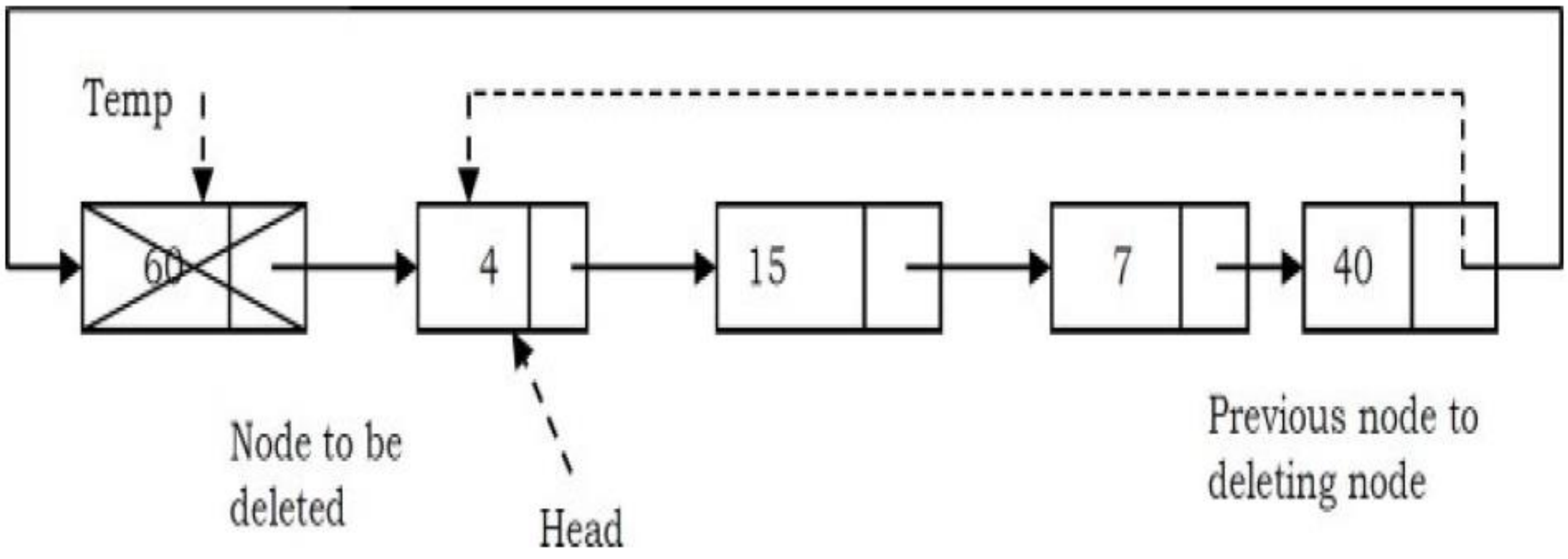
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
      struct CLLNode *temp = *head;
      struct CLLNode *current = *head;
      if(*head == NULL) {
         printf(“List Empty”);
         return;
      }
      while (current→next != *head)
         current = current→next;
      current→ next = *head→next;
      *head = *head→next;
      free(temp);
      return;
}
```

Time Complexity and Applications

- Time Complexity: O(*n*), for scanning the complete list of size *n.*
- Space Complexity: O(1), for a temporary variable.
- **Applications of Circular List**
  - Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.