

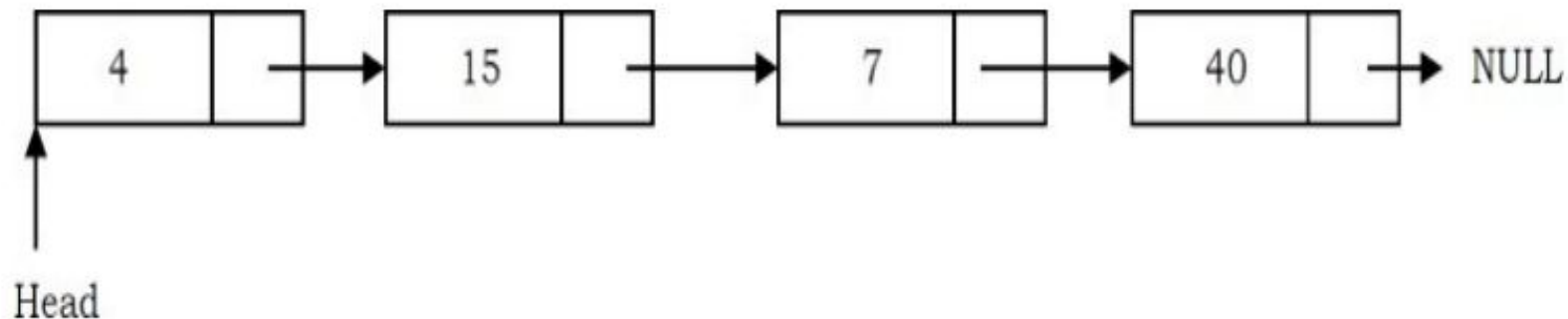


Estructuras Lineales

Prof.: MSc. Ashey John Masi Noblega

Linked List

- Successive elements are connected by pointer
- 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.



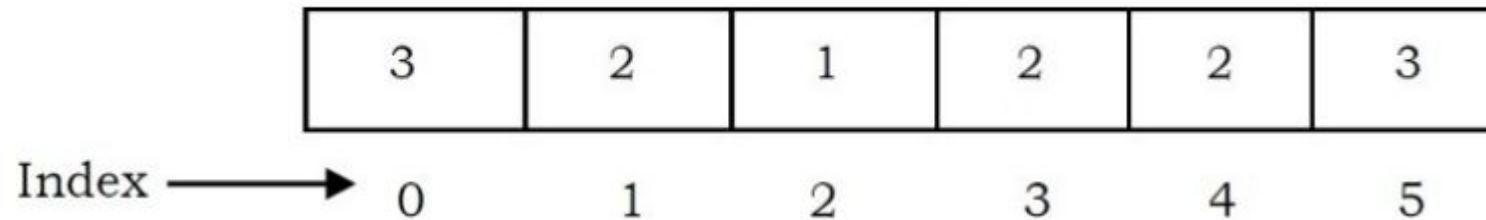


Operations:

- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list
- Delete List: removes all elements of the list (disposes the list) (**Auxiliar**)
- Count: returns the number of elements in the list (**Auxiliar**)
- Find n th node from the end of the list (**Auxiliar**)

Arrays vs. Linked Lists: Arrays

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.





Arrays vs. Linked Lists: Arrays

Advantages of Arrays

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

Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **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).
- **Complex position-based insertion:** To insert an element at a given position, we may need to shift the existing elements.



Arrays vs. Linked Lists: Linked List

Advantages:

- The advantage of linked lists is that they can be expanded in constant time. 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.

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.

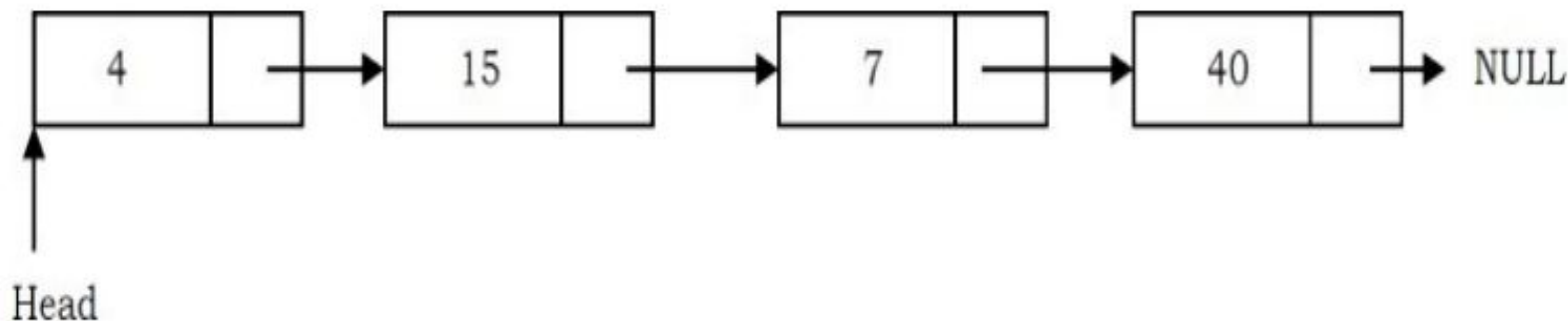


Arrays vs. Linked Lists:

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)	$O(n)$

Singly Linked Lists

- Generally “linked list” means a singly linked list.
- This list consists of a number of nodes in which 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.



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




Operation: Traversing

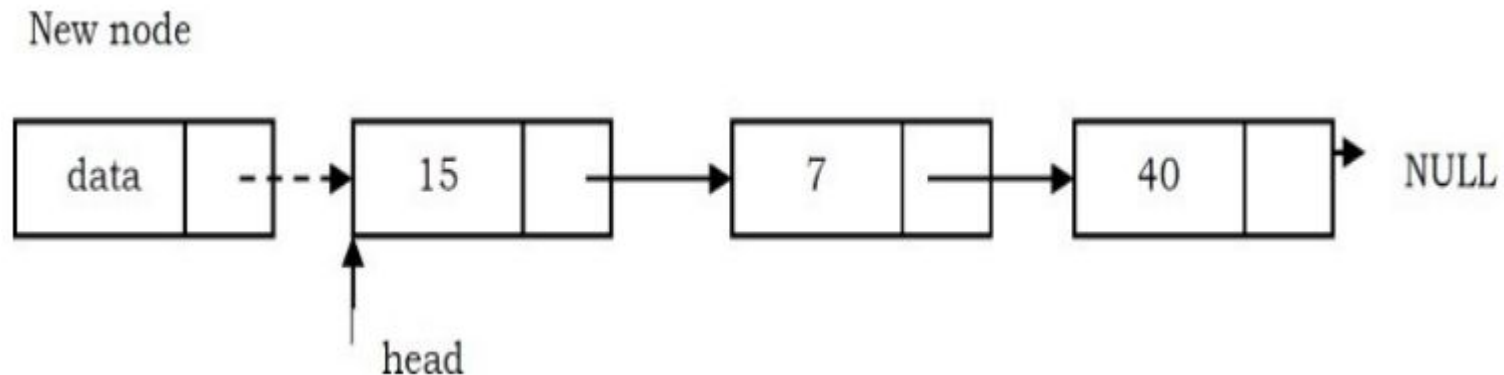
Let us assume that the head points to the first node of the list.

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.

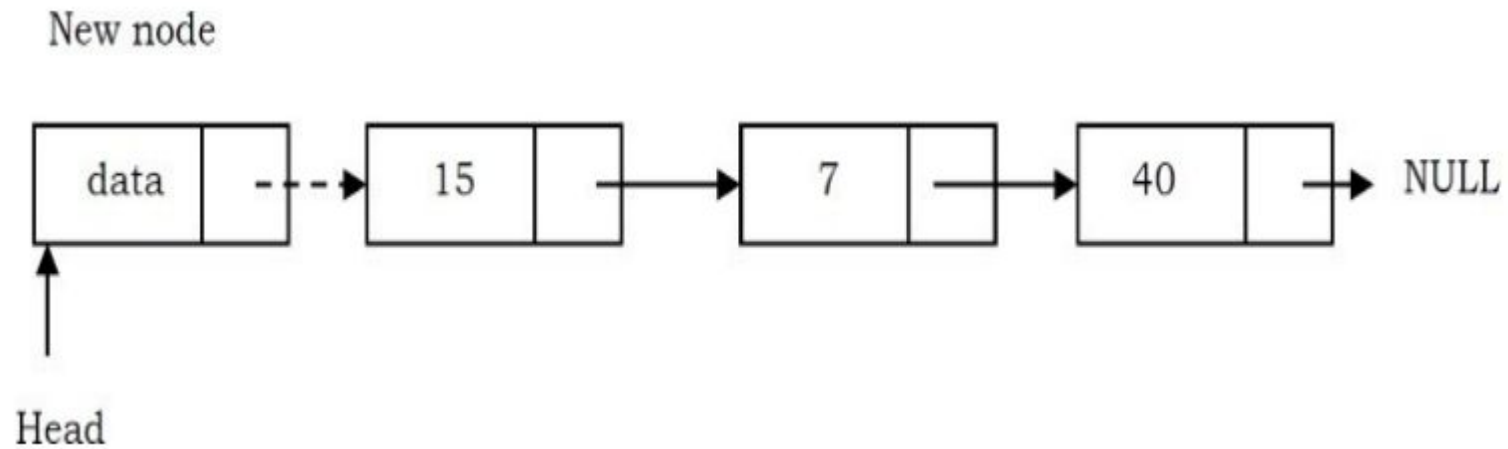
```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL) {
    printf("%d --->", temp->data);
    temp = temp->next;
}
```

Operation: Insertion - Beginning

- Allocate memory for new node
- Store data
- Change next of new node to point to head
- Change head to point to recently created node



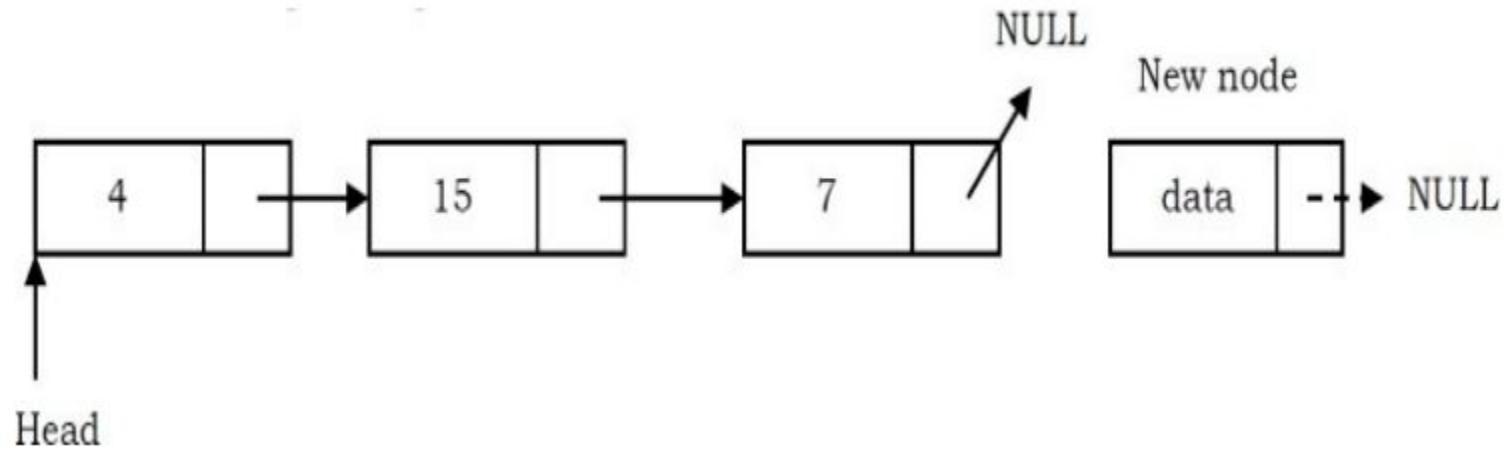
Operation: Insertion - Beginning



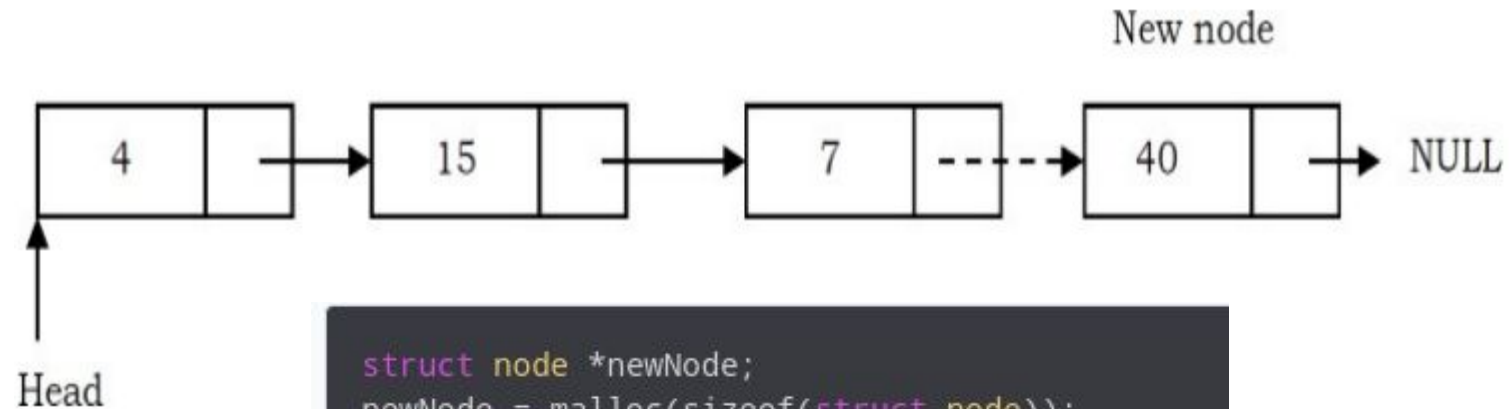
```
struct node *newNode;  
newNode = malloc(sizeof(struct node));  
newNode->data = 4;  
newNode->next = head;  
head = newNode;
```

Operation: Insertion - End

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node



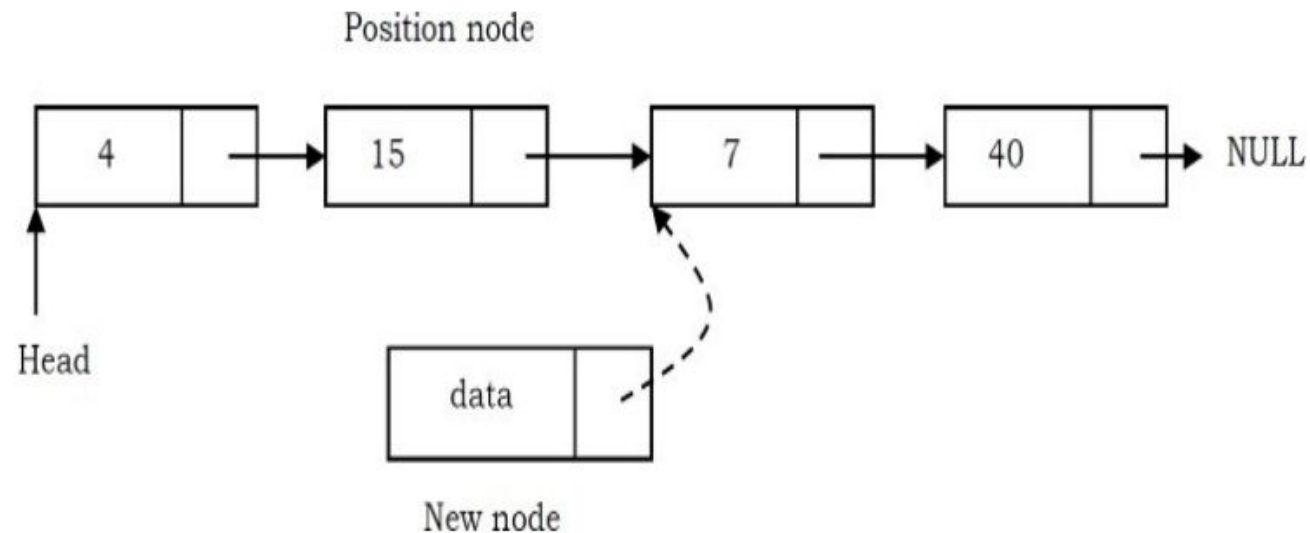
Operation: Insertion - End



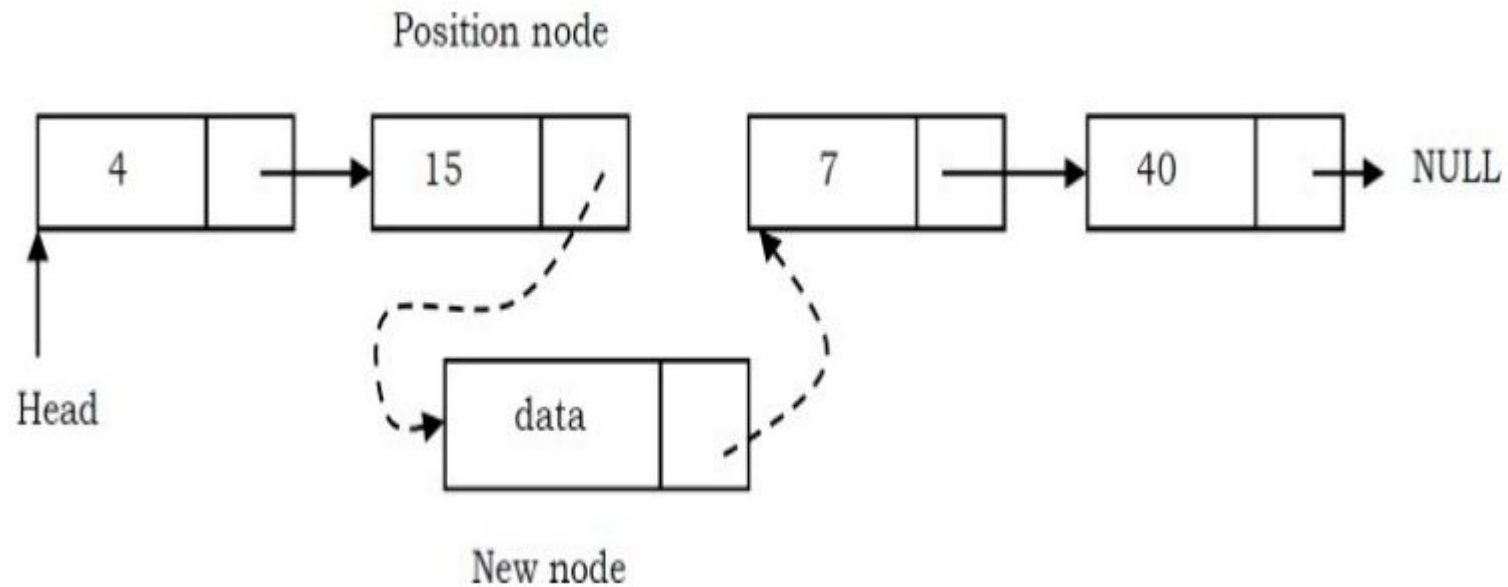
```
struct node *newNode;  
newNode = malloc(sizeof(struct node));  
newNode->data = 4;  
newNode->next = NULL;  
  
struct node *temp = head;  
while(temp->next != NULL){  
    temp = temp->next;  
}  
  
temp->next = newNode;
```

Operation: Insertion - Middle

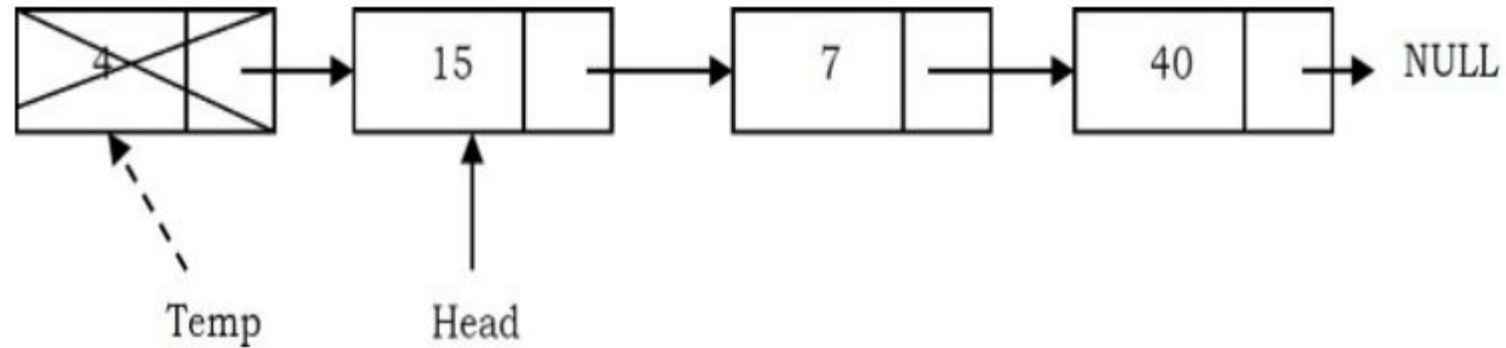
- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between



Operation: Insertion - Middle



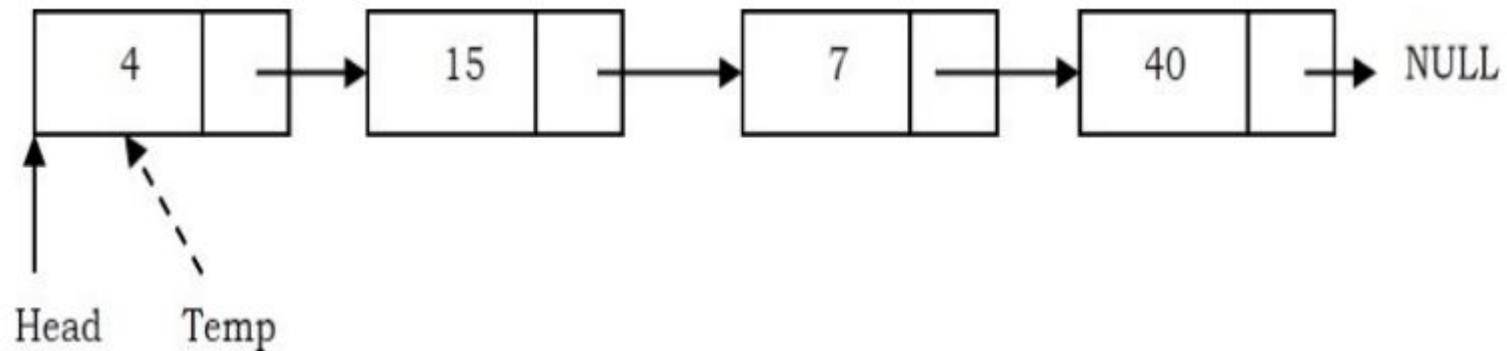
Operation: Delete - Beginning



```
head = head->next;
```

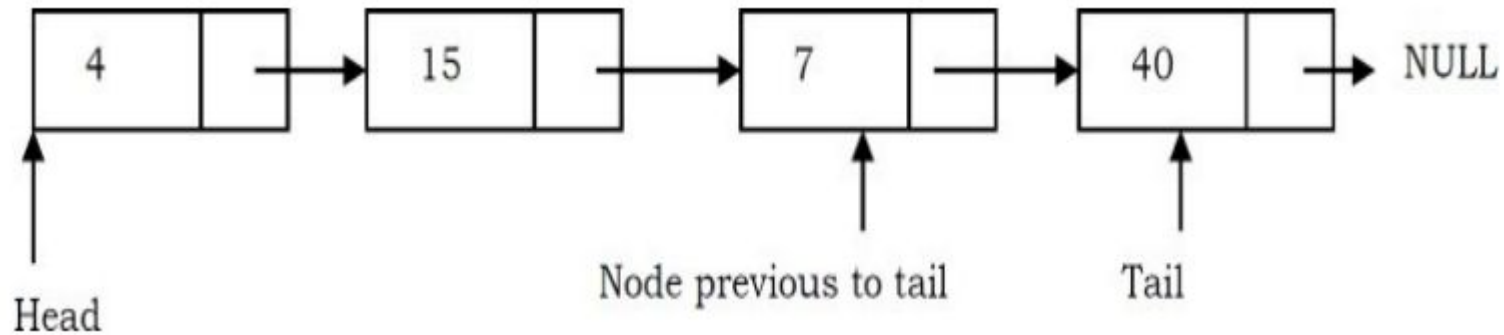

Operation: Delete - Beginning

- Point head to the second node

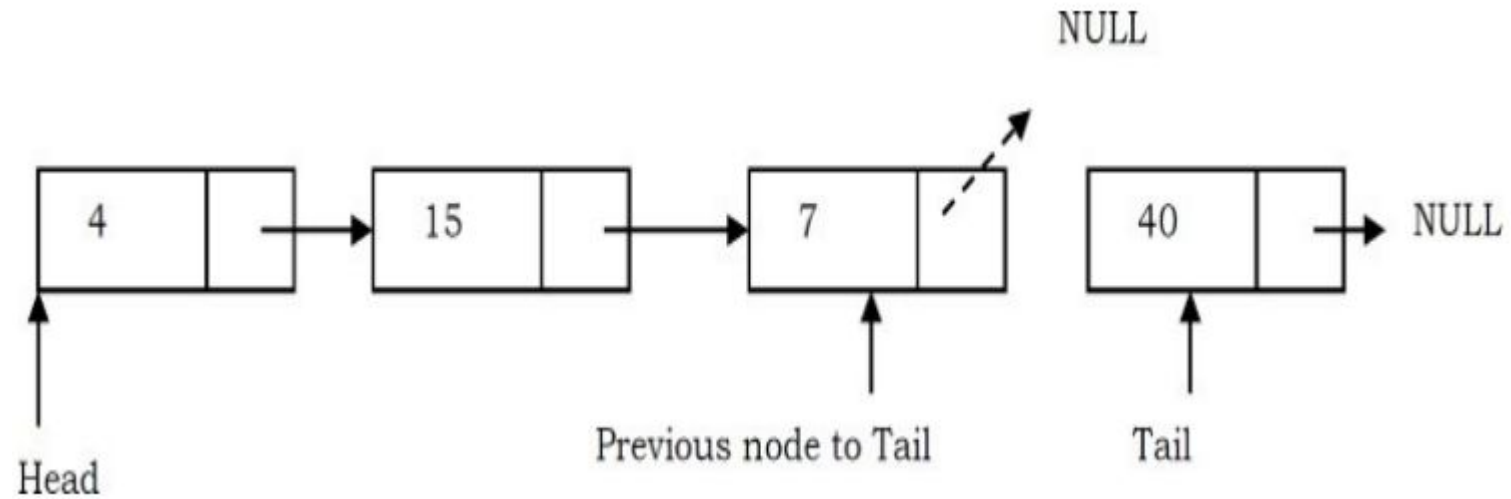


Operation: Delete - End

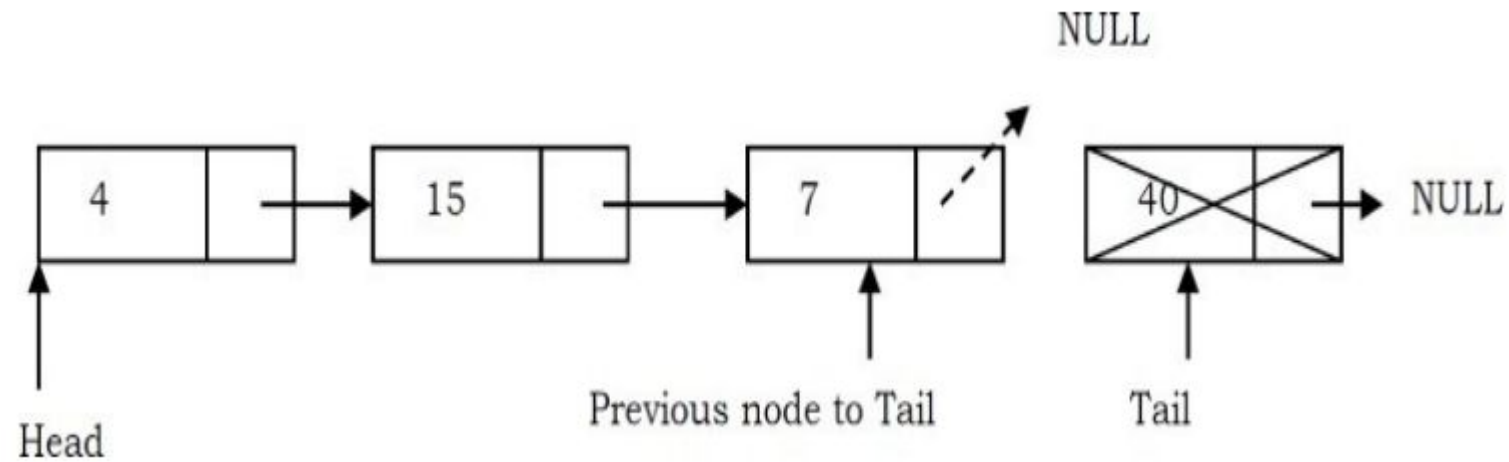
- Traverse to second last element
- Change its next pointer to null



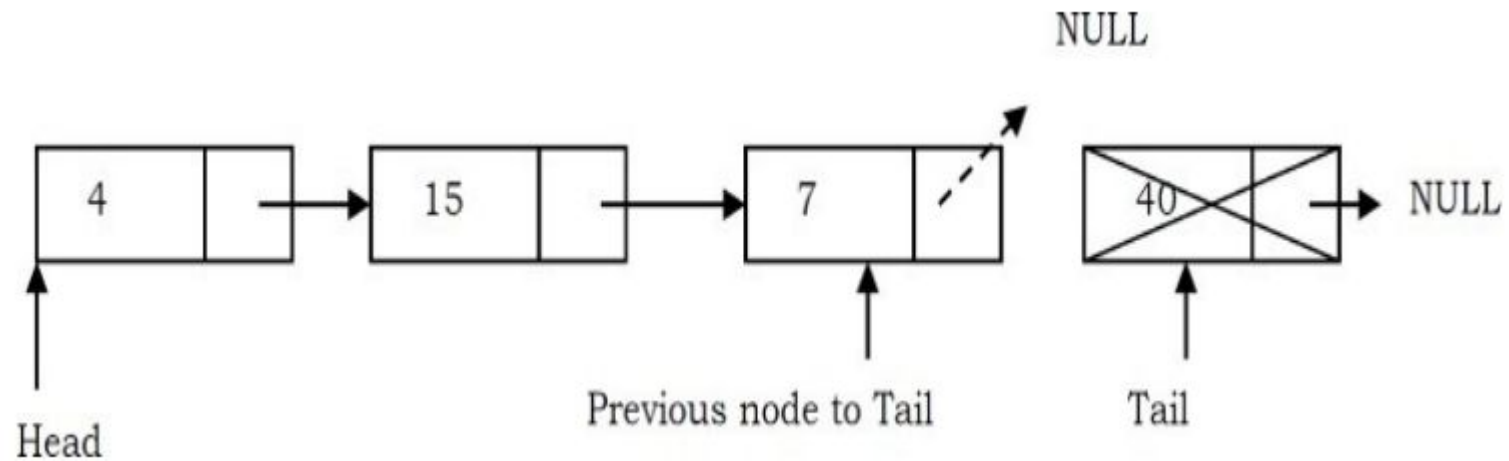
Operation: Delete - End



Operation: Delete - End

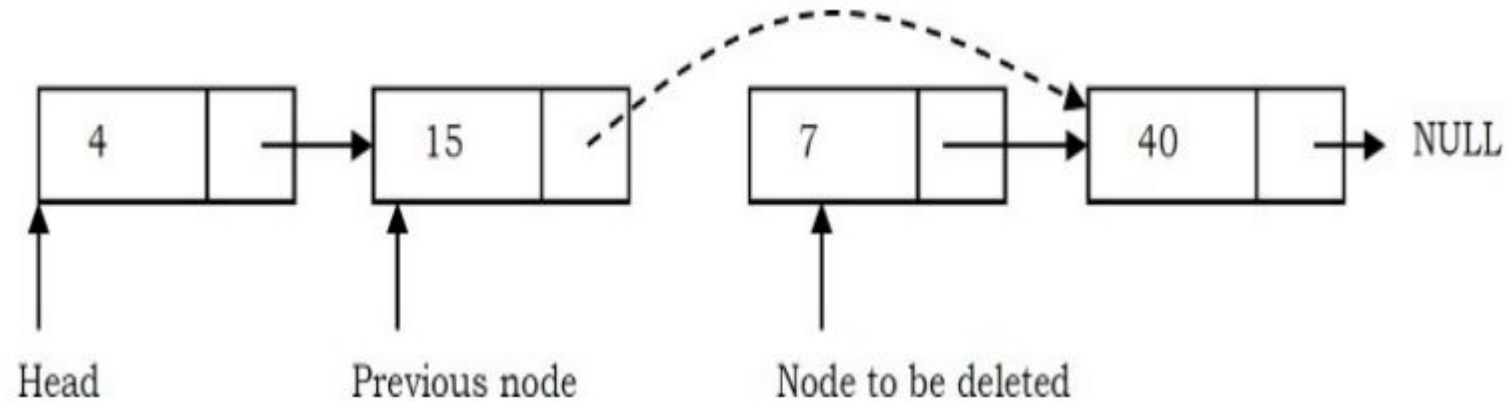


Operation: Delete - End

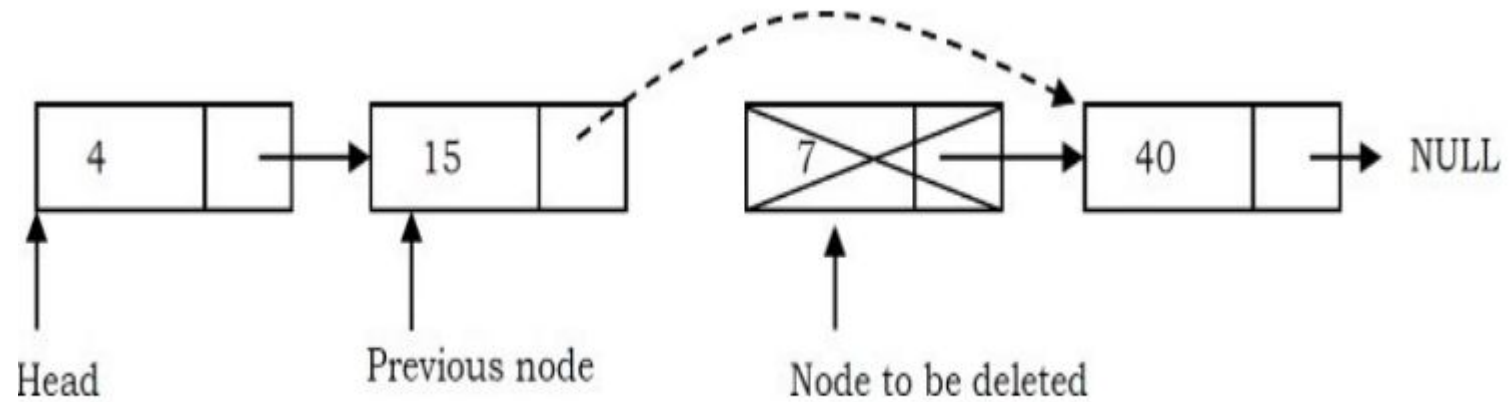


Operation: Delete - Middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

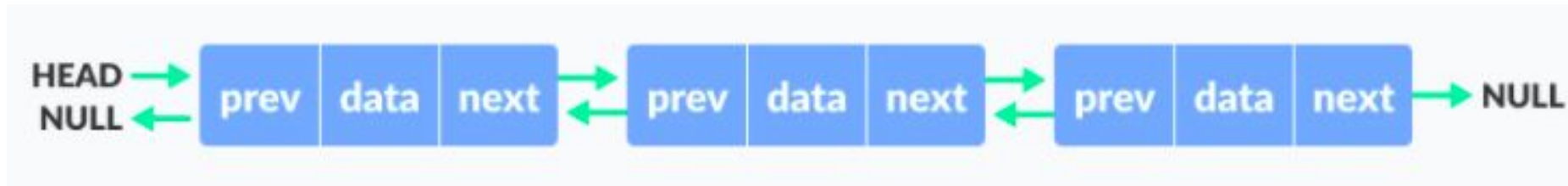


Operation: Delete - Middle



Doubly Linked 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 B, but in a doubly linked list, we can delete a node even if we don't have the previous node's address.
- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations).





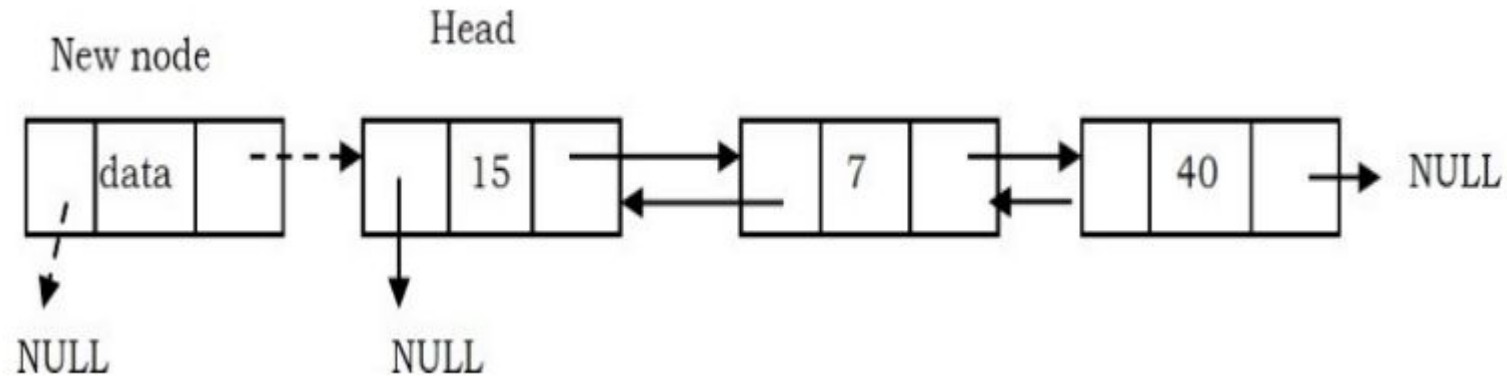
Doubly Linked List

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

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

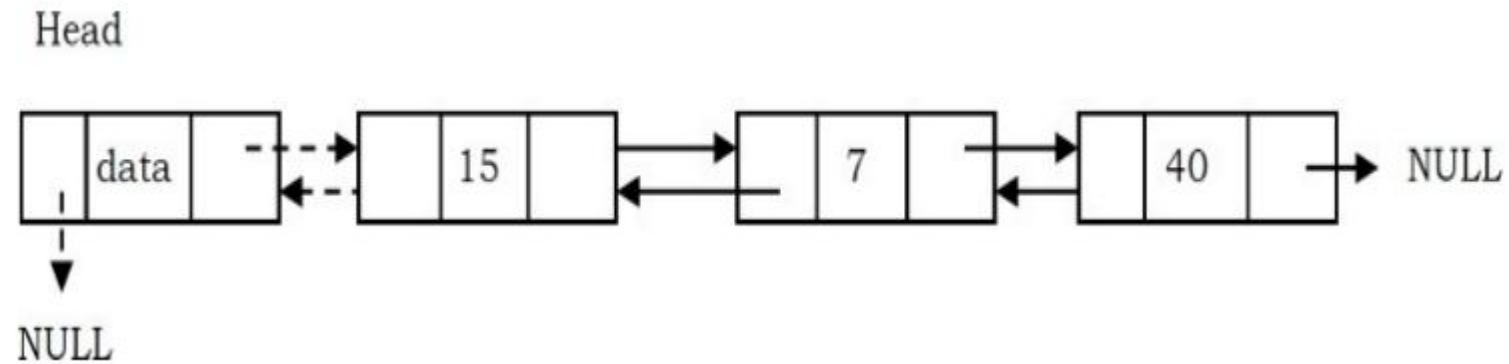
Operations: Insertion - Beginning

- Update the right pointer of the new node to point to the current head node and also make left pointer of new node as NULL.



Operations: Insertion - Beginning

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





Operations: Insertion - Beginning

```
void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node();

    new_node->data = new_data;

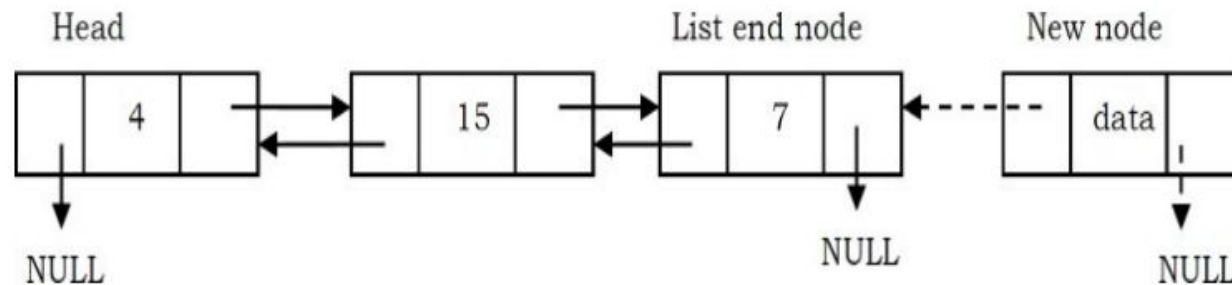
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    (*head_ref) = new_node;
}
```

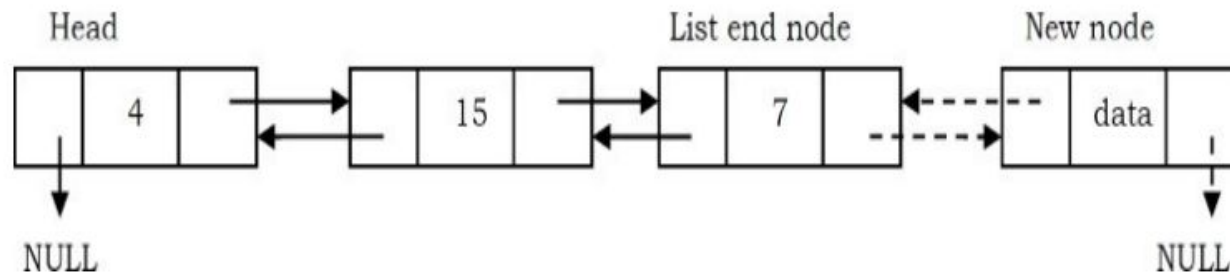
Operations: Insertion - End

- New node right pointer points to NULL and left pointer points to the end of the list.



Operations: Insertion - End

- Update right pointer of last node to point to new node.





Operations: Insertion - End

```
void append(Node** head_ref, int new_data)
{
    Node* new_node = new Node();
    Node* last = *head_ref;

    new_node->data = new_data;
    new_node->next = NULL;

    if (*head_ref == NULL) {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }

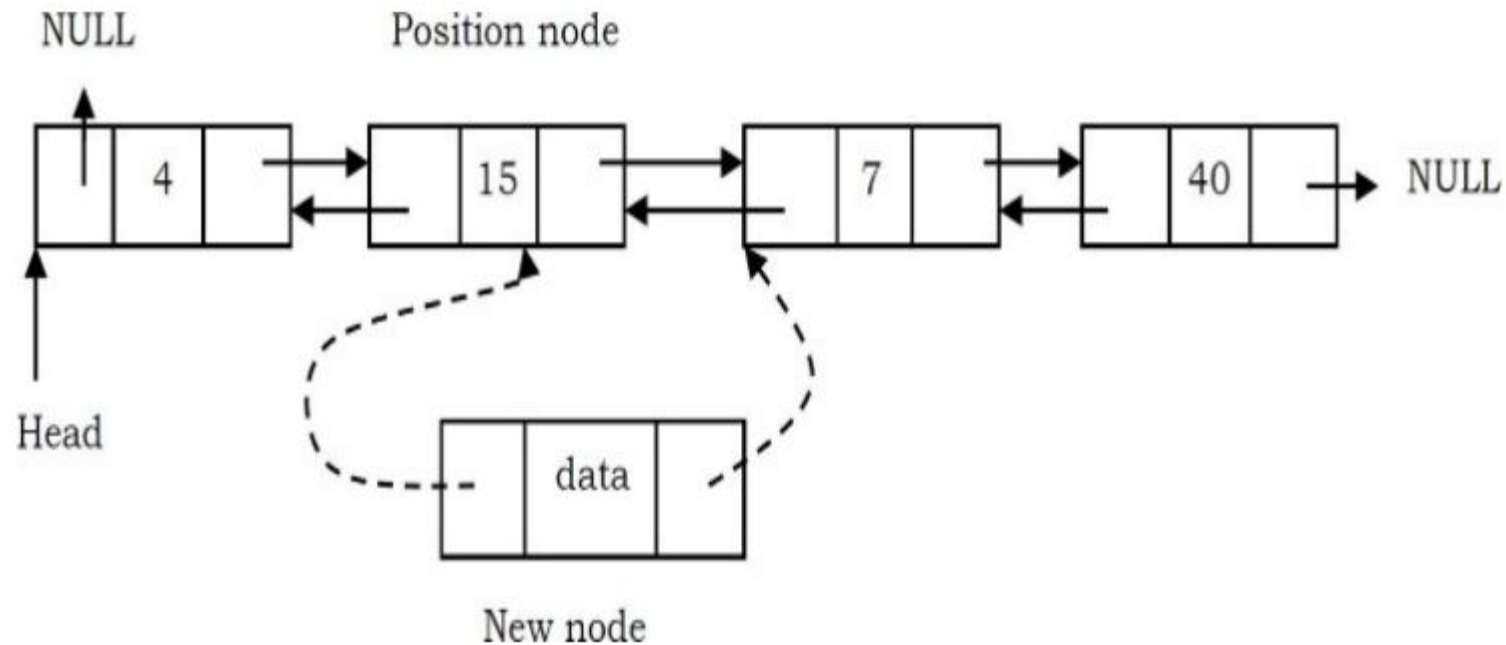
    while (last->next != NULL)
        last = last->next;

    last->next = new_node;
    new_node->prev = last;

    return;
}
```

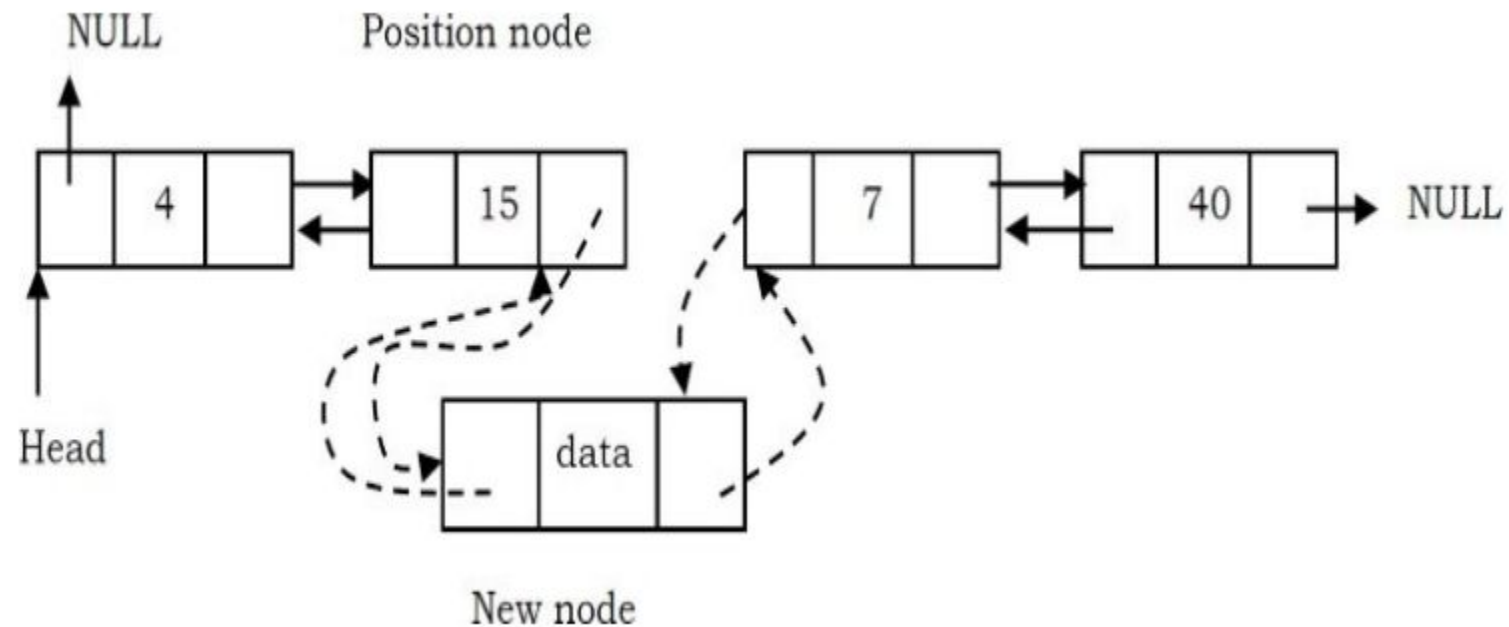
Operations: Insertion - Middle

- New node right pointer points to the next node of the position node where we want to insert the new node. Also, new node left pointer points to the position node.



Operations: Insertion - Middle

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





Operations: Insertion - Middle

```
void insertAfter(Node* prev_node, int new_data)
{
    if (prev_node == NULL) {
        cout << "the given previous node cannot be NULL";
        return;
    }

    Node* new_node = new Node();
    new_node->data = new_data;

    new_node->next = prev_node->next;
    prev_node->next = new_node;
    new_node->prev = prev_node;

    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

```

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;
    }

```

```

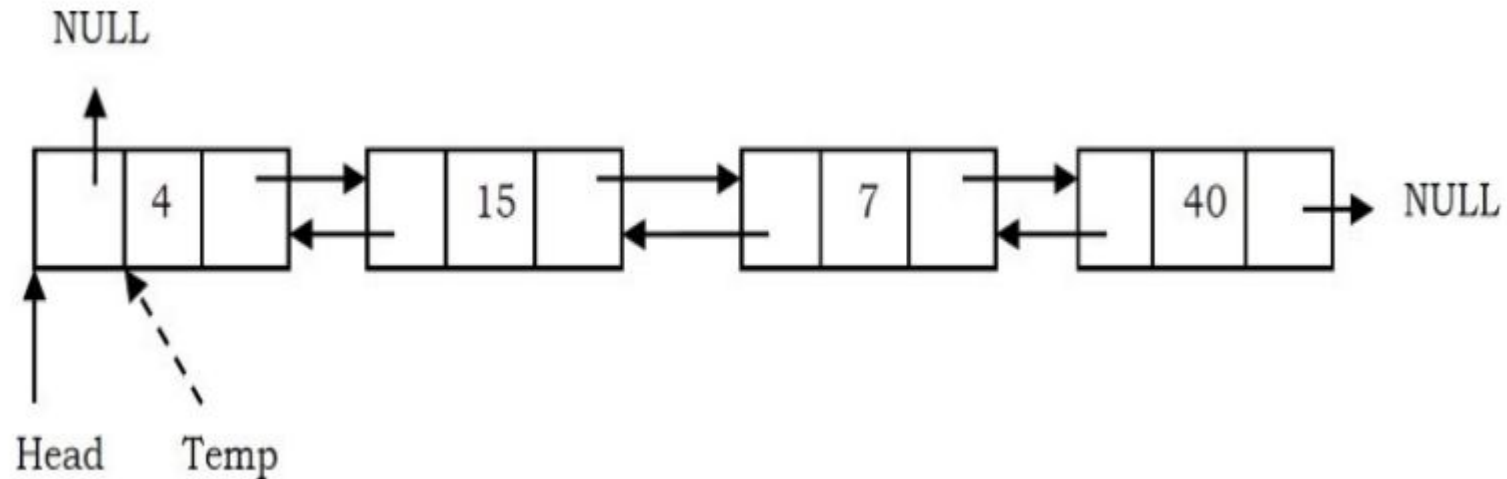
temp = *head;
while ( (k < position - 1) && temp->next!=NULL) {
    temp = temp->next;
    k++;
}
if(k!=position){
    printf("Desired position does not exist\n");
}
newNode->next=temp->next;
newNode->prev=temp;
if(temp->next)
    temp->next->prev=newNode;

temp->next=newNode;
return;
}

```

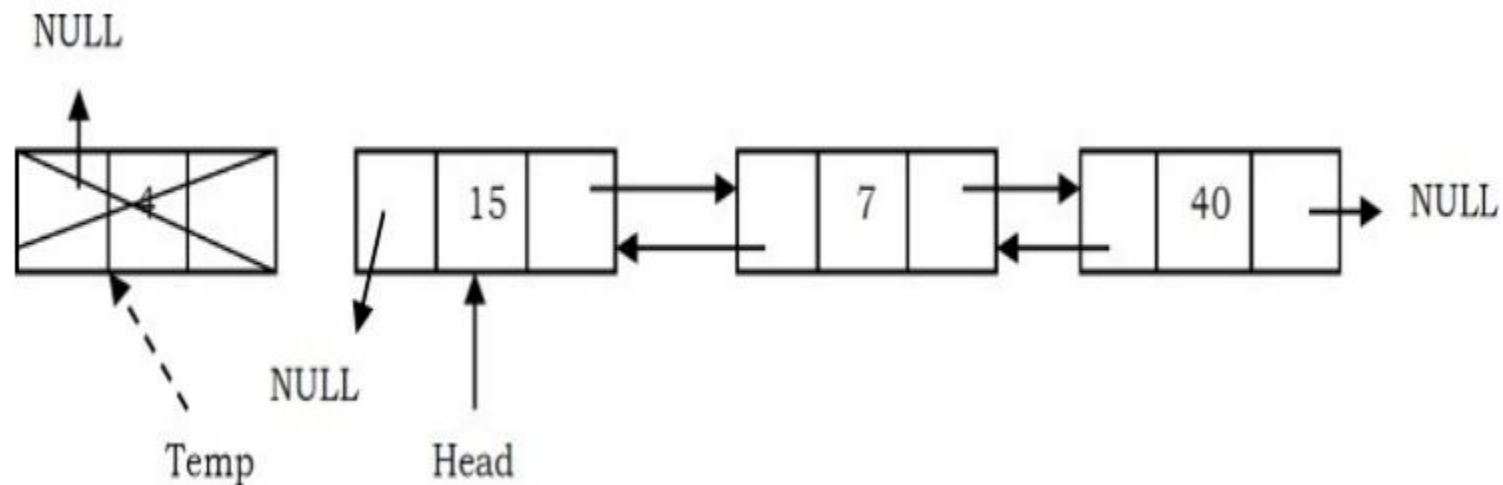
Operations: Delete the First Node

- Create a temporary node which will point to the same node as that of head.



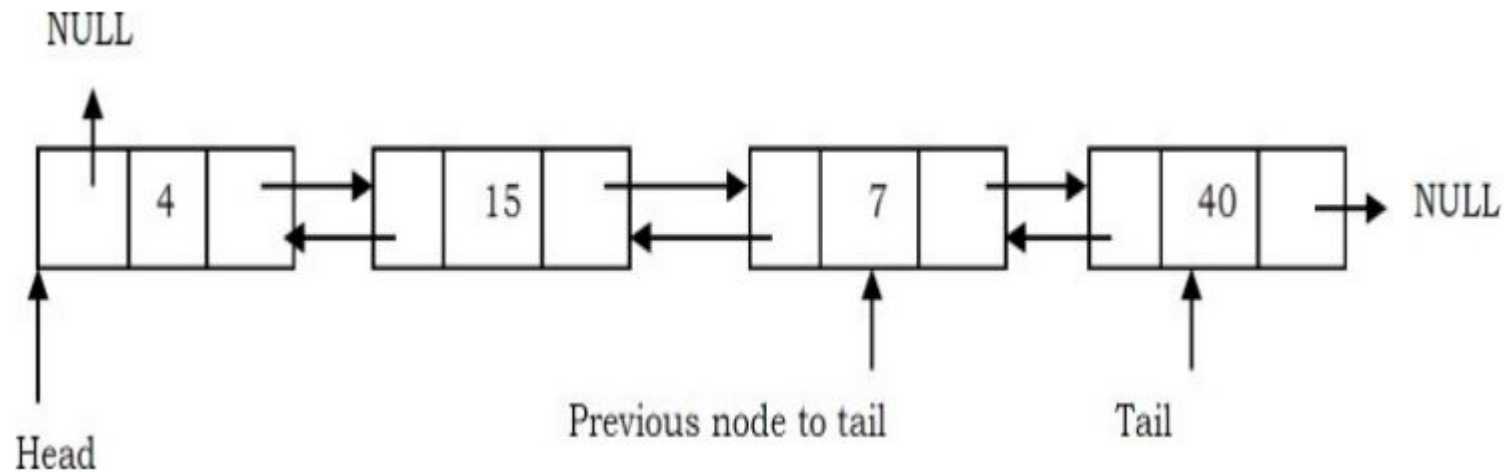
Operations: Delete the First Node

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



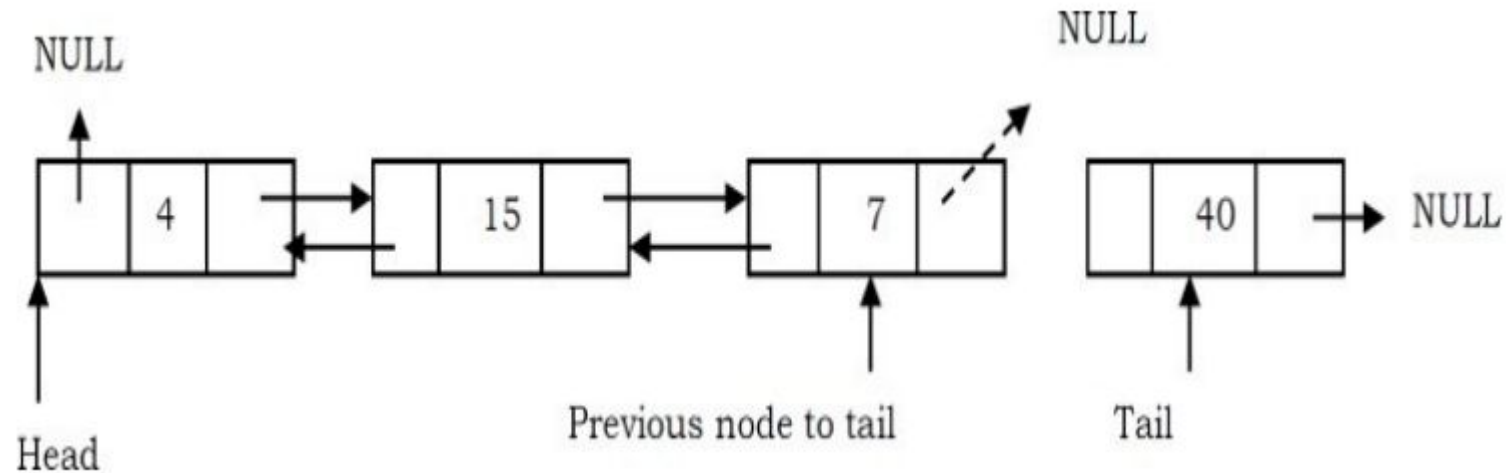
Operations: Delete the Last Node

- 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.



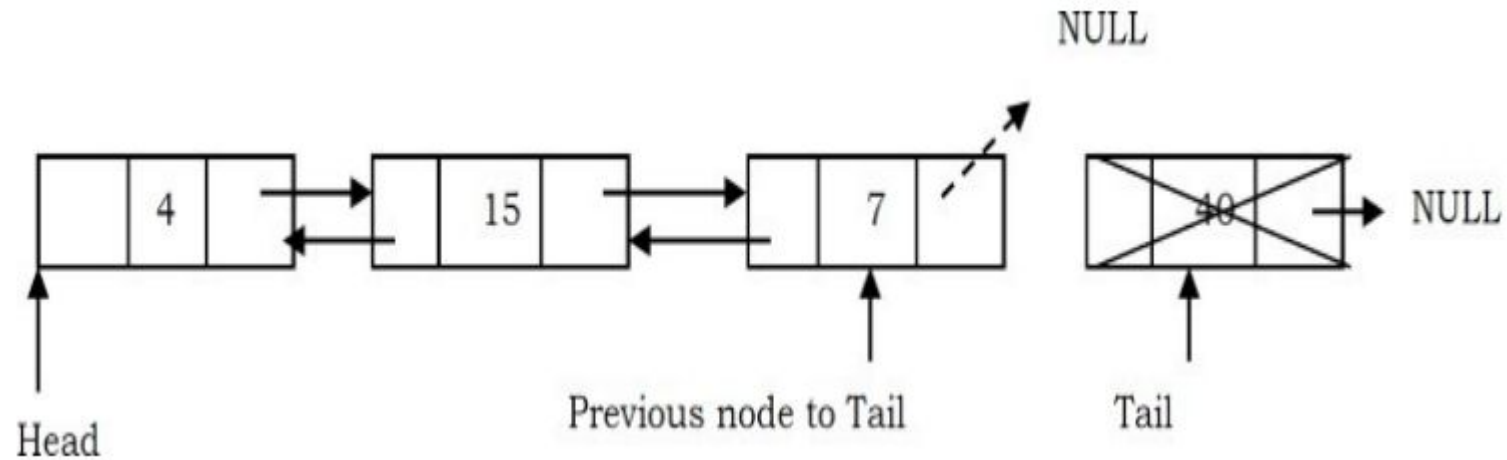
Operations: Delete the Last Node

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



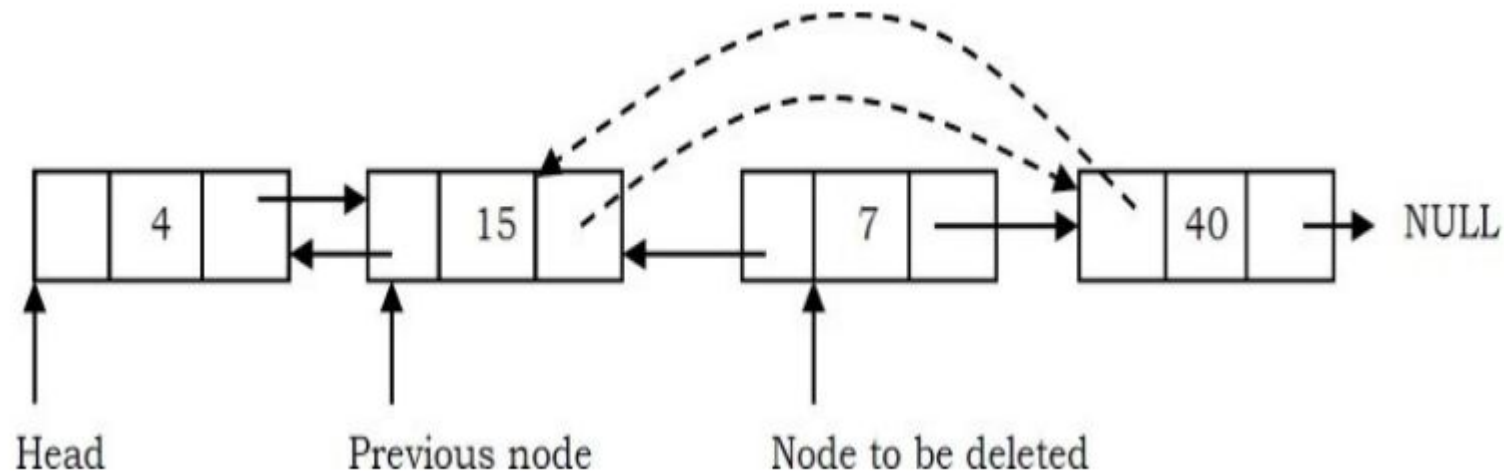
Operations: Delete the Last Node

- Dispose the tail node.



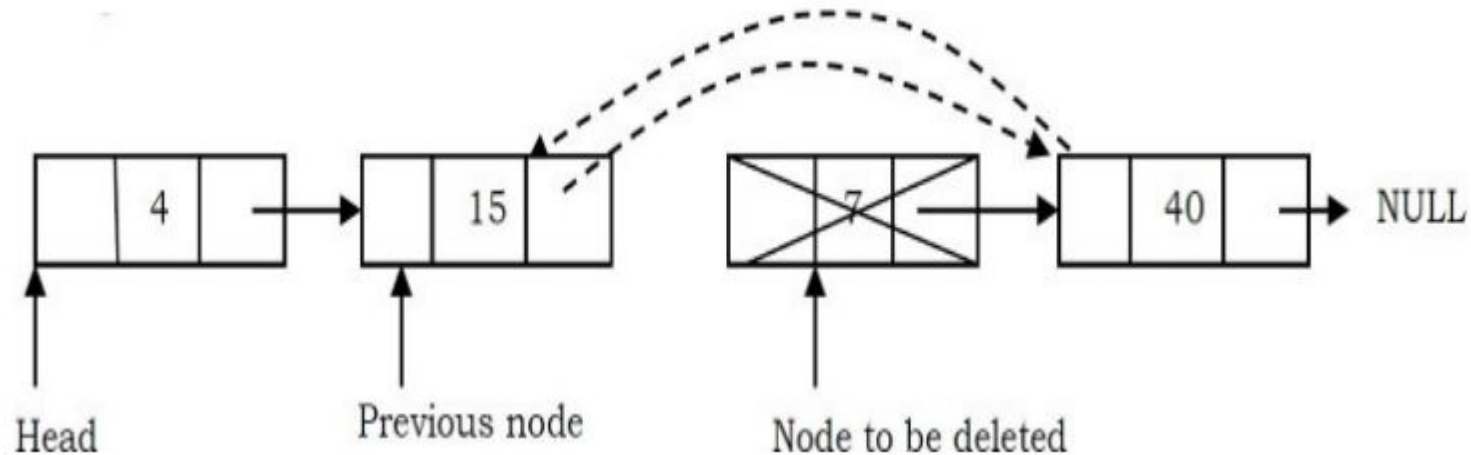
Operations: Delete an Intermediate Node


- 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.



Operations: Delete an Intermediate Node

- 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;
    }
```

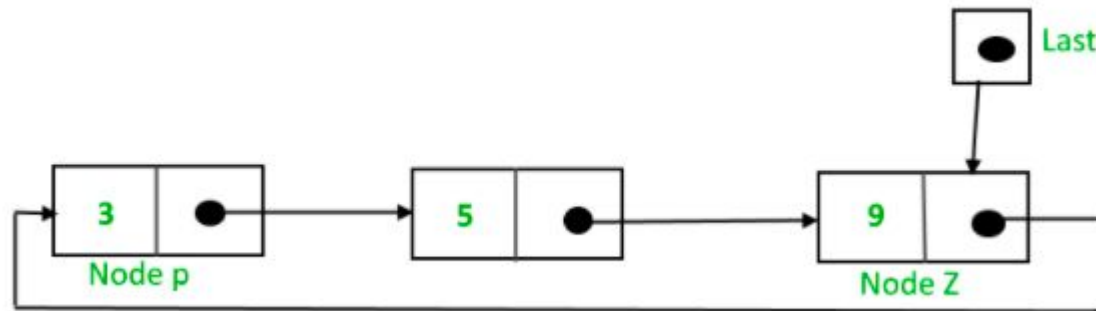
```
    while((k < position) && temp→next!=NULL) {
        temp = temp→next;
        k++;
    }
    if(k!=position-1){
        printf("Desired position does not exist\n");
    }

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

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

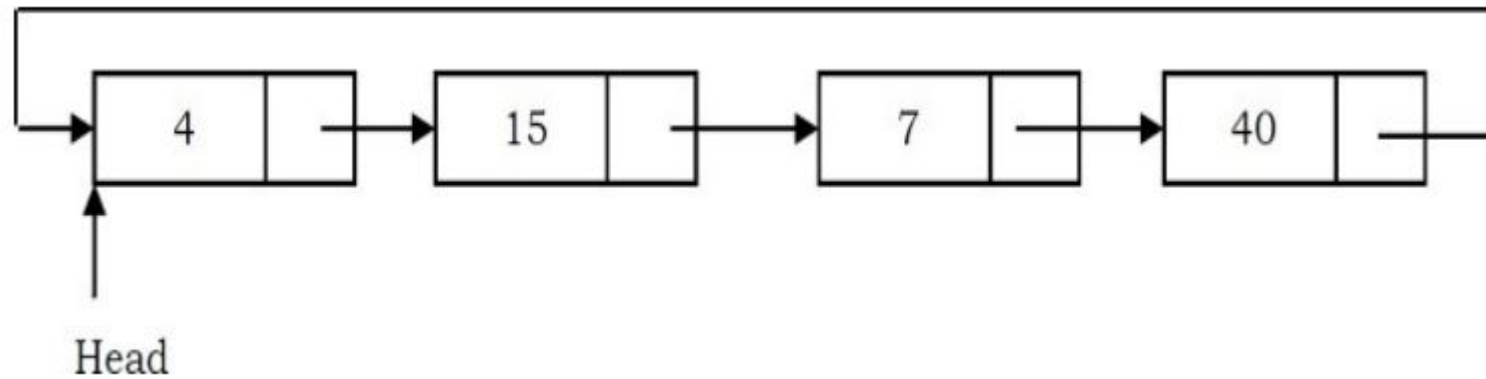
Circular Linked Lists

- In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value. But circular linked lists do not have ends.
- While traversing the circular linked lists we should be careful; otherwise we will be traversing the list infinitely.
- Implementations - Option 1:



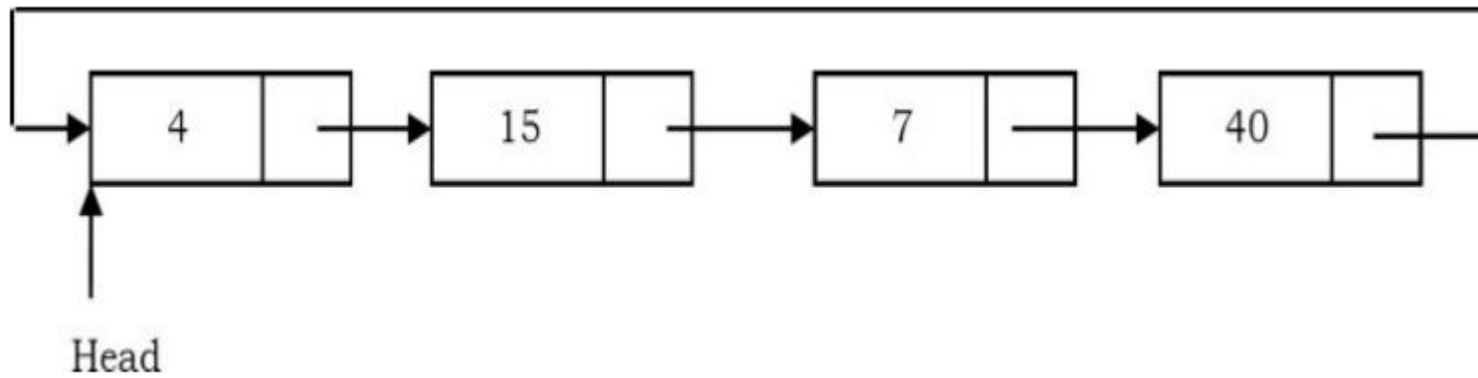
Circular Linked Lists


- Implementations - Option 2:



Operations: Traversing

- 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.



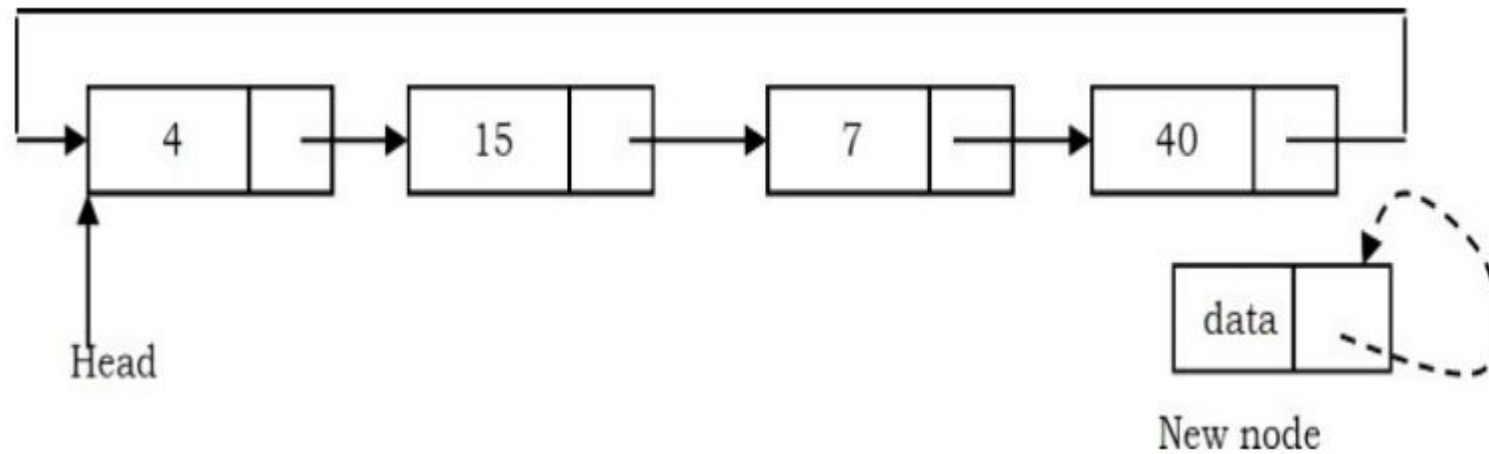


```
void PrintCircularListData(struct CLLNode *head) {
    struct CLLNode *current = head;
    if(head == NULL)
        return;

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

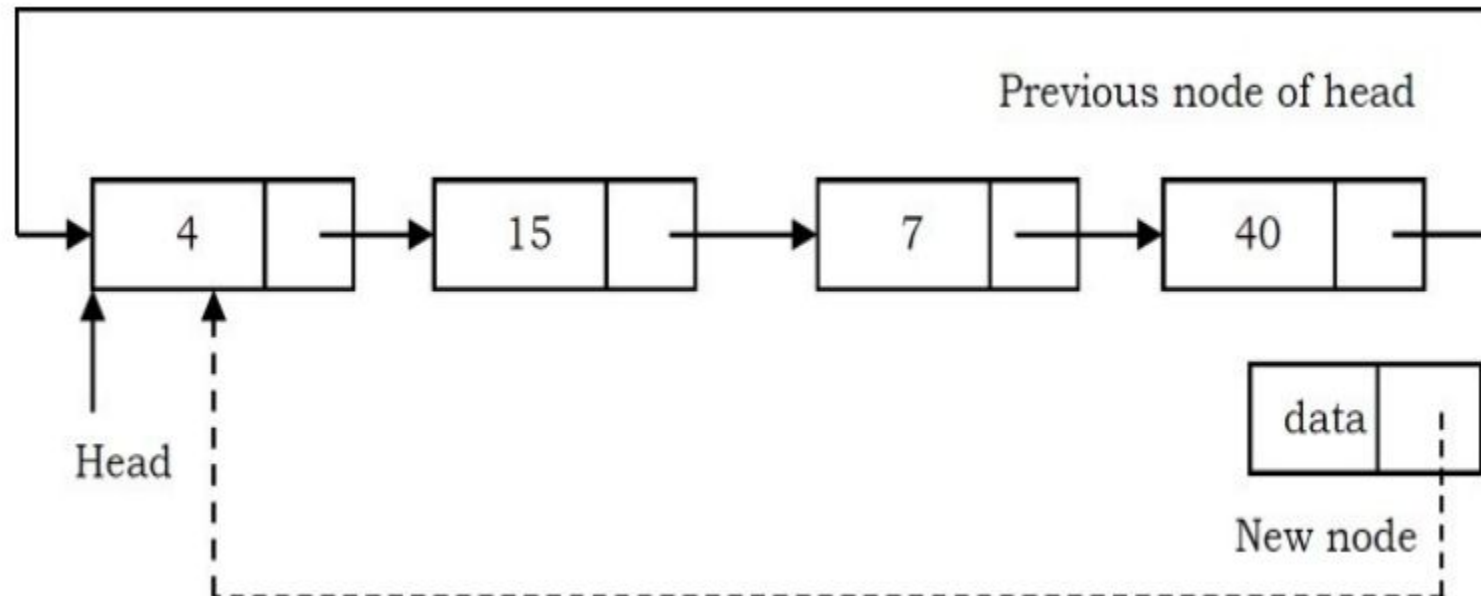
Operations: Insert - End

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



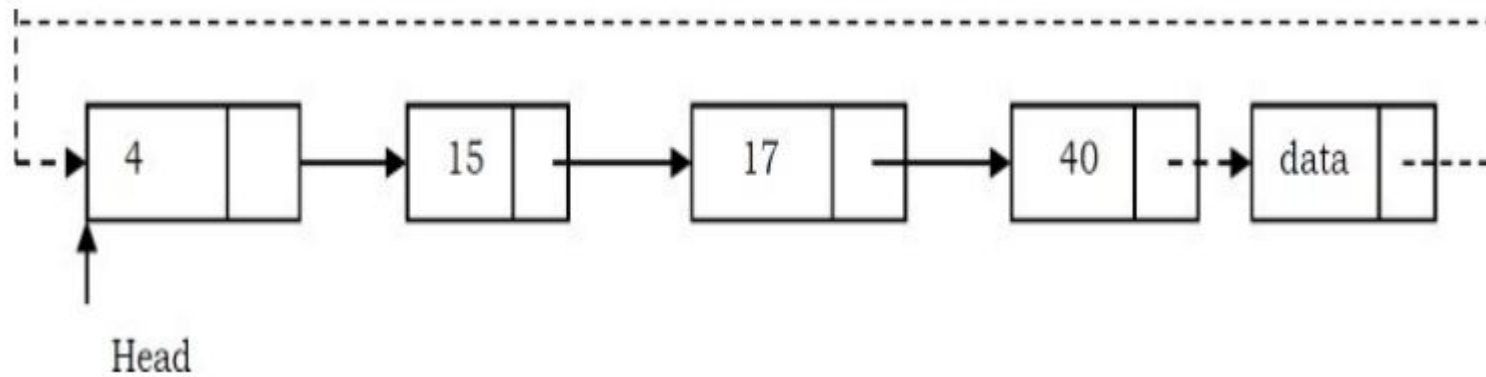
Operations: Insert - End


- 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.



Operations: Insert - End

- Update the next pointer of the previous node to point to the new node and we get:





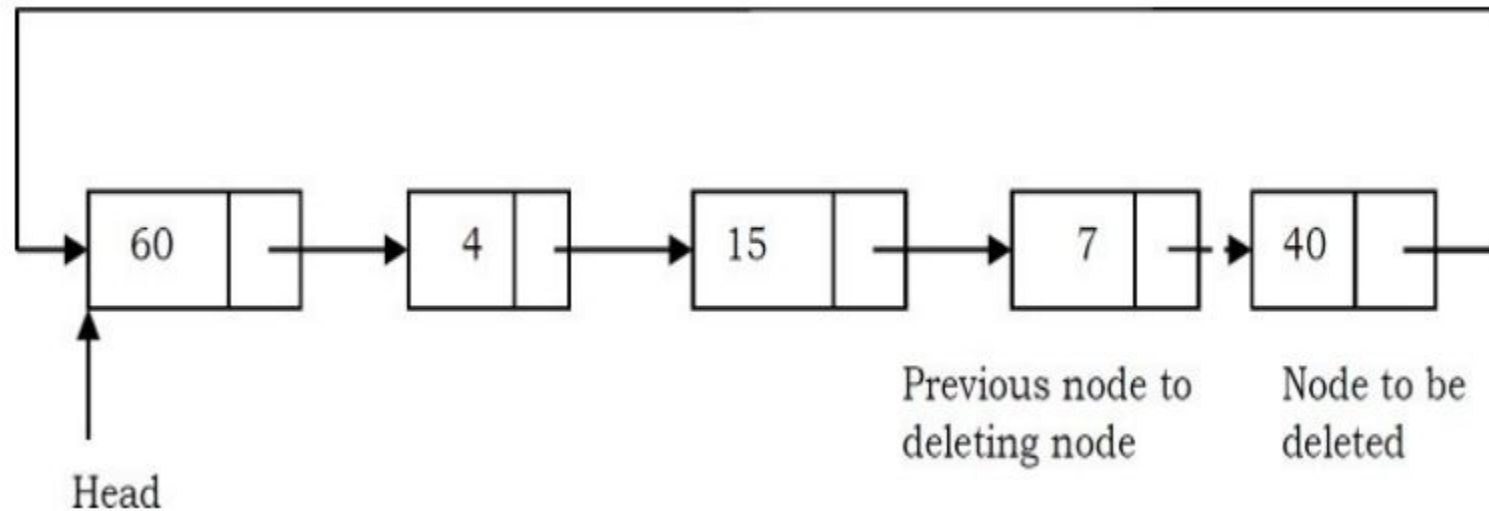
```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
    struct CLLNode *current = *head;
    struct CLLNode *newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    newNode->data = data;
    while (current->next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

    newNode->next = *head;

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

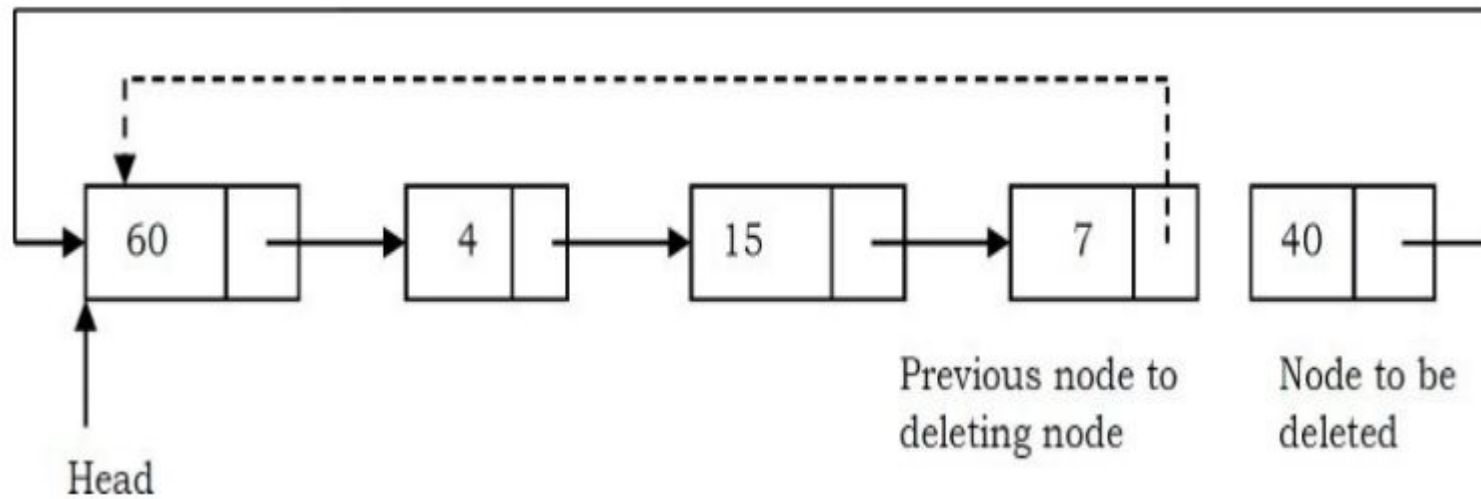
Operations: Delete the Last Node

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



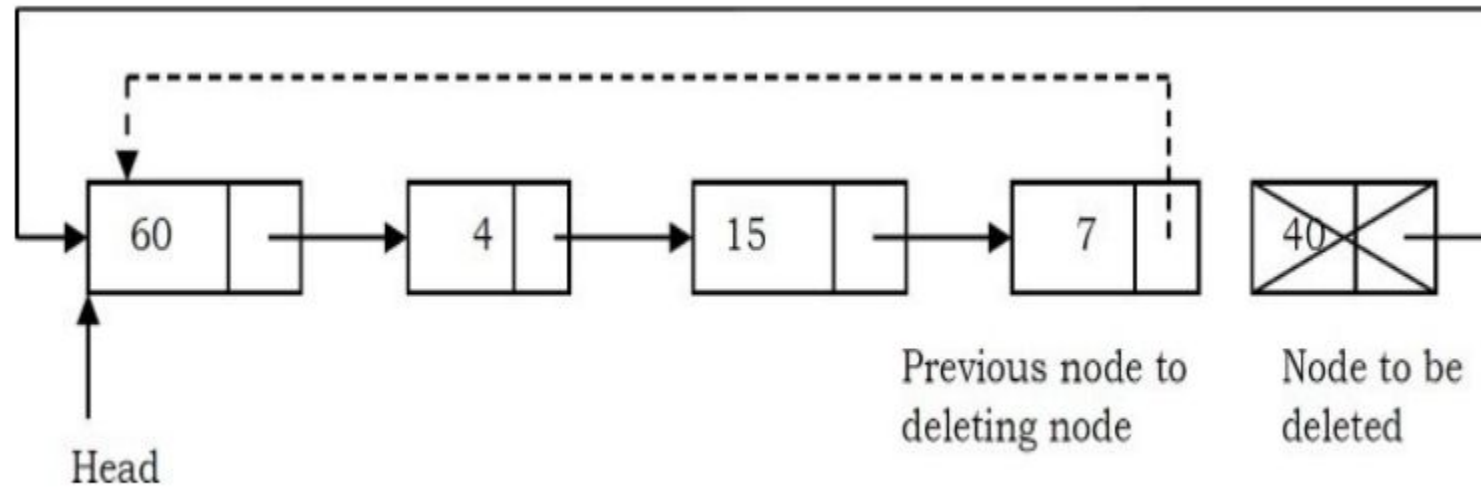
Operations: Delete the Last Node


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



Operations: Delete the Last Node

- Dispose of the tail node.





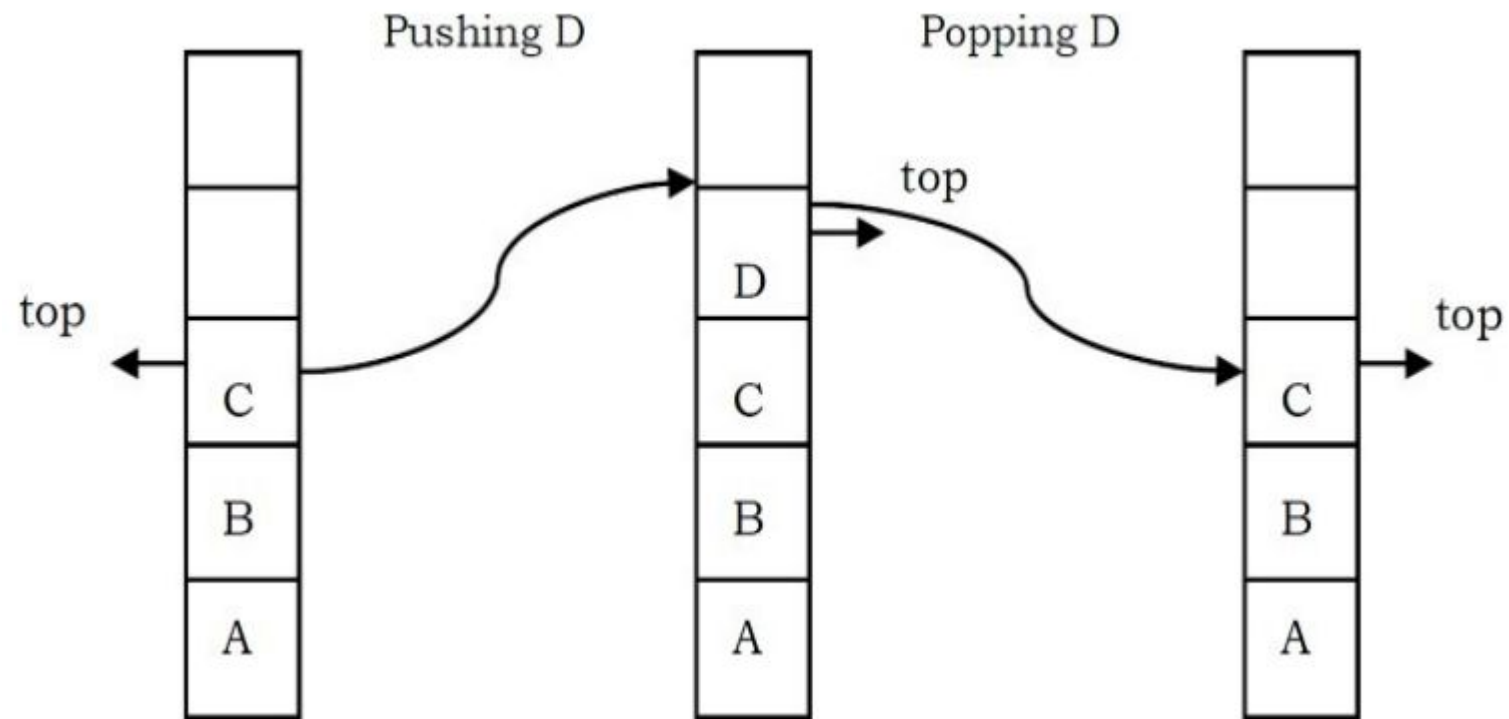
```
void DeleteLastNodeFromCLL (struct CLLNode **head) {  
    struct CLLNode *temp = *head, *current = *head;  
    if(*head == NULL) {  
        printf( "List Empty"); return;  
    }  
    while (current->next != *head) {  
        temp = current;  
        current = current->next;  
    }  
    temp->next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



Stacks

- A stack is a simple data structure used for storing data (similar to Linked Lists).
- In a stack, the order in which the data arrives is important.
- **Definition:** *A stack is an ordered list in which insertion and deletion are done at one end, called top. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.*

Stacks





Operations:

- `Push (int data)`: Inserts data onto stack.
- `int Pop()`: Removes and returns the last inserted element from the stack.
- `Top()`: Returns the last inserted element without removing it.
- `int Size()`: Returns the number of elements stored in the stack.
- `int IsEmptyStack()`: Indicates whether any elements are stored in the stack or not.
- `int IsFullStack()`: Indicates whether the stack is full or not.



Applications:

- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Page-visited history in a Web browser
- Undo sequence in a text editor



Implementations

There are many ways of implementing stack ADT; below are the commonly used methods:

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Implementations: Simple Array

- We add elements from left to right and use a variable to keep track of the index of the top element.
- The maximum size of the stack must first be defined and it cannot be changed.
- Trying to push a new element into a full stack causes an implementation-specific exception.



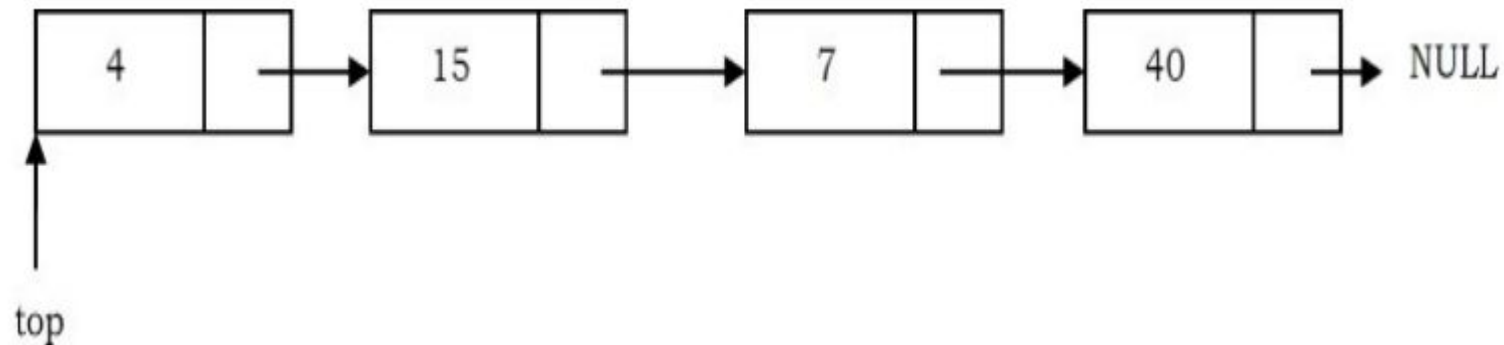


Implementations: Dynamic Array

- The issue that still needs to be resolved is what we do when all the slots in the fixed size array stack are occupied?
- What if we increment the size of the array by 1 every time the stack is full?
 - Push(); increase size of $S[]$ by 1
 - Pop(): decrease size of $S[]$ by 1
- If the array is full, create a new array of twice the size, and copy the items. With this approach, pushing n items takes time proportional to n .
- Too many **doublings** may cause memory overflow exception.

Implementations: Linked List

- Push operation is implemented by inserting element at the beginning of the list.
- Pop operation is implemented by deleting the node from the beginning (the header/top node).





Implementations: Linked List

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) – “amortized” bound takes time proportional to n .

Linked List Implementation

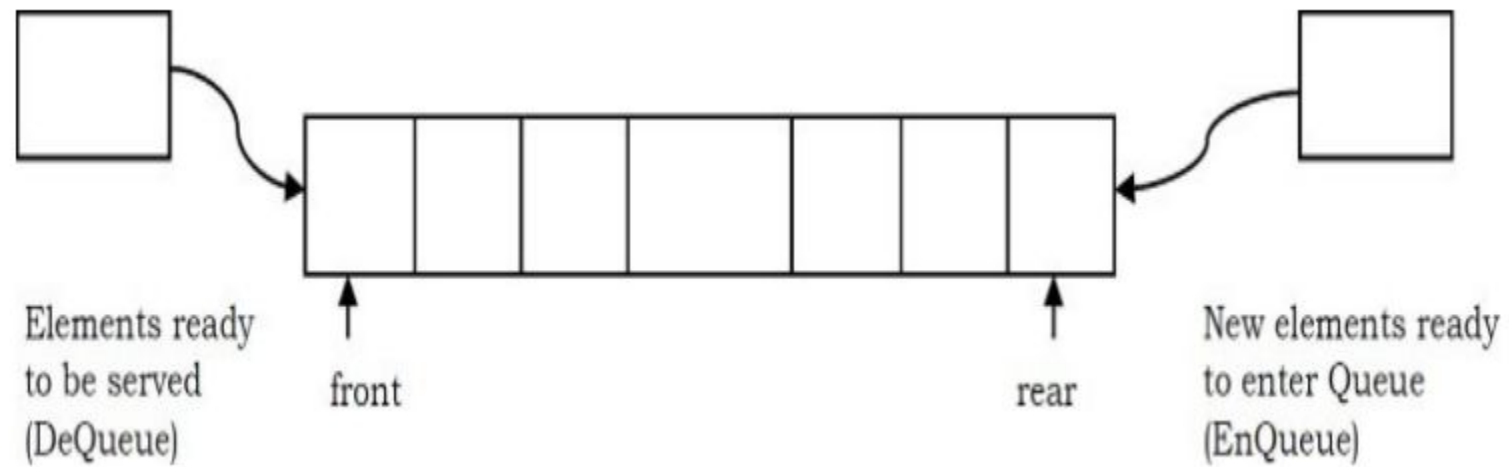
- Grows and shrinks gracefully.
- Every operation takes constant time $O(1)$.
- Every operation uses extra space and time to deal with references.



Queues

- A queue is a data structure used for storing data (similar to Linked Lists and Stacks). I
- The order in which data arrives is important.
- **Definition:** A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Queues





Operations

- `EnQueue(int data)`: Inserts an element at the end of the queue
- `int DeQueue()`: Removes and returns the element at the front of the queue
- `int Front()`: Returns the element at the front without removing it
- `int QueueSize()`: Returns the number of elements stored in the queue
- `int IsEmptyQueueQ`: Indicates whether no elements are stored in the queue or not



Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.

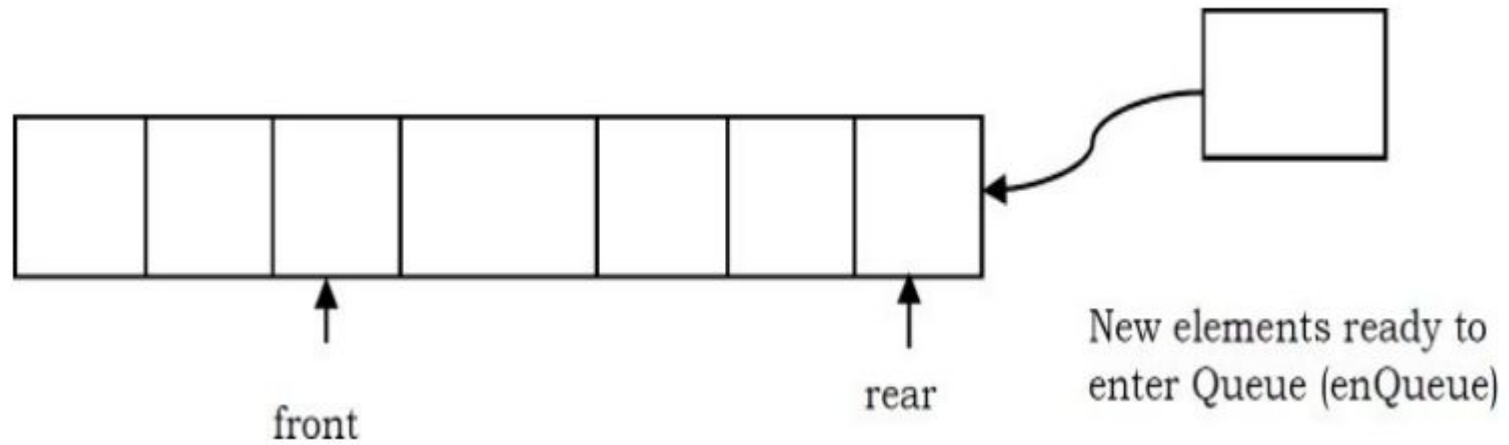


Implementations

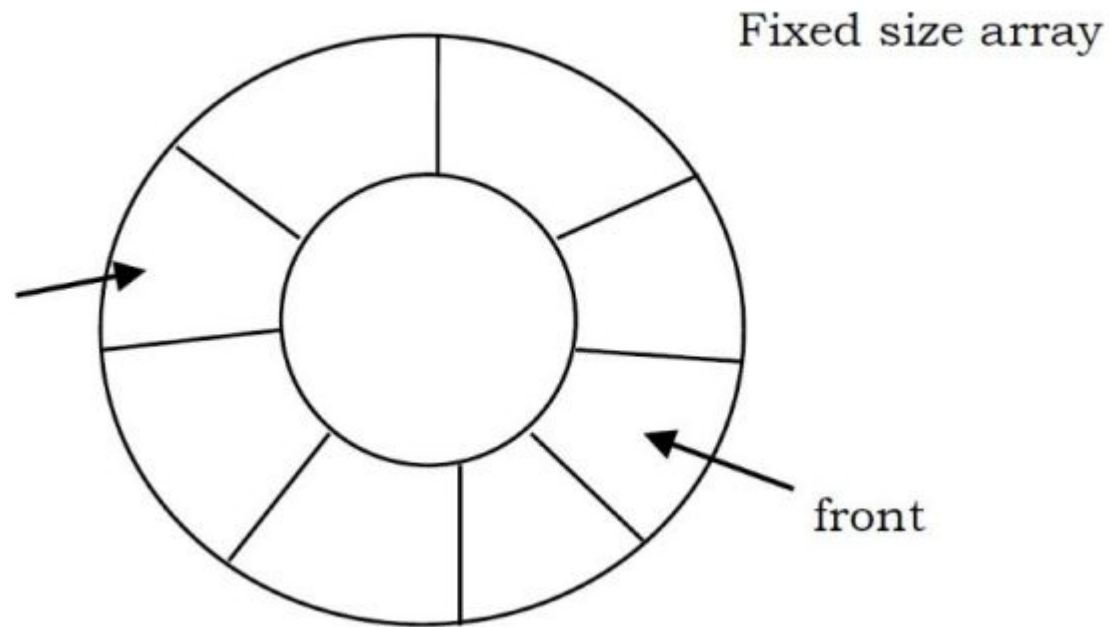
There are many ways (similar to Stacks) of implementing queue operations and some of the commonly used methods are listed below.

- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation

Implementations: Simple/Dynamic Circular Array



Implementations: Simple/Dynamic Circular Array



Implementations: Linked List

- EnQueue operation is implemented by inserting an element at the end of the list.
- DeQueue operation is implemented by deleting an element from the beginning of the list.

