By-Debajyoti Dutta

LINKED LIST

OVERVIEW & OBJECTIVES

A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node. In this chapter, we are going to discuss different types of linked lists and the operations that can be performed on these lists.

NODE

DATA	ADDRESS

USE CASES

- Implementation of stacks and queues

- Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names.
- Performing arithmetic operations on long integers.
- Manipulation of polynomials by storing constants in the node of linked list.
- Representing sparse matrices.

Applications of linked list in real world

- ***Image viewer*** – Previous and next images are linked, hence can be accessed by next and previous button.
- ***Previous and next page in web browser*** – We can access previous and next url searched in web browser by pressing back and next button since they are linked as linked list.
- ***Music Player*** – Songs in the music player are linked to previous and next song. you can play songs either from starting or ending of the list.

OPERATIONS

- Create node
- Insertion
- Deletion
- Traversing
- Searching
- Concatenation
- Display etc…

ADVANTAGES

- ***Dynamic data structure:*** A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.
- ***No memory wastage:*** In the Linked list, efficient memory utilization can be achieved since the size of the linked list increase or decrease at run time so there is no memory wastage and there is no need to pre-allocate the memory.
- ***Implementation:*** Linear data structures like stack and queues are often easily implemented using a linked list.
- ***Insertion and Deletion Operations:*** Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element; only the address present in the next pointer needs to be updated.

DISADVANTAGES

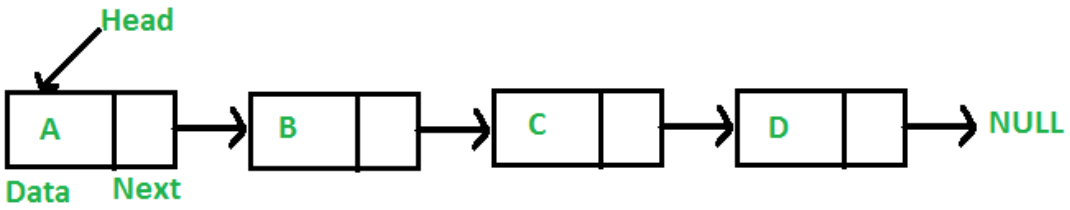
- ***Memory usage:*** More memory is required in the linked list as compared to an array. Because in a linked list, a pointer is also required to store the address of the next element and it requires extra memory for itself.
- ***Traversal:*** In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list as in an array by index. For example, for accessing a mode at position n, one has to traverse all the nodes before it.
- ***Reverse Traversing:*** In a singly linked list reverse traversing is not possible, but in the case of a doubly-linked list, it can be possible as it contains a pointer to the previously connected nodes with each node. For performing this extra memory is required for the back pointer hence, there is a wastage of memory.

- ***Random Access:*** Random access is not as possible in a linked list due to its dynamic memory allocation.

TYPES

- ➢ Singly linked list
- ➢ Doubly linked list
- ➢ Circular linked list
- ➢ Doubly circular linked list

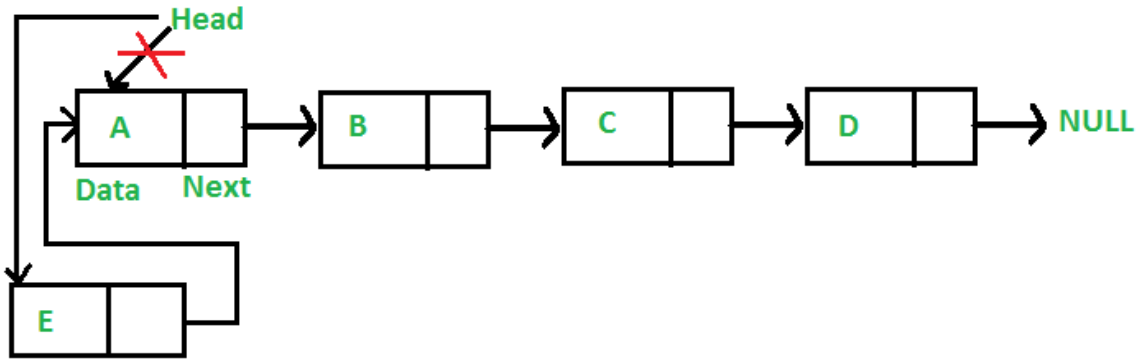
SINGLY LINKED LIST



CREATE NODE

```
typedef Struct
{
        int data;
        Struct node *next;
}NODE;
NODE *start=NULL;
NODE *P=(NODE *) malloc(sizeof(NODE));
```

INSERT AT FIRST POSITION



Algorithm

NODE *PTR

int item

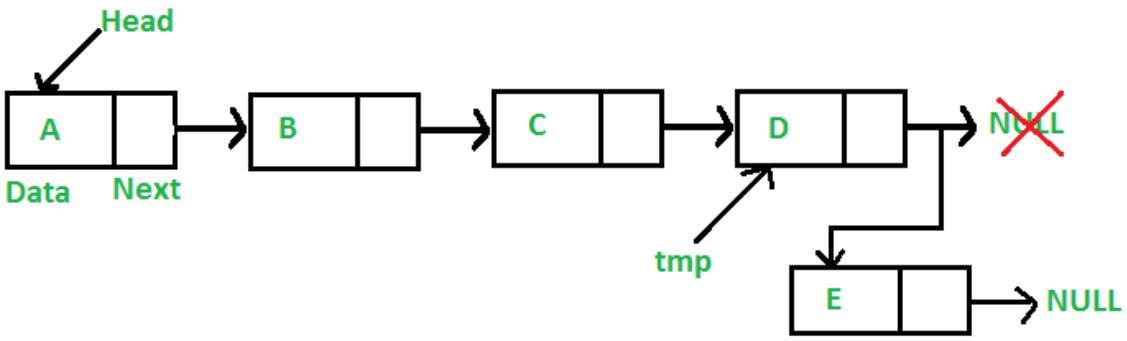
Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: PTR -> next = start

Step 4: start = PTR

INSERT AT LAST POSITION



Algorithm

NODE *PTR , *LOC

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: PTR -> next = NULL

Step 4: If start = NULL then

Set start = PTR

(End of if structure)

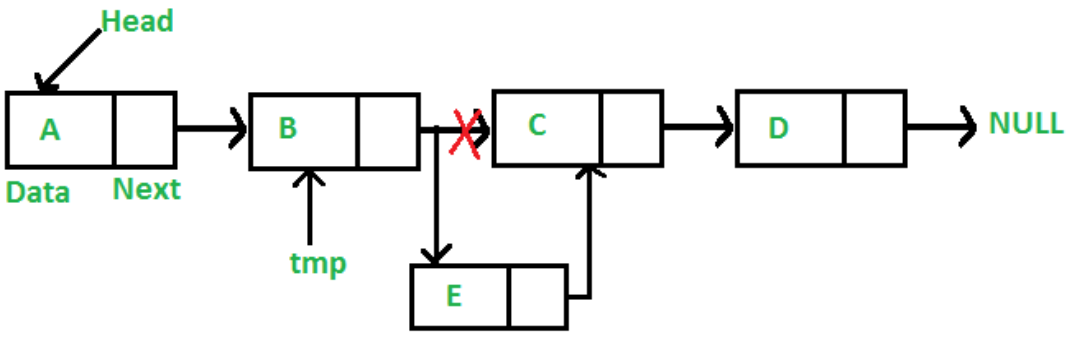
Step 5: LOC = start

Step 6: Repeat step 7 until LOC -> next != NULL

Step 7: LOC = LOC -> next

Step 8: LOC -> next = PTR

INSERT AT SPECIFIED POSITION



Algorithm

NODE *PTR , *temp

int item , loc , I

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: If start = NULL then

Set start = PTR

Set PTR -> next = NULL

(End of if structure)

Step 4: Set I = 1

Set temp = start

Step 5: Repeat step 6 and 7 until I < loc - 1

Step 6: Set temp = temp -> next

Step 7: Set I = I + 1

Step 8: Set PTR -> next = temp -> next

Step 9: temp -> next = PTR

TRAVERSING

Algorithm

NODE *PTR

Step 1: If start = NULL then

Print “Underflow”

Exit

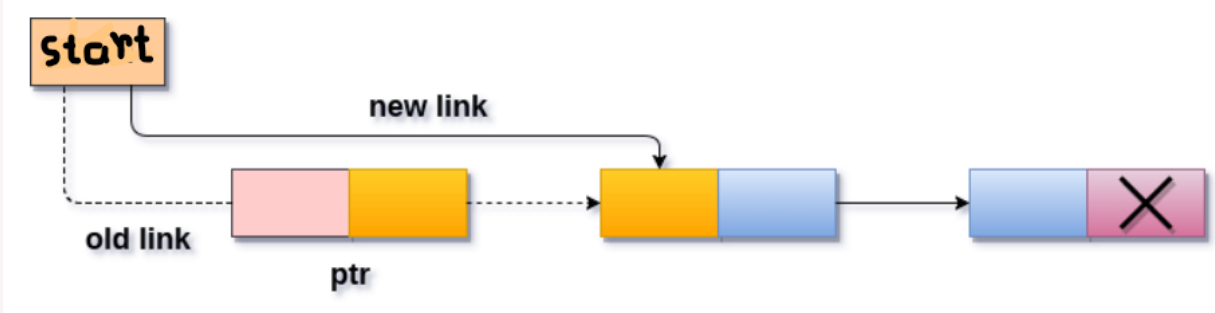
Step 2: PTR = start

Step 3: Repeat step 4 and 5 until PTR != NULL

Step 4: Print PTR -> data

Step 5: PTR = PTR -> next

DELETE AT FIRST POSITION



Algorithm

NODE *PTR

Step 1: PTR = start

Step 2: If PTR = NULL then

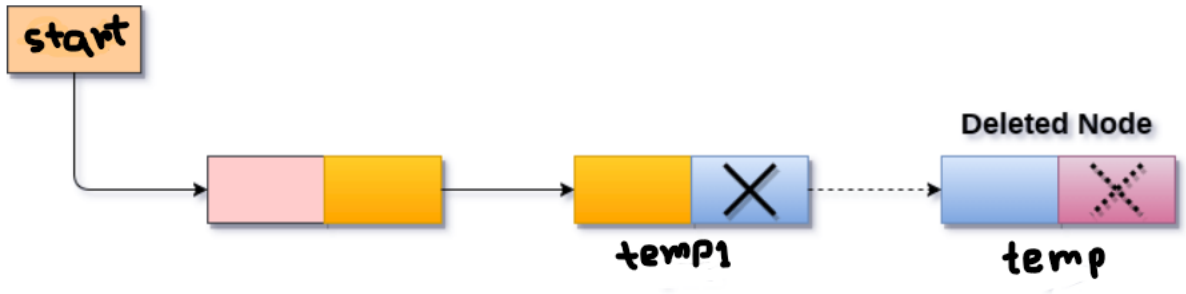
Print “Underflow”

Exit

Step 3: start = start -> next

Step 4: free(PTR)

DELETE AT LAST POSITION



Algorithm

NODE *temp , *temp1

Step 1: If start = NULL then

Print “Underflow”

Exit

Step 2: temp1 = start

temp = start -> next

Step 3: Repeat step 4 until temp != NULL

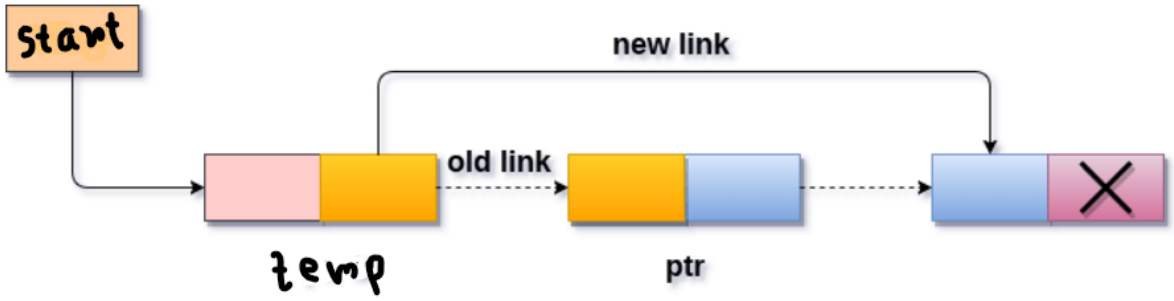
Step 4: temp1 = temp1 -> next

temp = temp -> next

Step 5: temp1 -> next = NULL

Step 6: free(temp)

DELETE AT SPECIFIED POSITION



Algorithm

NODE *temp , *ptr

int I , loc

Step 1: If start = NULL then

Print “Underflow”

Exit

Step 2: set I=0

ptr=start

Step 3: Repeat step 4 until I<=loc

Step 4: set temp=ptr

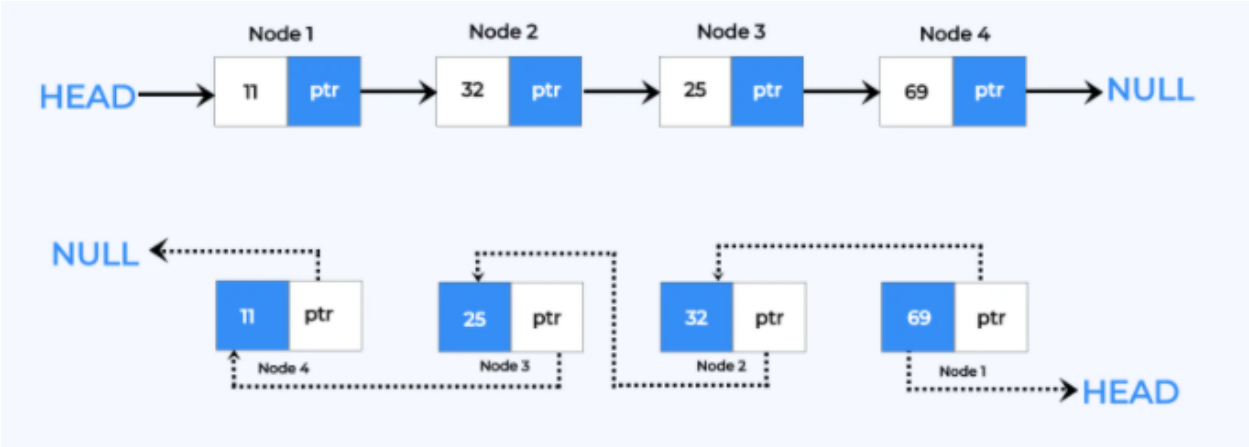
Set ptr = ptr -> next

Set I=I+1

Step 5: Set temp->next=ptr->next

Step 6: free(ptr)

REVERSE OF A LINKED LIST



Algorithm

NODE *next , *current , *prev

Step 1: If start = NULL then

Print “Underflow”

Exit

Step 2: current=start

Step 3: Repeat step 4 until current !=NULL

Step 4: next=current->next

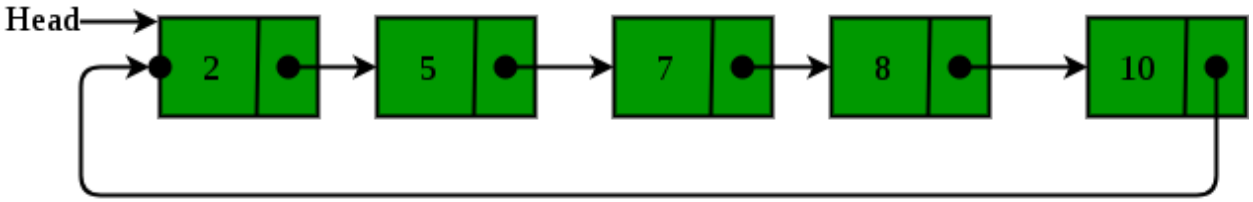
current->next=prev

prev=current

current=next

Step 5: start=prev

CIRCULAR LINKED LIST



TRAVERSING

Algorithm

NODE *PTR

Step 1: If start = NULL then

Print “Underflow”

Exit

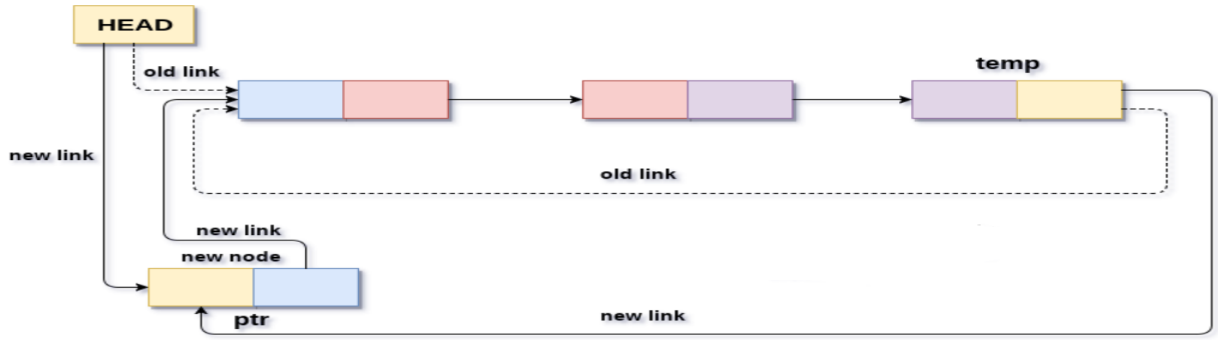
Step 2: PTR = start

Step 3: Repeat step 4 and 5 until PTR != start

Step 4: Print PTR -> data

Step 5: PTR = PTR -> next

INSERT AT FIRST POSITION



Algorithm

NODE *PTR , *temp

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: If start=NULL then

Set PTR -> data = item

Set PTR -> next = PTR

Set start = PTR

End of if structure

Step 3: Set PTR -> data = item

Step 4: Set PTR -> next = start

Step 5: temp= start

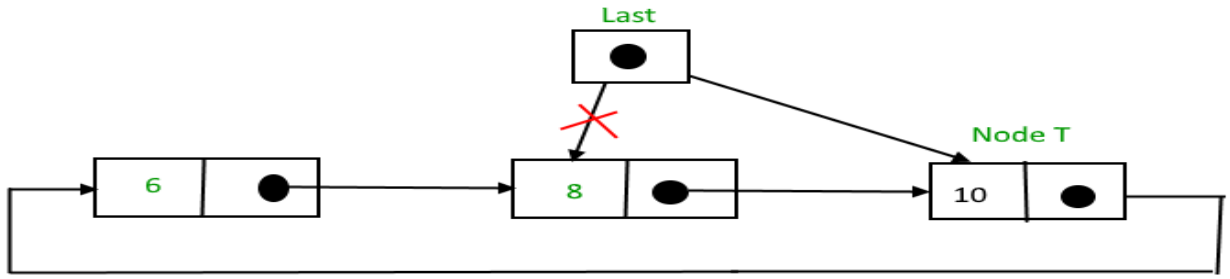
Step 6: Repeat step 7 until temp -> next != start

Step 7: temp = temp -> next

Step 8 temp -> next = PTR

Step 9: start = PTR

INSERT AT LAST POSITION



Algorithm

NODE *PTR , *temp

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: If start=NULL then

Set PTR -> data = item

Set PTR -> next = PTR

Set start = PTR

End of if structure

Step 3: Set PTR -> data = item

Step 4: temp= start

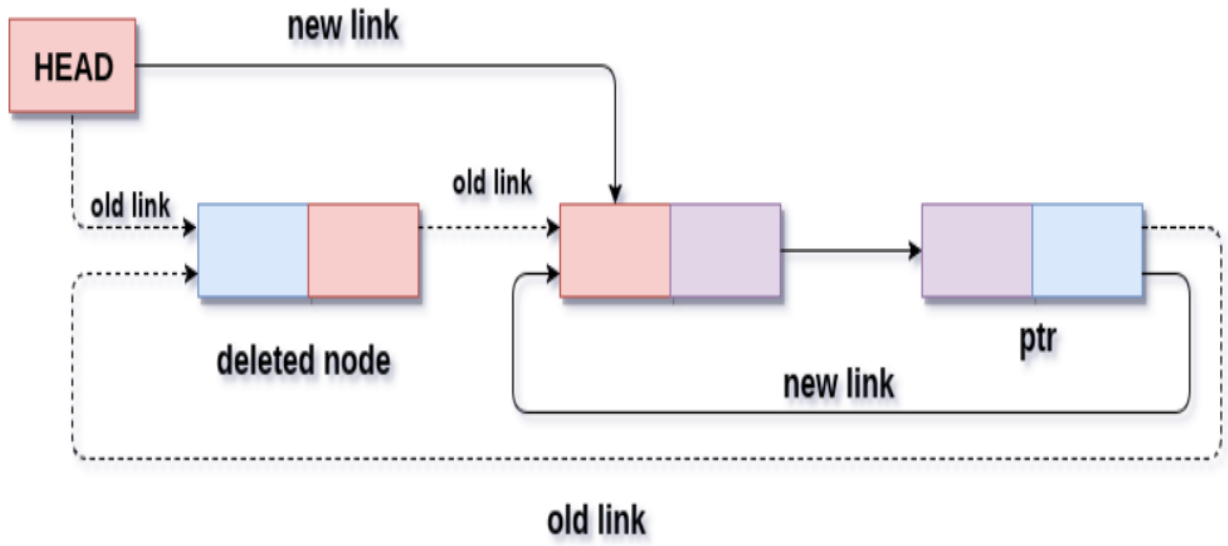
Step 5: Repeat step 6 until temp -> next != start

Step 6: temp = temp -> next

Step 7: temp -> next = PTR

Step 8: PTR -> next = start

DELETE AT FIRST POSITION



Algorithm

NODE *PTR , *temp

Step 1: If start=NULL then

Print “List is empty”

Exit

Step 2: Set PTR = start

Set temp = start

Step 3: Repeat step 4 until temp -> next != start

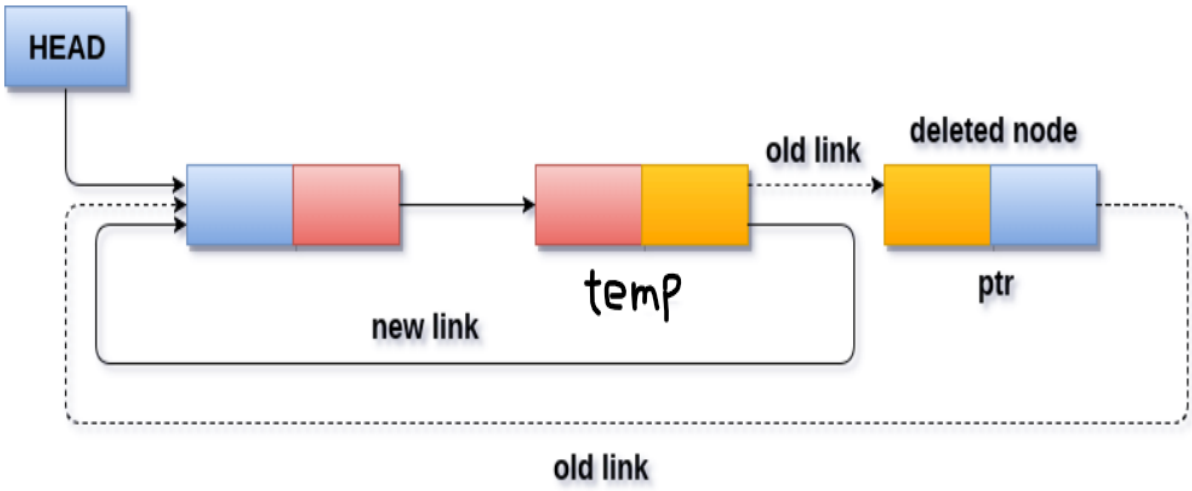
Step 4: temp = temp -> next

Step 5: start = start -> next

Step 6: temp -> next = start

Step 7: free(PTR)

DELETE AT LAST POSITION



Algorithm

NODE *PTR , *temp

Step 1: If start=NULL then

Print “List is empty”

Exit

Step 2: Set PTR = start

Step 3: Repeat step 4 and 5 until PTR -> next != start

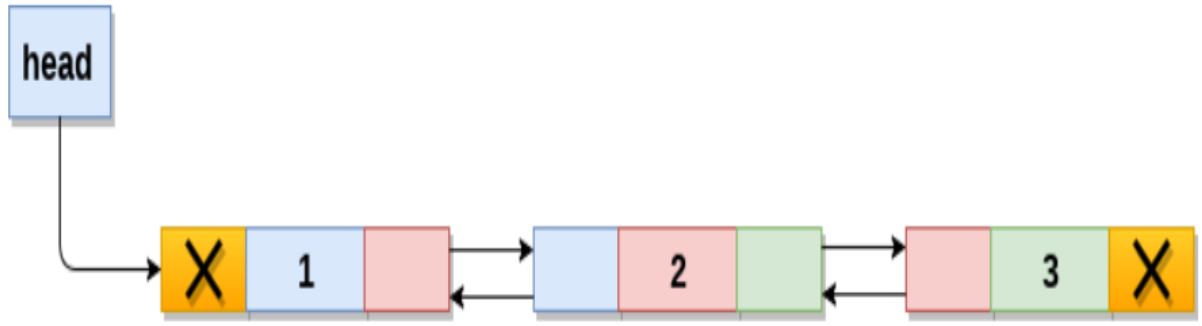
Step 4: Set temp = PTR

Step 5: PTR = PTR -> next

Step 6: temp -> next = start

Step 7: free(PTR)

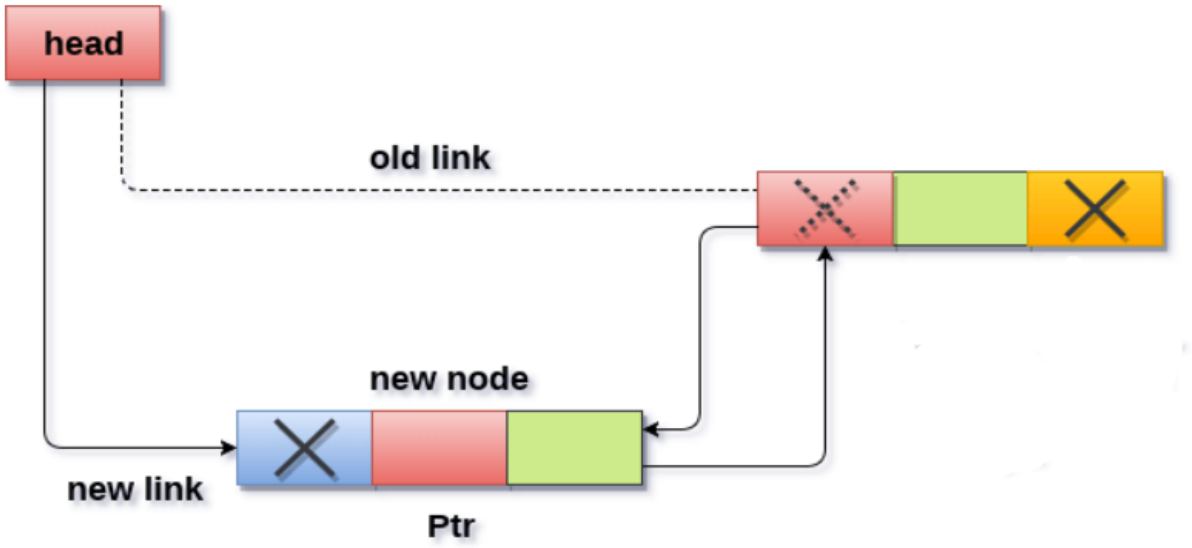
DOUBLY LINKED LIST



CREATE NODE

```
typedef Struct
{
        Struct node *prev;
        int data;
        Struct node *next;
}NODE;
NODE *head=NULL;
NODE *P=(NODE *) malloc(sizeof(NODE));
```

INSERT AT FIRST POSITION



Algorithm

NODE *PTR

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

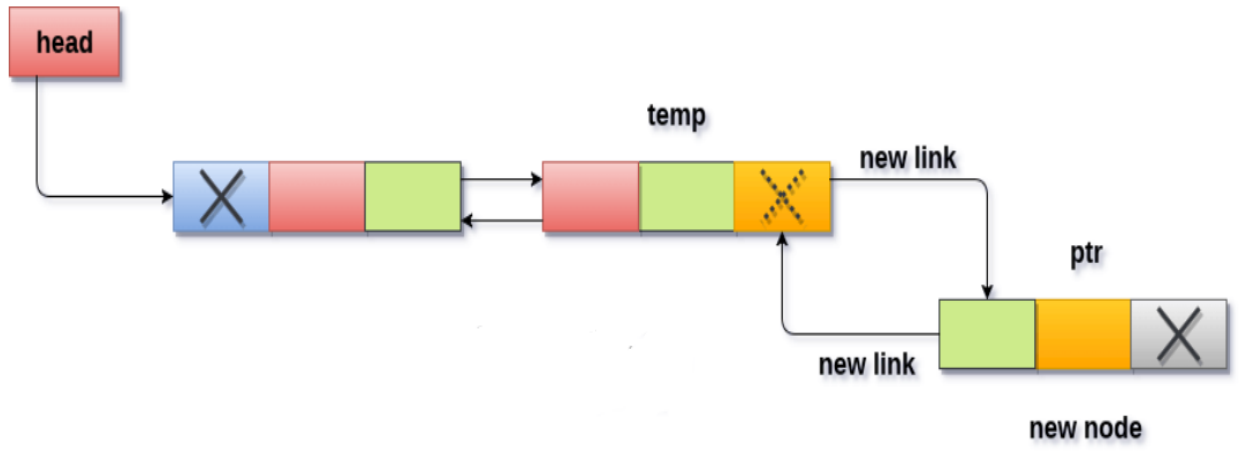
Step 3: PTR -> prev = NULL

Step 4: PTR -> next = head

Step 5: head -> prev = PTR

Step 6: head = PTR

INSERT AT LAST POSITION



Algorithm

NODE *PTR , *temp

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: Set temp = head

Step 4: Repeat step 5 until temp -> next != NULL

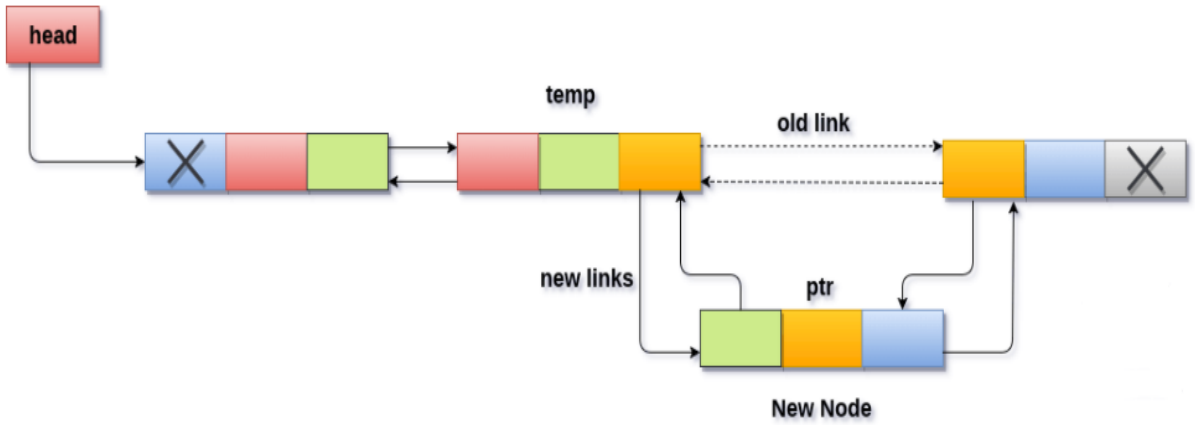
Step 5: temp = temp -> next

Step 6: temp -> next = PTR

Step 7: PTR -> prev = temp

Step 8: PTR -> next = NULL

INSERT AT SPECIFIED POSITION



Algorithm

NODE *PTR , *temp

int item ,I , loc

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: Set temp = head

Step 4: Set I=1

Step 5: Repeat step 6 until I < loc - 1

Step 6: temp = temp -> next

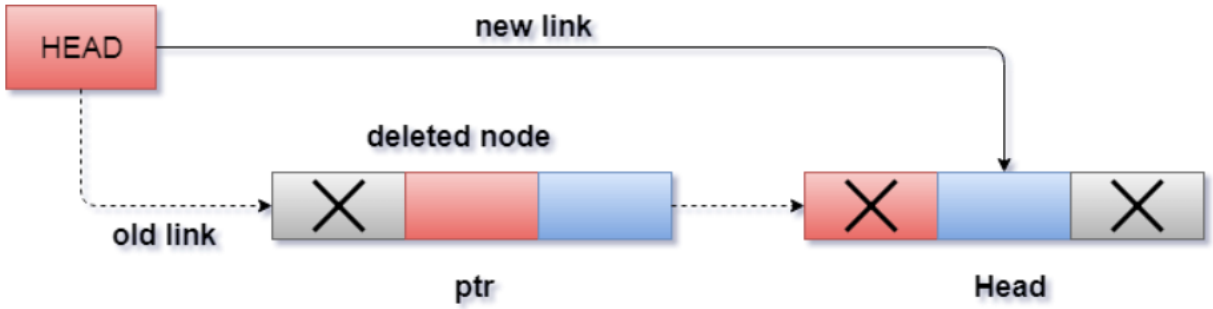
Step 7: PTR -> prev = temp

Step 8: PTR -> next = temp -> next

Step 9: temp -> next -> prev = PTR

Step 10: temp -> next = PTR

DELETE AT FIRST POSITION



Algorithm

NODE *PTR

Step 1: If head=NULL then

Print “List is empty”

Exit

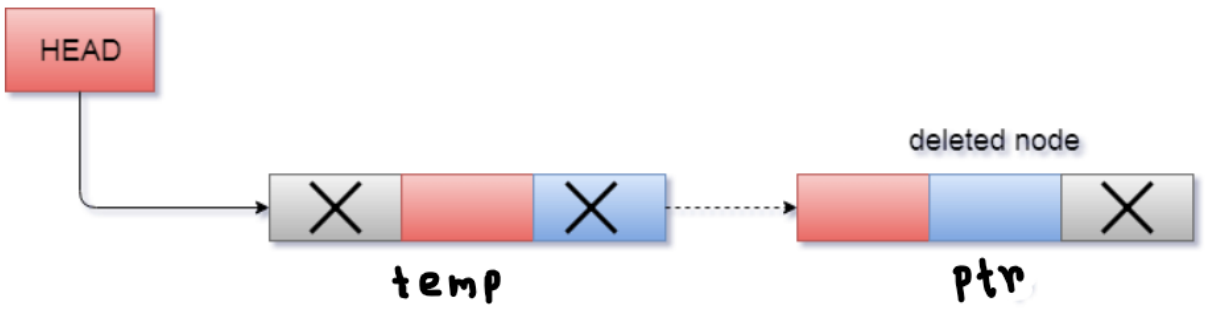
Step 2: Set PTR = head

Step 3: head = head -> next

Step 4: head -> prev = NULL

Step 5: free(PTR)

DELETE AT LAST POSITION



Algorithm

NODE *PTR , *temp

Step 1: If head=NULL then

Print “List is empty”

Exit

Step 2: Set PTR = head

Step 3: Repeat step 4 and 5 until ptr -> next != NULL

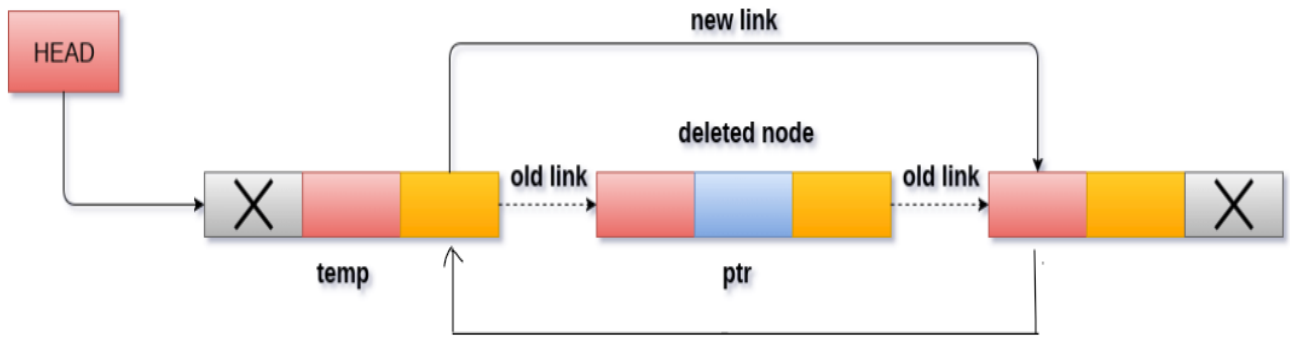
Step 4: temp = PTR

Step 5: PTR = PTR -> next

Step 6: temp -> next =NULL

Step 7: free(PTR)

DELETE AT SPECIFIED POSITION



Algorithm

NODE *PTR , *temp

Int I , loc

Step 1: If head=NULL then

Print “List is empty”

Exit

Step 2: Set temp = head

Set I = 1

Step 3: Repeat step 4 until I < loc - 1

Step 4: temp = temp -> next

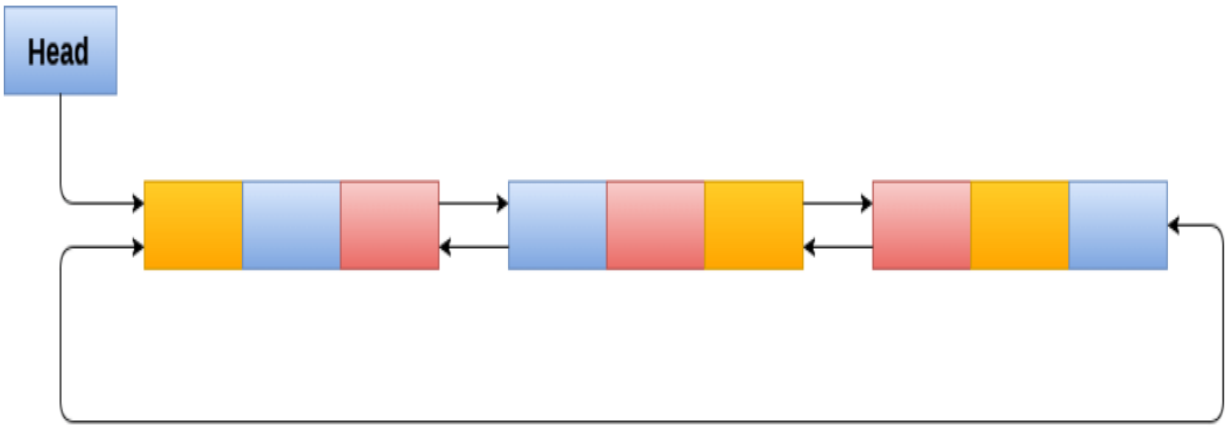
Step 5: PTR = temp -> next

Step 6: temp -> next =PTR -> next

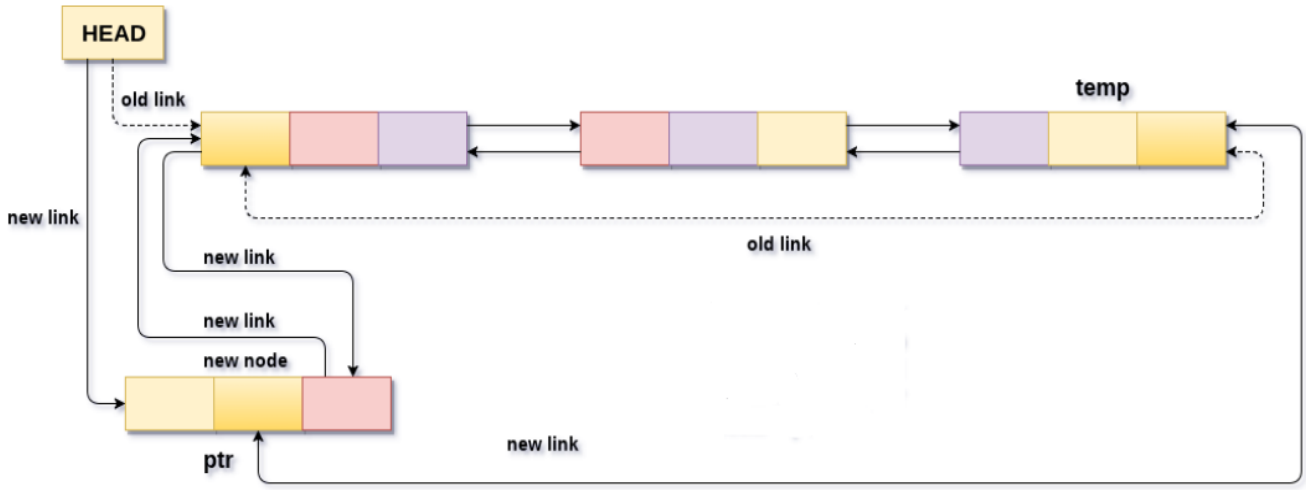
Step 7: PTR -> next -> prev = temp

Step 8: free(PTR)

DOUBLY CIRCULAR LINKED LIST



INSERT AT FIRST POSITION



Algorithm

NODE *PTR , *temp

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: PTR -> next = head

Step 4: head -> prev = PTR

Step 5: Set temp = head

Step 6: Repeat step 8 until temp -> next != head

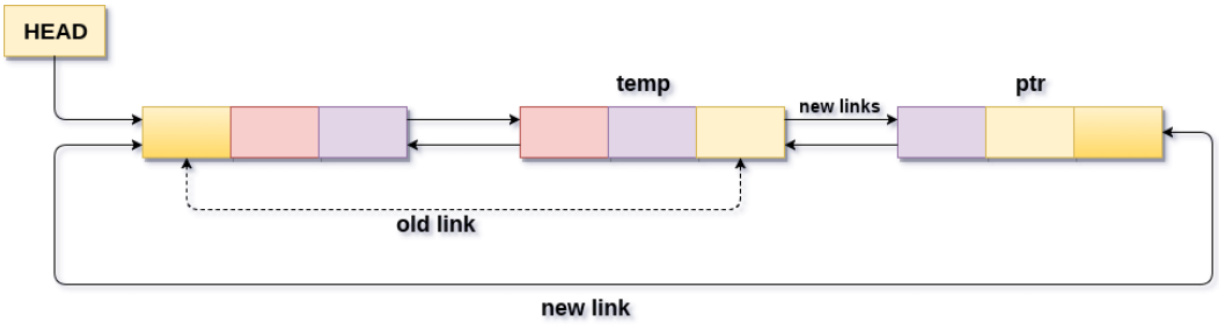
Step 7: temp = temp -> next

Step 8: temp -> next = PTR

Step 9: PTR -> prev = temp

Step 10: head = PTR

INSERT AT LAST POSITION



Algorithm

NODE *PTR , *temp

int item

Step 1: PTR=(NODE*) malloc(sizeof(NODE))

Step 2: PTR -> data = item

Step 3: Set temp = head

Step 4: Repeat step 5 until temp -> next != head

Step 5: temp = temp -> next

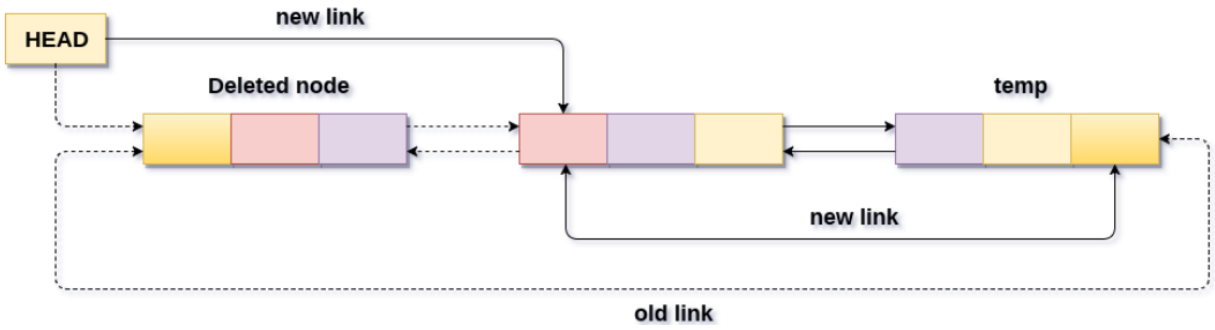
Step 6: temp -> next = PTR

Step 7: PTR -> prev = temp

Step 8: PTR -> next = head

Step 9: head -> prev = PTR

DELETE AT FIRST POSITION



Algorithm

NODE *PTR , *temp

Step 1: If head=NULL then

Print “List is empty”

Exit

Step 2: Set PTR = head

Step 3: Set temp = head

Step 4: Repeat step 5 until temp -> next != head

Step 5: temp = temp -> next

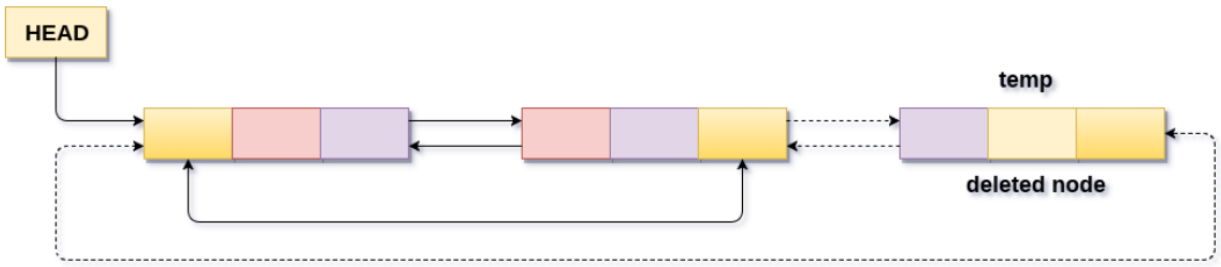
Step 6: head = head -> next

Step 7: temp -> next = head

Step 3: head -> prev = temp

Step 5: free(PTR)

DELETE AT LAST POSITION



Algorithm

NODE *PTR , *temp

Step 1: If head=NULL then

Print “List is empty”

Exit

Step 2: Set PTR = head

Step 3: Repeat step 4 and 5 until ptr -> next != head

Step 4: temp = PTR

Step 5: PTR = PTR -> next

Step 6: temp -> next =head

Step 7: head -> prev = temp

Step 8: free(PTR)

MULTI LINKED LIST