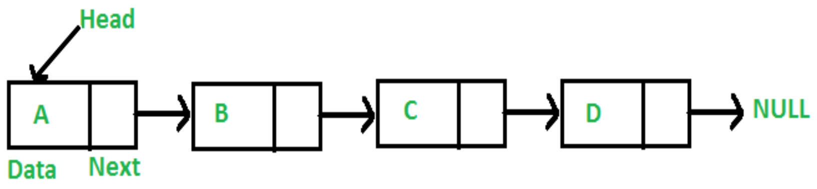
Chapter 4 Linked list

Introduction:

- A linked list is a collection of elements called ‘nodes’ where each node consists of two parts:
    - **Info**: Actual element to be stored in the list. It is called data field.
    - **Next**: one or more link that points to next and previous node in the list. It is also called pointer field.

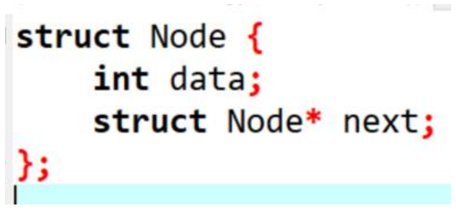


- The link list is a dynamic structure i.e. it grows or shrinks depending on different operations performed. The whole list is pointed to by an external pointer called head which contains the address of the first node. It is not the part of linked list.
- The last node has some specified value called NULL as next address which means the end of the list.

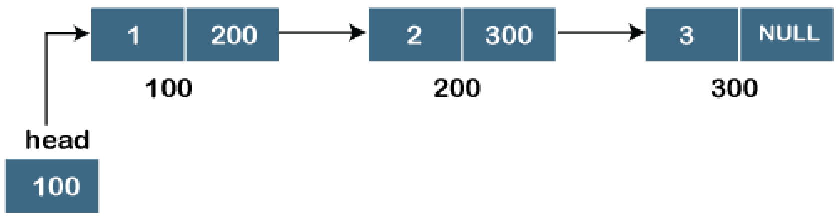
Types of Linked List:

1. Single 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 traversal of data only in one way.

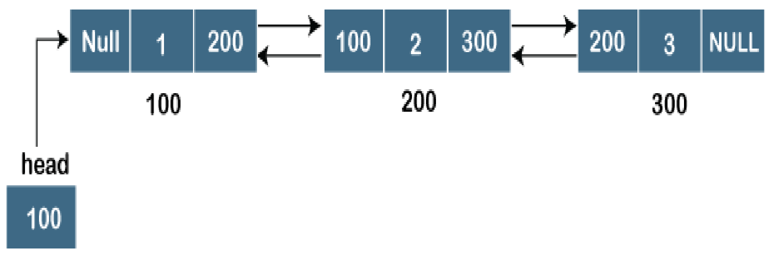


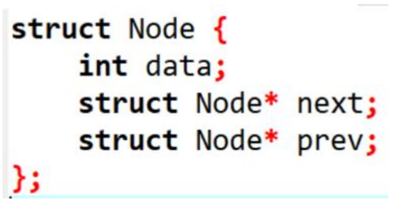
```
struct Node {
    int data;
    struct Node* next;
};
```



2. Doubly Linked list:

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in sequence, Therefore, it contains three parts are 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.

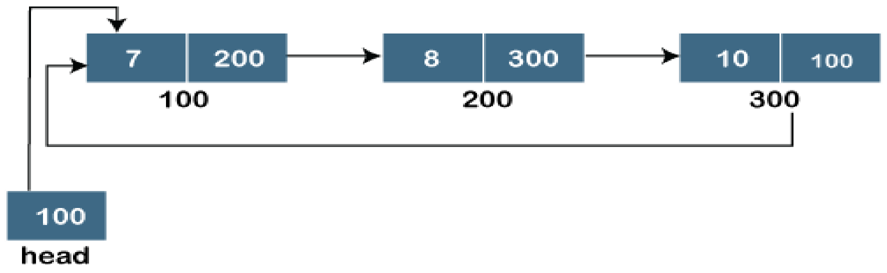




```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
```

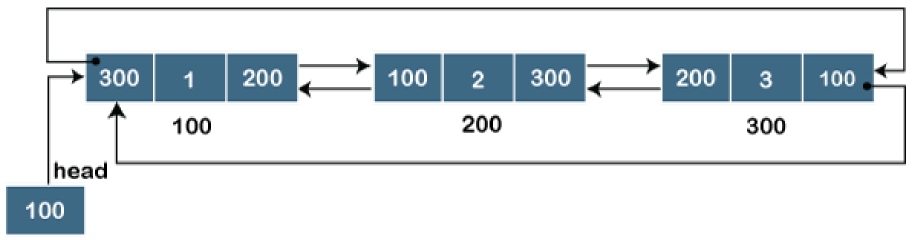
3. Circular linked list:

- 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 liked 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.



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 circular doubly linked list does not contain null in the previous field of the first node.



Chapter 4 Linked list

Dynamic implementation:

Dynamic Memory Allocation: it is a procedure in which the size of data structure is changed during the runtime.

Malloc ():

It is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initializes memory at execution time so that it has initializes each block with the default garbage value initially.

`ptr = (cast-type*) malloc(byte-size)`

e.g. ptr= (int*) malloc(100*size of (int));

Calloc ():

"calloc" or "contiguous allocation" method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0' and has two parameters.

ptr = (cast-type*)calloc(n, element-size);

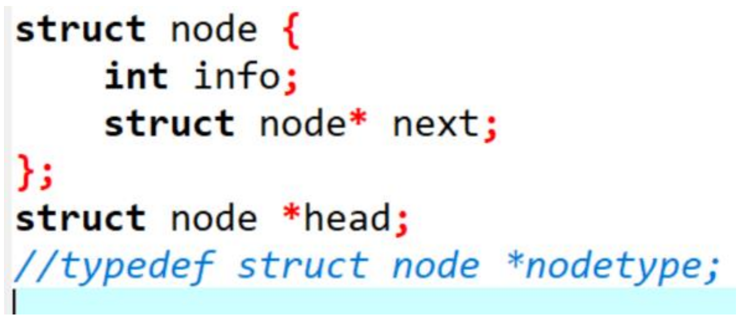
here, n is the no. of elements and element-size is the size of each element.

Free ():

It is used to dynamically de-allocate the memory. The memory allocated using functions malloc () and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Free (ptr);

Representation in C:



```
struct node {
    int info;
    struct node* next;
};
struct node *head;
//typedef struct node *nodetype;
```

For getnode ():

- Let getnode() be a function that allocates memory for a node, assigns data to the node's info, makes node's next pointer NULL and returns the address of the node.

```
struct node *getnode()
{
    struct node *ptr;
    ptr=(struct node *) malloc(sizeof(struct node));
    printf("Enter a data:");
    scanf("%d",&ptr->info);
    prt->next=NULL;
    return ptr;
}
```

- It allocates the memory for a node dynamically. It is a user defined function that returns a pointer to newly created node.

Operations in Single linked list:

Insertion:

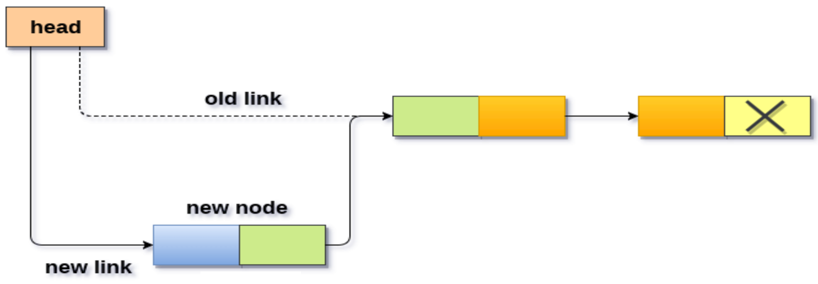
The insertion into a singly linked list can be performed at different positions.

1. **Insertion at beginning:**

   It involves inserting any element at the front of the list. We just need make the new node as the head of the list.

   Algorithm:

```
1. Create a node using malloc function
       newnode=(struct node *)malloc(sizeof(struct node));
2. Assign data to info field of new node
       newnode->info = data;
3. if head is NULL then set head=newnode and exit
           head=newnode;
4. otherwise
       i.  Set next of newnode to head
               newnode->next=head;
       ii. Set the head pointer to point to the new node
               head=newnode;
5. End
```



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.

Algorithm:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set next of newnode to NULL

```
newnode->next =NULL;
```

4. if head is NULL then set head=newnode and exit
5. Otherwise
   i. Set

   ```
   ptr=head;
   ```

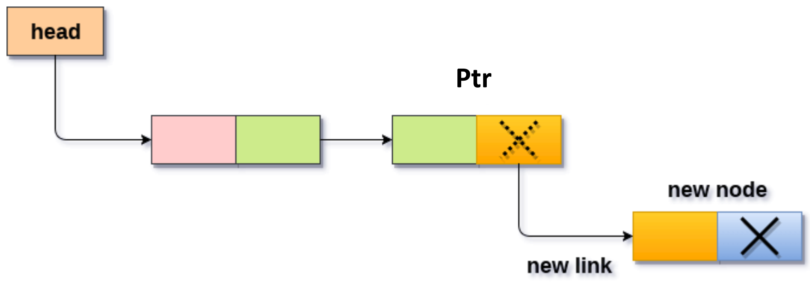
   ii. Find the last node

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

   iii. Set ptr->next =newnode

   ```
   ptr->next =newnode;
   ```

6. end



3. Insertion at specified position:

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

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Enter the position of a node at which you want to insert a newnode.
4. Let the position be pos
5. Set

```
ptr=head;
```

```
for(i=0;i<pos-1;i++)
{ ptr=ptr->next;
        if(ptr==NULL)
        {
                printf("\nPosition not found:[Handle with care]\n");
                return;
        }
}
```

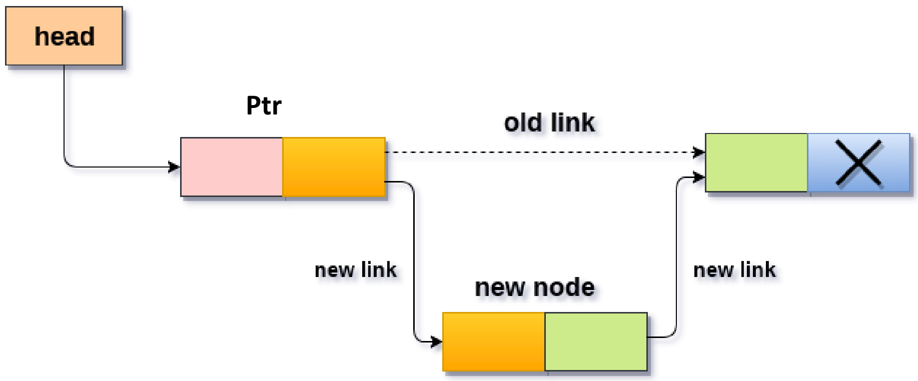
6. Set

```
newnode->next =ptr->next ;
```

7. Set next of ptr to point to the newnode

```
ptr->next=newnode;
```

8. End



Deletion:

1. Deletion at beginning:

It involves deletion of a node from the beginning of the list.

Let head be the pointer to the first node in the linked list

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise store the address of the first node in temporary variable ptr

```
ptr=head;
```

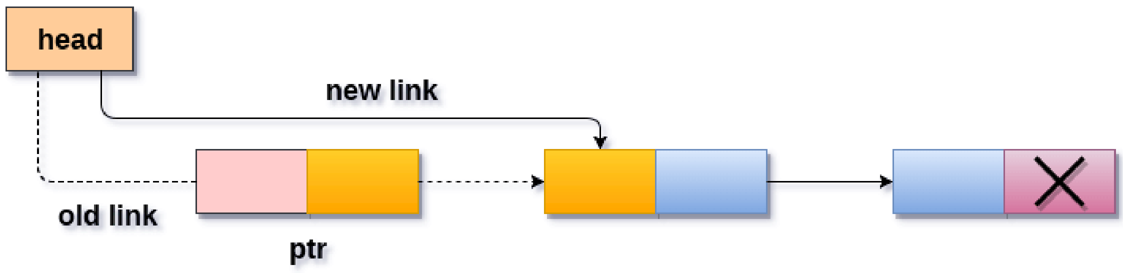
3. Set head of the next node to head

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

4. Free the memory reserved by temp variable

```
free(ptr);
```

5. End



2. Deletion at the end of the list:

It involves deleting the last node of the list.

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

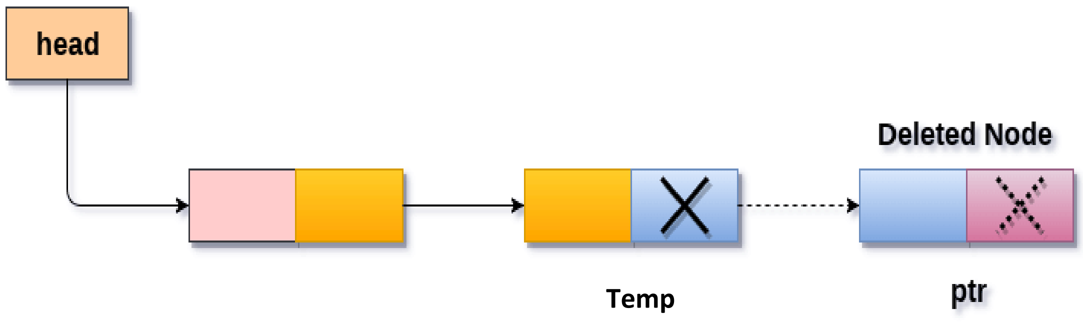
2. Otherwise if (head->next==NULL) then set ptr=head, head=NULL and free ptr. i.e.

```
else if(head->next ==NULL)
{
        ptr=head;
        head=NULL;
        printf("\n The deleted element is:%d \t",ptr->info);
        free(ptr);
}
```

3. Otherwise

```
ptr=head;
while(ptr->next!=NULL)
{
        temp=ptr;
        ptr=ptr->next;
}
temp->next=NULL;
printf("\n The deleted element is:%d \t",ptr->info);
free(ptr);
```

4. End



3. Deletion of specified node:

It involves deletion of the specified node in the list. We need to skip the desired number of nodes to reach the node which will be deleted.

1. If head=NULL print empty list and exit i.e.

```
if(head==NULL)
{
        printf("\n The List is Empty: \n");
        exit(0);
}
```

2. Otherwise
   i. Enter the position pos of the node to be deleted
   ii. If pos=0
      i. Set ptr=head and head=head->next and free ptr i.e.

```
ptr=head;
head=head->next ;
printf("\n The deleted element is:%d \t",ptr->info  );
free(ptr);
```

   iii. Otherwise
      i. Set

```
ptr=head;
for(i=0;i<pos;i++)
 {
  temp=ptr;
  ptr=ptr->next ;
        if(ptr==NULL)
        {
                printf("\n Position not Found: \n");
                return;
        }
 }
```

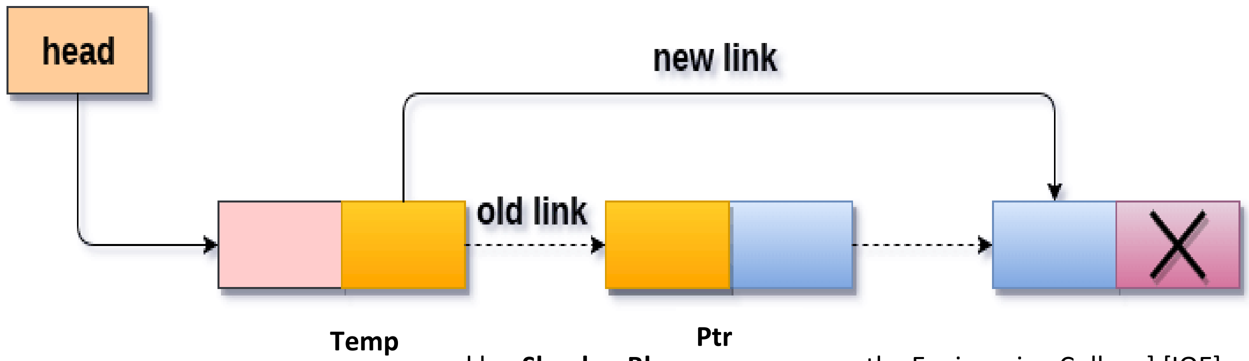
      ii. Set

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

      iii. Free ptr i.e.

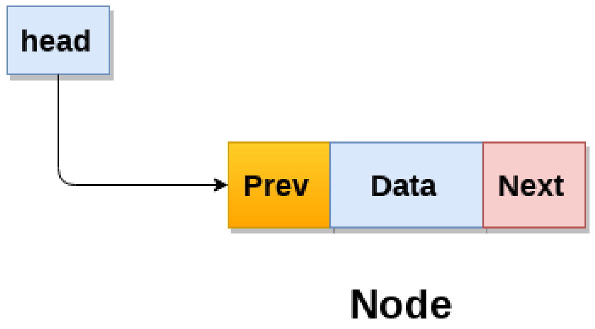
```
free(ptr);
```

3. End

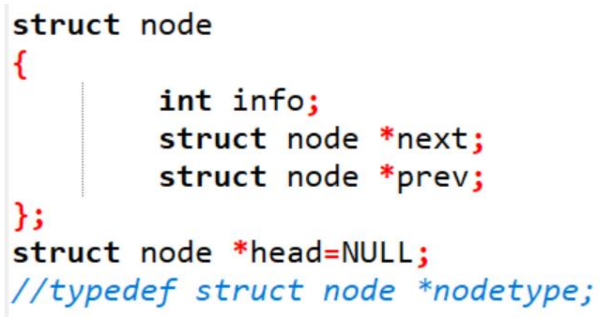


Doubly Linked list:

A doubly linked list is one in which all nodes are linked together by multiple number of links which helps in accessing both the successor node and predecessor node for the given node position. It is bi-directional traversing. Each node in a doubly linked list has two pointer fields and one data field. The pointer fields are used to point successor and predecessor node.



Representation in C:



```
struct node
{
        int info;
        struct node *next;
        struct node *prev;
};
struct node *head=NULL;
//typedef struct node *nodetype;
```

Chapter 4 Linked list

Operations in linked list:

Insertion:

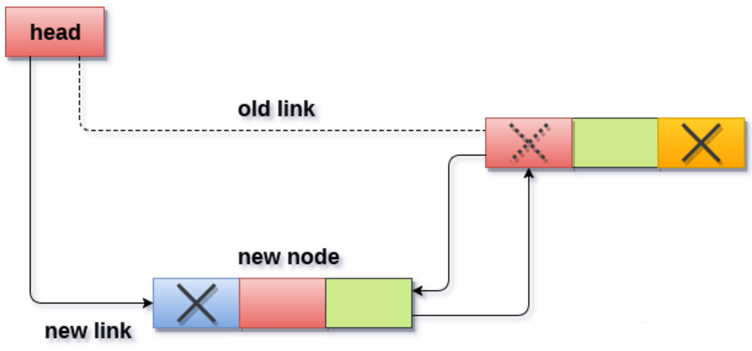
The insertion into a doubly linked list can be performed at different positions.

1. **Insertion at beginning:**

   It involves inserting any element at the front of the list. We just need to make the new node as the head of the list.

   Algorithm:

```
1. Create a node using malloc function
      newnode=(struct node *)malloc(sizeof(struct node));
2. Assign data to info field of new node
      newnode->info = data;
3. Set
      newnode->prev =NULL;
4. Set
      newnode->next=NULL;
5. if head is NULL then set head=newnode
                  head=newnode;
6. otherwise
      i.   Set next of newnode to head
                newnode->next=head;
      ii.  Set prev of head to newnode
                head->prev =newnode;
      iii. Set the head pointer to point to the new node
                head=newnode;
7. End
```



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.

Algorithm:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set prev of newnode to NULL

```
newnode->prev =NULL;
```

4. Set next of newnode to NULL

```
newnode->next =NULL;
```

5. if head is NULL then set head=newnode and exit
6. Otherwise
    i. Set

    ```
    ptr=head;
    ```

    ii. Find the last node

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

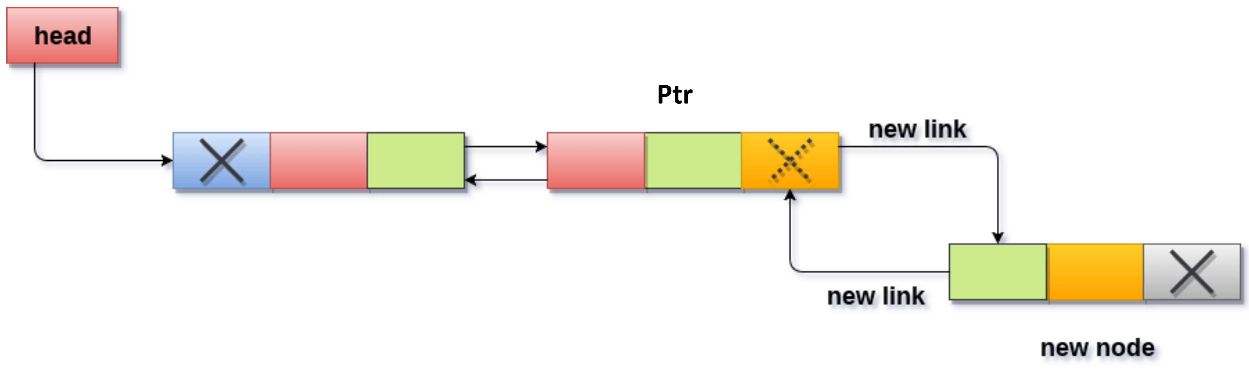
    iii. Set ptr->next =newnode

    ```
    ptr->next =newnode;
    ```

    iv. Set

    ```
    newnode->prev=ptr;
    ```

7. End



3. Insertion at specified position:

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

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. Set prev of newnode to NULL

```
newnode->prev =NULL;
```

4. Set next of newnode to NULL

```
newnode->next =NULL;
```

5. Enter the position of a node at which you want to insert a newnode.
6. Let the position be pos
7. Set

```
ptr=head;


for(i=0;i<pos-1;i++)
{ ptr=ptr->next;
        if(ptr==NULL)
        {
                printf("\nPosition not found:[Handle with care]\n");
                return;
        }
}
```

8. Set

```
newnode->next =ptr->next ;
```

9. Set

```
newnode->prev=ptr;
```

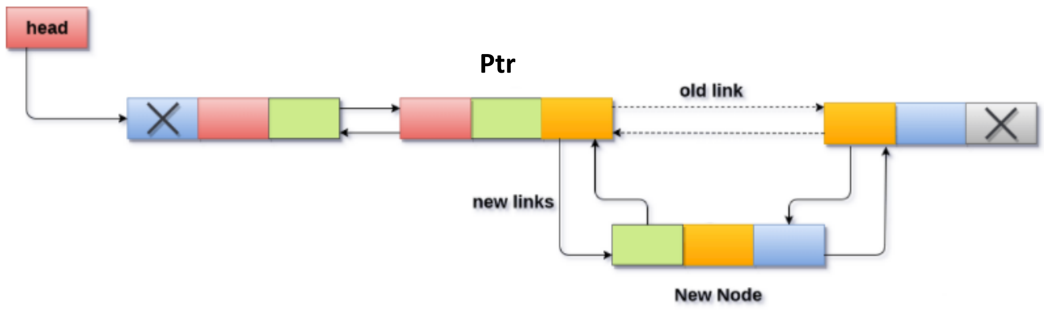
10. Set

```
ptr->next->prev=newnode;
```

11. Set

```
ptr->next=newnode;
```

12. End



Deletion:

1. Deletion at beginning:

It involves deletion of a node from the beginning of the list.

Let head be the pointer to the first node in the linked list

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise store the address of the first node in temporary variable ptr

```
ptr=head;
```

3. Set head of the next node to head

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

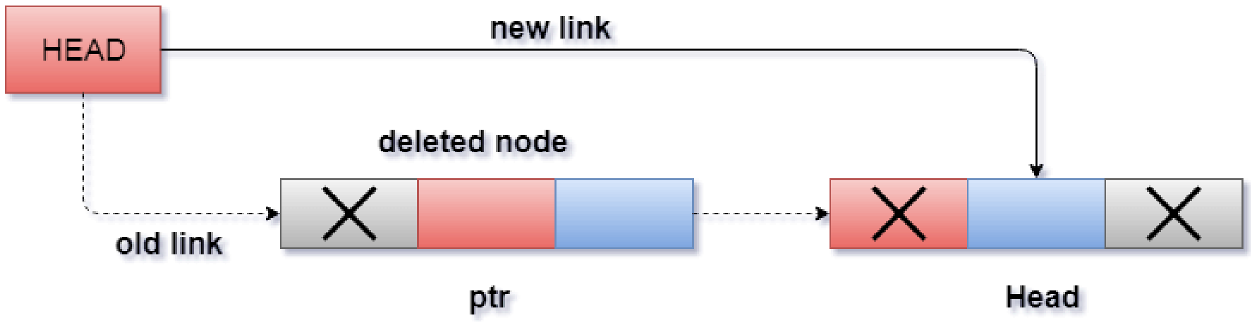
4. Set

```
head->prev=NULL;
```

5. Free the memory reserved by temp variable

```
free(ptr);
```

6. End



2. Deletion at the end of the list:

It involves deleting the last node of the list.

1. If (head==NULL) then print void deletion and exit i.e.

```
if(ptr==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

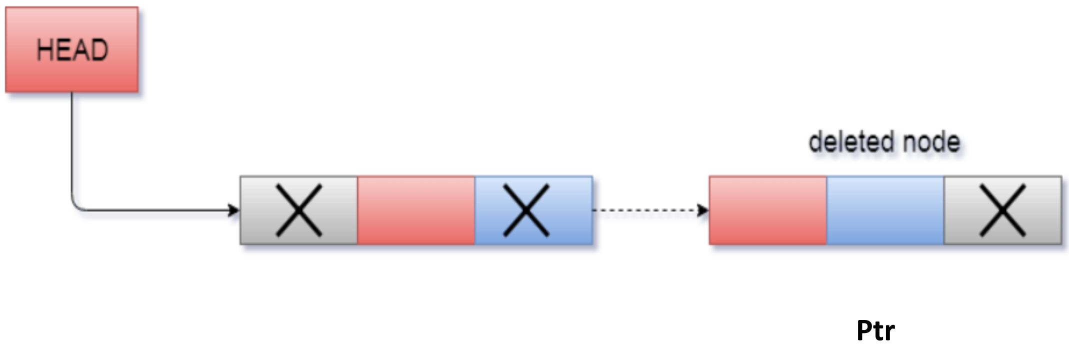
2. Otherwise if (head->next==NULL) then set ptr=head, head=NULL and free ptr. i.e.

```
else if(head->next ==NULL)
{
        ptr=head;
        head=NULL;
        printf("\n The deleted element is:%d \t",ptr->info);
        free(ptr);
}
```

3. Otherwise

```
ptr=head;
while(ptr->next!=NULL)
{
        ptr=ptr->next;
}
ptr->prev->next=NULL;
printf("\n The deleted element is:%d \t",ptr->info);
free(ptr);
```

4. End



4. Deletion of specified node:

It involves deletion of the specified node in the list. We need to skip the desired number of nodes to reach the node which will be deleted.

1. If head=NULL print empty list and exit i.e.

```
if(head==NULL)
{
        printf("\n The List is Empty: \n");
        exit(0);
}
```

2. Otherwise
3. Enter the position pos of the node to be deleted
4. If pos=0

```
ptr=head;
head=head->next ;
head->prev=NULL;
printf("\n The deleted element is:%d \t",ptr->info  );
free(ptr);
```

5. Otherwise
   i. Set

```
ptr=head;
for(i=0;i<pos;i++)
 {
  ptr=ptr->next ;
        if(ptr==NULL)
        {
                printf("\n Position not Found: \n");
                return;
        }
}
```

   ii. Set

```
ptr->prev->next =ptr->next ;
```

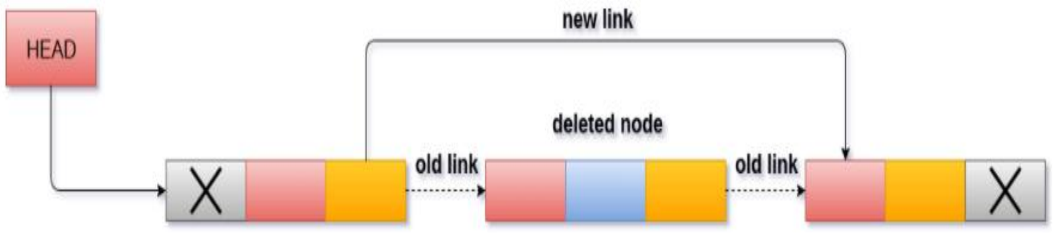
   iii. Set

```
ptr->next->prev=ptr->prev;
```

   iv. Free ptr i.e.

```
free(ptr);
```

6. End



Ptr

Chapter 4 Linked list

Advantages of Doubly linked list:

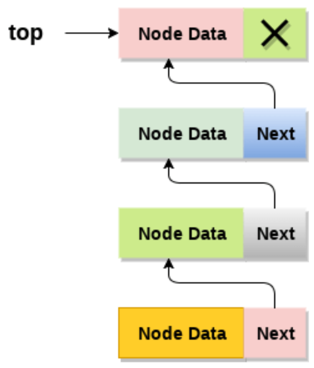
- Reversing the doubly linked list is very easy.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only required the pointer which is to be deleted.

Disadvantages of Doubly linked list:

- It uses extra memory when compared to the array and singly linked list.
- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.

Linked stacks and queues

A stack can be easily implemented through the linked list. In stack Implementation, a stack contains a top pointer. Which is "head" of the stack where pushing and popping items happens at the head of the list. Each node contains a pointer to its immediate successor node in the stack.



Stack

Push Operation:

It is similar to the insertion at the beginning of the link list. It involves inserting any element at the beginning of the list.

1. Create a node using malloc function

   ```
   newnode=(struct node *)malloc(sizeof(struct node));
   ```

2. Assign data to info field of new node

   ```
   newnode->info = data;
   ```

3. Set

   ```
   newnode->next =NULL;
   ```

4. if top == NULL then set head=newnode and exit

   ```
   top=newnode;
   ```

5. otherwise
   1. Set next of newnode to top

      ```
      newnode->next=top;
      ```

   2. Set the top pointer to point to the new node

      ```
      top=newnode;
      ```

6. Print item pushed
7. End

Pop Operation:

It is similar to the deletion from the beginning of the link list. It involves deletion of a node from the beginning of the list.

Let top be the pointer to the first node in the linked list

1. If (top==NULL) then print void deletion and exit i.e.

```
if(top==NULL)
{
        printf("\nList is Empty:\n");
        return;
}
```

2. Otherwise
   i. store the address of the first node in temporary variable ptr

   ```
   ptr=top;
   ```

   ii. Set top of the next node to top

   ```
   top=top->next ;
   ```

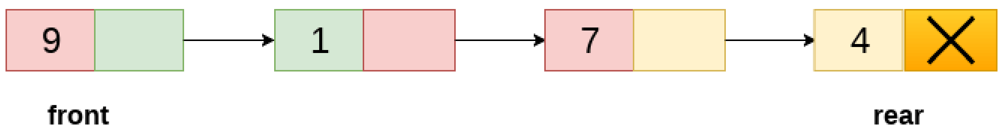
   iii. Free the memory reserved by temp variable

   ```
   free(ptr);
   ```

3. End

Linked list as 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.



Enqueue Operation:

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

```
1. Allocate the memory for the new node
     newnode=(struct node *)malloc(sizeof(struct node));
2. Assign data to info field of new node
        newnode->info = data;
3. If front=NULL then
     i.   Set
          front=newnode;
          rear=newnode;
     ii.  Set
          front->next=NULL;
          rear->next=NULL;
4. Otherwise
     i.   Set
          rear->next = newnode;
          rear = newnode;
          rear->next = NULL;
5. End
```

Deque Operation:

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not.

```
1. If front=NULL
     i. Print underflow and exit i.e.
        printf("\nUNDERFLOW\n");
        return;
2. Otherwise
        ptr = front;
        front = front -> next;
        free(ptr);
3. End
```

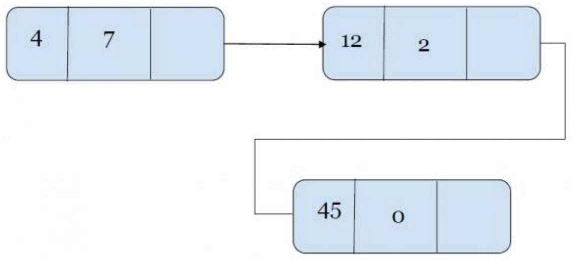
Adding two polynomials using Linked List:

In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

A linked list that is used to store Polynomial looks like –

Polynomial: $4x^7 + 12x^2 + 45$



We start with the term which is of highest degree in any of the polynomials. If there is no item having same exponent, we simply append the term to the new list, and continue with the process. Wherever we find that the exponent match, we simply add the coefficients and then store the term in the new list. If one list gets exhausted earlier and the other list still contains some lower order terms than simply append the remaining terms to the new list.

Example:

Input:

p1= $13x^8 + 7x^5 + 32x^2 + 54$

p2= $3x^{12} + 17x^5 + 3x^3 + 98$

Output:

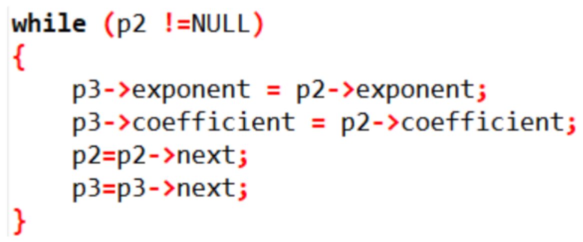
P3= $3x^{12} + 13x^8 + 24x^5 + 3x^3 + 32x^2 + 152$

Algorithm:

Let phead1, phead2 and phead3 represent the pointer of the three lists under consideration. We want to add phead1 and phead2, and store the result in phead3. This addition can be performed using the procedure below.

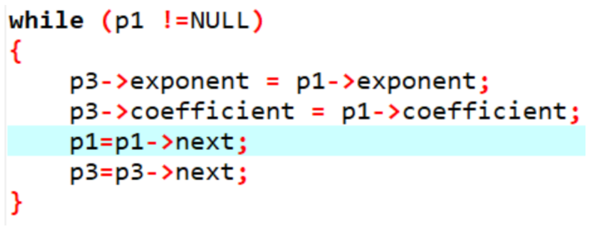
p1=phead1; p2=phead2; p3=phead3;

1. If both the polynomials are null then return
2. else if **p1 = Null** then



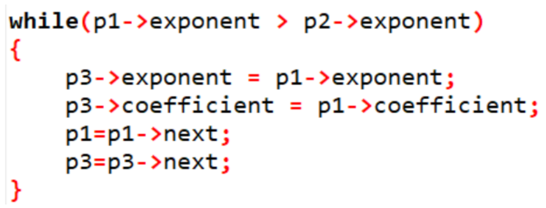
```
while (p2 !=NULL)
{
    p3->exponent = p2->exponent;
    p3->coefficient = p2->coefficient;
    p2=p2->next;
    p3=p3->next;
}
```

3. Else if **p2 = Null** then



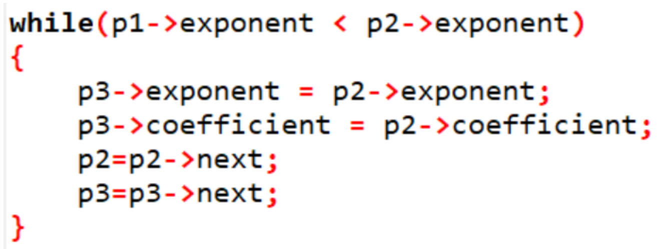
```
while (p1 !=NULL)
{
    p3->exponent = p1->exponent;
    p3->coefficient = p1->coefficient;
    p1=p1->next;
    p3=p3->next;
}
```

4. Else
   a. case 1



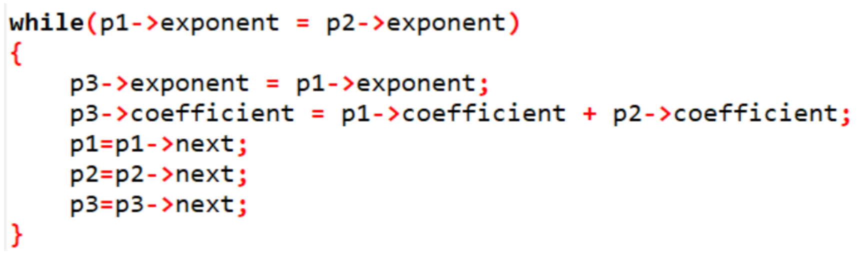
```
while(p1->exponent > p2->exponent)
{
    p3->exponent = p1->exponent;
    p3->coefficient = p1->coefficient;
    p1=p1->next;
    p3=p3->next;
}
```

   b. case 2



```
while(p1->exponent < p2->exponent)
{
    p3->exponent = p2->exponent;
    p3->coefficient = p2->coefficient;
    p2=p2->next;
    p3=p3->next;
}
```

   c. Case 3



```
while(p1->exponent = p2->exponent)
{
    p3->exponent = p1->exponent;
    p3->coefficient = p1->coefficient + p2->coefficient;
    p1=p1->next;
    p2=p2->next;
    p3=p3->next;
}
```

5. End

Circular linked list

It is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

Advantages of Circular Linked Lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of queue.
3. Circular linked list are useful in applications to repeatedly go around the list like CPU scheduling.

Operations:

Insertion:

At the beginning:

1. Create a node using malloc function

```
newnode=(struct node *)malloc(sizeof(struct node));
```

2. Assign data to info field of new node

```
newnode->info = data;
```

3. If list is empty i.e.

```
if(head==NULL)
{
        head=newnode;
        newnode->next=head;
}
```

4. Otherwise

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

ptr->next =newnode;
head=newnode;
```

5. End