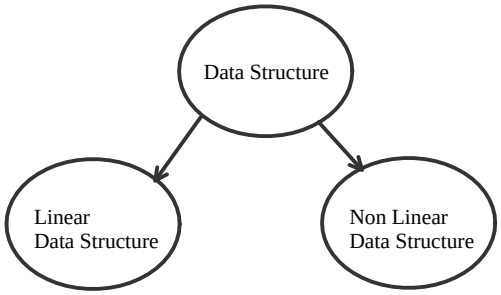
# Linear and Non Linear Data Structure, Polynomial using arrays

14 September 2020 20:43

**Data Structure :**

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an efficient way.




| | | |
|---|---|---|
| **Example** | Arrays<br>Stack<br>Queue<br>Linked List | Trees<br>Graphs |

| Linear Data Structure | Non Linear Data Structure |
|---|---|
| In the linear data structure, the data is organized in a linear order in which elements are linked one after the other sequentially. | In the non-linear data structure the data elements are not stored in a sequential manner rather the elements are hierarchically related. |
| The data elements can be accessed in single run | traversing of data elements in one run is not possible |
| Linear data structures are easier to implement. | Non-linear data structures are difficult to implement as compared to linear data structures. |
| Implementation is easy, as elements will be in sequential order | Implementation will be complex when compare to linear data structure |
| memory utilization is not efficient | Memory will be utilized very efficiently |
| Examples include<br>Array,<br>stack,<br>queue,<br>linked list, etc. | Examples include<br>Tree,<br>Graph |

- ✓ An **Array** is a collection of data elements having similar data types where memory is allocated in sequential order
- ✓ A **Linked list** is a collection of data elements, where each node has a two units
  - The data
  - A reference(address) to the next node in the sequence.
- ✓ **Stack** is a **LIFO** (Last In First Out) data structure where
  - Last Inserted Element will be deleted first
  - First inserted element will be deleted last.
- ✓ **Queue** is a **FIFO** (First in First Out) data structure where
  - Element that was inserted first will be deleted first
  - Element that was inserted Last will be deleted Last
- ✓ A **Tree** is a collection of nodes where first level contains Single node called root, it is connected to other nodes and forms a parent-child relationship.
- ✓ A **Graph** is a collection of vertices and edges. Each vertex in a graph can be connected to more than one vertex. The connectivity between two vertices is represented by an Edge

## Arrays

Arrays is a data structure used to store elements of the same type in sequential order. All the data elements can be accessed with same array name differed by their indices.

**Declaration** :

dataType arrayName[arraySize];

Ex - int Ary1[5]; float Ary2[6]; char Ary3[10];

↓

We declared an array Ary1(name of the array) which is of type int, it can store 5 integer elements

**Initialization**

We can initialize the array at the time of declaration itself as below

int mark[5] = {19, 10, 8, 17, 9}; ( or ) int mark[] = {19, 10, 8, 17, 9};

↓

Here based on no of elements assigned, the size of array will taken internally

If we don't know the array value at the time of declaration, we will just declare the array with its size and later we can initialize the values at run time

```
int ary[5];
int k = 1;
for(i=0; i <= 4; i++) // loop terminates when i value becomes 4
{
    a[i] = k;
    k++;
}
```

```
int ary[5];
int k = 1;
for(i=0; i < 5; i++) // Here also loop terminates when i value becomes 4
{
    a[i] = k;
    k++;
}
```

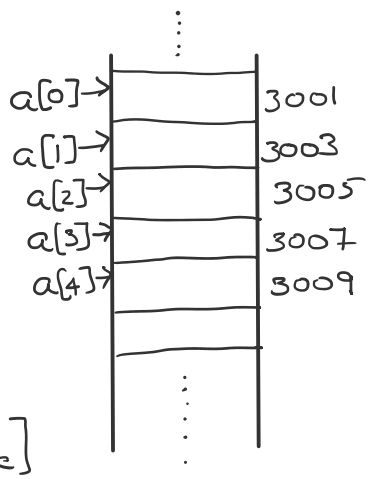
**Accessing array elements**

arrayName[index]

We can access array elements by indices

```
int main()
{
    int a[5] = {2,4,6,8,10};
    a[0] = 10;
    a[2] = 20;
}
```

| Index | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Value | 2 | 4 | 6 | 8 | 10 |




address for a[3] = Base address + offset [size of datatype]

= 3001 + 3[2]

= 3007

( Base address = &a = &a[0] = 3001,

offset = index subscript

= 3 )

**Polynomial Representation Using Arrays**

In Array representation we consider the exponents of the given expression are arranged from 0 to the highest degree, which is represented by the index of the array. The coefficients of the respective exponent are assigned at appropriate index in the array.

Example :

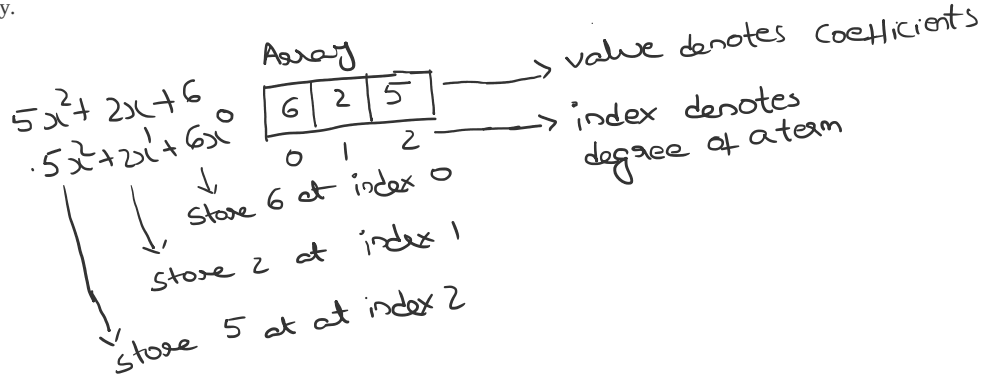
$5X^2 + 2X^1 + 6X^0$

int A[3];

A[0] = 6

A[1] = 2

A[2] = 5



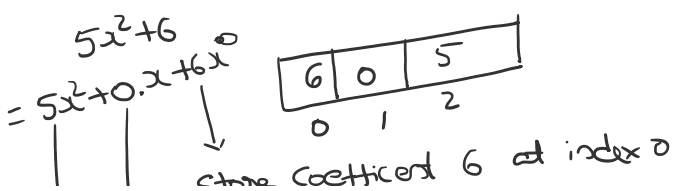

$5X^2 + 6X^0$

int A[3];

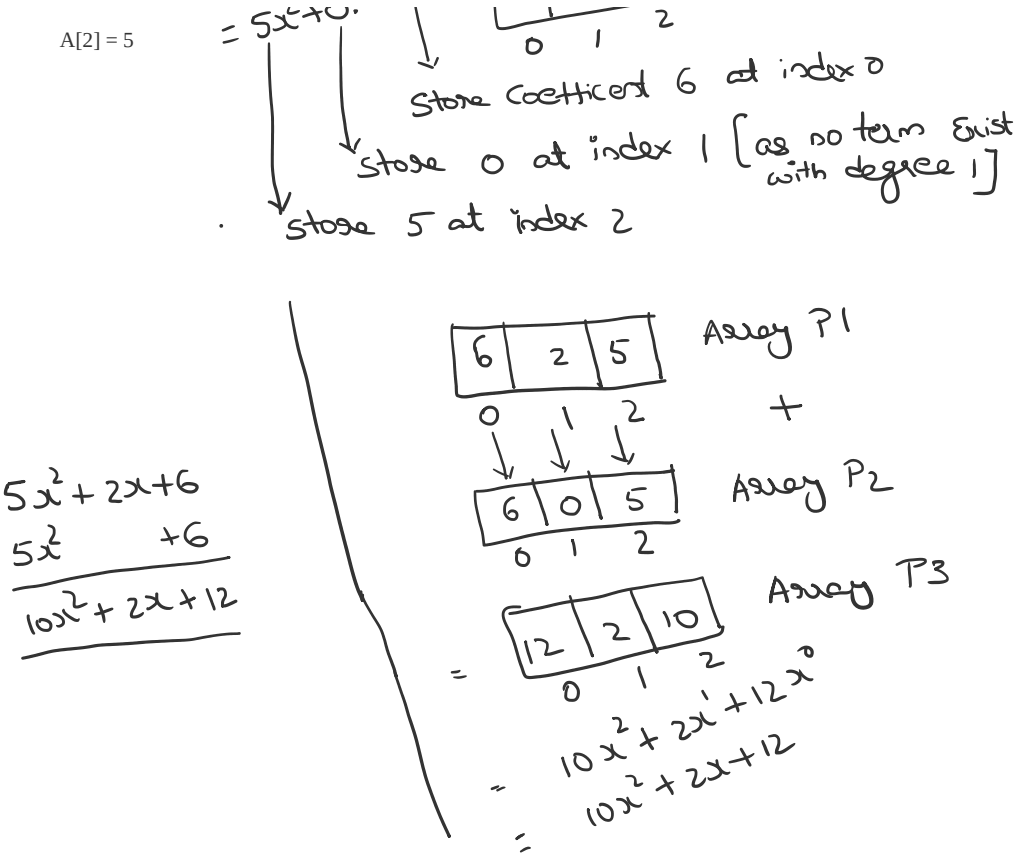
A[0] = 6

A[1] = 0

A[2] = 5



A[2] = 5




```
// Adding two polynomials using Arrays

#include<stdio.h>
#define MAX 10
void print(int []);
void init(int []);
void read(int []);
void add(int [], int [], int[]);
int main()
{
        int p1[MAX], p2[MAX], p3[MAX];
        init(p1);
        init(p2);               // intialize array elements with Zero values
        init(p3);
        printf("\n Enter 1st polynomial\n");
        read(p1);               // read all the terms in first polynomial
        printf("\n Enter 2nd polynomial\n");
        read(p2);               // read all the terms in second polynomial
        add(p1,p2,p3);
        return 0;
}
void init(int p[])
{
   int i;
   for(i=0;i<MAX;i++)
        {
                p[i]=0;
        }
}
void read(int p[])
{
        int n, i, power,coeff;
        printf("\n Enter number of terms :");
        scanf("%d",&n);
        /* read n terms */
        for (i=0;i<n;i++)
        {
                printf("\n enter a term(coeff  power)");
                scanf("%d %d",&coeff,&power);
                p[power]=coeff;
        }
}
void add(int p1[], int p2[], int p3[])
{
   int i;
   for(i=MAX-1;i>=0;i--)
        {

// Adding Polynomial coefficients in Arrays P1 and P2  and storing in Array P

                p3[i]=p1[i]+p2[i];
```

```
        // Adding Polynomial coefficients in Arrays P1 and P2  and storing in Array P

                p3[i]=p1[i]+p2[i];
        }
        print(p3);
}
void print(int p[])
{
        int i;
        printf("\nAfter adding Two Polynomials result is\n" );
        for(i=MAX-1;i>=0;i--)
        {
                if(p[i]!=0)
                {
                        printf("%dX^%d   ",p[i],i);
                }
        }
}
```

OUTPUT:

```
Enter 1st polynomial
 Enter number of terms :2

 enter a term(coeff  power)3 2
 enter a term(coeff  power)4 1


 Enter 2nd polynomial
 Enter number of terms :3

 enter a term(coeff  power)4 3
 enter a term(coeff  power)2 2
 enter a term(coeff  power)5 0

After adding Two Polynomials result is
4X^3   5X^2   4X^1   5X^0
```

# Sparse Matrix

22 September 2020 11:34

While representing Matrix as a two-dimensional array with m rows and n columns, having total m x n values. If most of the elements(atleast half) in the matrix have **0 Values**, then it is called a sparse matrix.

Storage space :

In case of sparse matrix, the memory occupied to represent zero elements will lead to inefficient memory utilization

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 4 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 3 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 6 | 0 | 0 | 0 |
| 3 | 0 | 12 | 0 | 0 | 0 | 0 |

4 Rows

4 bytes int element size
rows cols elements
4 x 6 = 24 * 4 = 96 bytes

Row col val
Triplet

## Triplet Representation

From the sparse matrix, we consider only non-zero values along with their row and column index. In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

| | Row | Col | Val |
|---|---|---|---|
| 0 | 4 | 6 | 5 |
| 1 | 0 | 0 | 4 |
| 2 | 0 | 2 | 1 |
| 3 | 1 | 1 | 3 |
| 4 | 2 | 2 | 6 |
| 5 | 3 | 1 | 12 |

No of Row, col, non zero values

Non zero Elements Position & Values

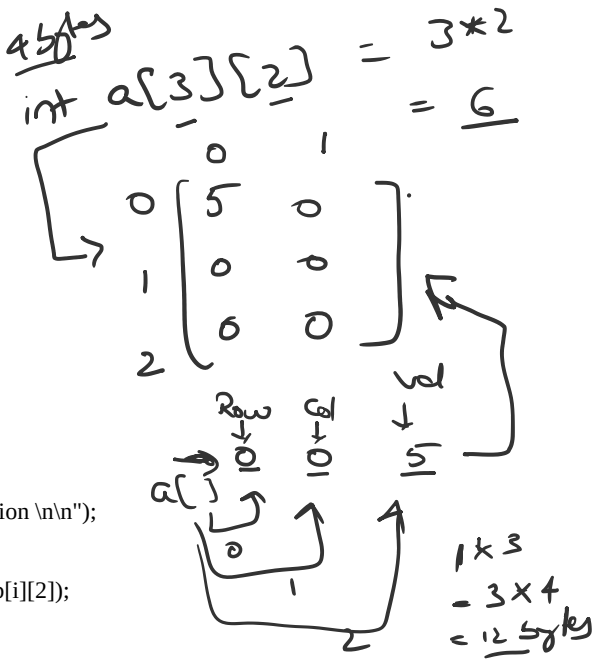
row cols elements size
6 * 3 = 18 * 4 = 72 bytes

Triplet Representation
- Arrays
- Linked List

Code Triplet Representation

```
#include <stdio.h>
#define MAX 20

void read_matrix(int a[10][10], int row, int column);
void print_sparse(int b[MAX][3]);
void create_sparse(int a[10][10], int row, int column, int b[MAX][3]);

int main()
{
   int a[10][10], b[MAX][3], row, column;
   printf("\nEnter the size of matrix (rows, columns): ");
   scanf("%d%d", &row, &column);

   read_matrix(a, row, column);
   create_sparse(a, row, column, b);
   print_sparse(b);
   return 0;
}

void read_matrix(int a[10][10], int row, int column)
{
   int i, j;
   printf("\nEnter elements of matrix\n");
   for (i = 0; i < row; i++)
   {
      for (j = 0; j < column; j++)
      {
         printf("[%d][%d]: ", i, j);
         scanf("%d", &a[i][j]);
      }
   }
}
```

```
void create_sparse(int a[10][10], int row, int column, int b[MAX][3])
{
   int i, j, k;
   k = 1;
   b[0][0] = row;
   b[0][1] = column;
   for (i = 0; i < row; i++)
   {
      for (j = 0; j < column; j++)
      {
         if (a[i][j] != 0)
         {
            b[k][0] = i;
            b[k][1] = j;
            b[k][2] = a[i][j];
            k++;
         }
      }
      b[0][2] = k - 1;
   }
}

void print_sparse(int b[MAX][3])
{
   int i, column;
   column = b[0][2];
   printf("\n Sparse Matrix Triplet representation \n\n");
   for (i = 0; i <= column; i++)
   {
      printf("%d\t%d\t%d\n", b[i][0], b[i][1], b[i][2]);
   }
}
```




6*4
= 24 bytes ✓

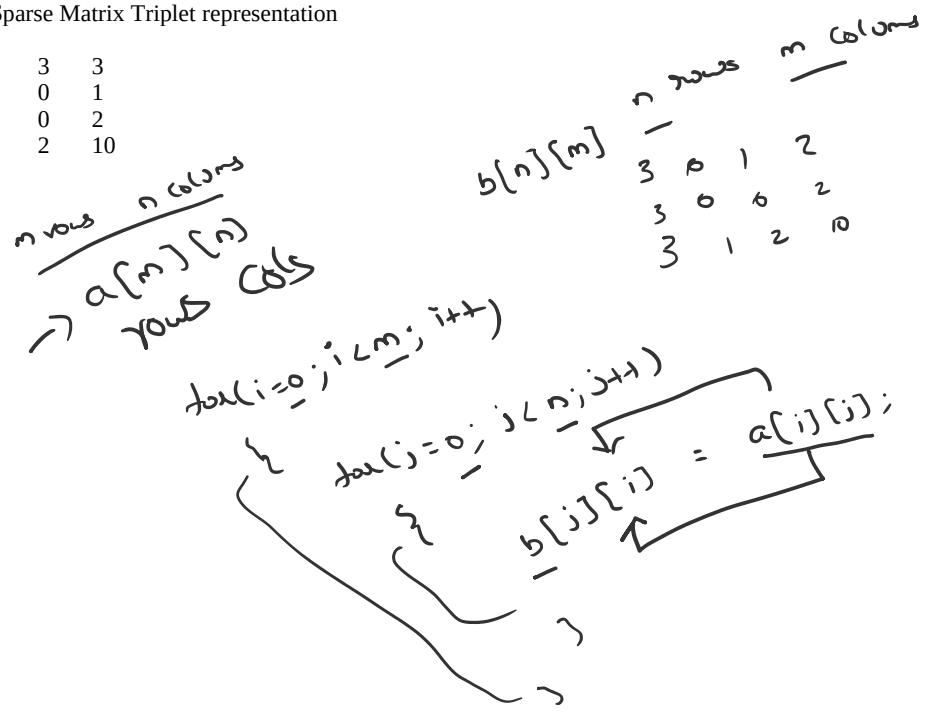
Output:

Enter the size of matrix (rows, columns): 3 3

Enter elements of matrix
[0][0]: 1
[0][1]: 0
[0][2]: 0
[1][0]: 2
[1][1]: 0
[1][2]: 0
[2][0]: 0
[2][1]: 0
[2][2]: 10

Sparse Matrix Triplet representation

```
3    3    3
0    0    1
1    0    2
2    2    10
```




*1 5x3
*11 3x5

a[i][j]
↳ b[j][i]

# UNIT-2
# LINKED LISTS

**Linked List:**

A linked list is a non-sequential collection of data items(not in consecutive memory locations). For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.

**Disadvantages of arrays:**

- The size of the array is fixed, this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at beginning of array is very expensive because existing elements need to be shifted over to make front index free.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 | | |

Now If we wish to insert 1 at 1st position shift all elements to right, which takes lot of time

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

- Deleting an element from an array is not possible(memory can not be freed). Linked lists have the strength, to allocate and deallocate required memory

- Generally array's allocates the memory for all its elements in one block sequentially whereas linked lists doesn't follow sequential memory allocation. Linked lists allocate memory for each element at different locations and only when necessary.

**Advantages of linked lists:**

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not preallocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

**Disadvantages of linked lists:**

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming, as there will be no concept of indexing like in arrays

**Types of Linked Lists:**

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

A single linked list is one in which all nodes are linked together, with only one link from one node to next node in single direction. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by two links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse list in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

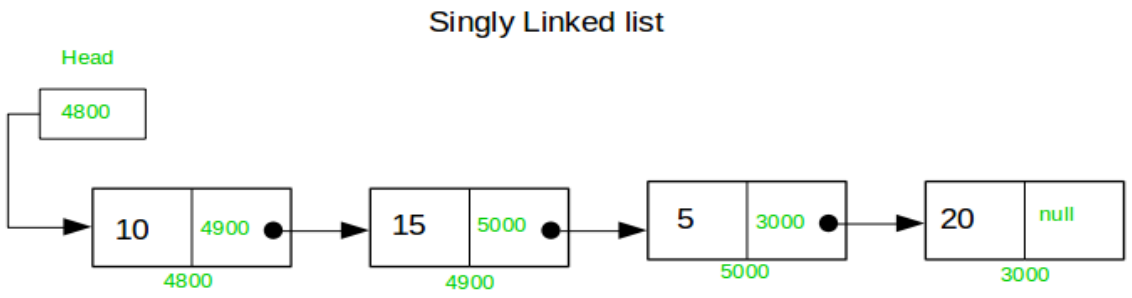
**Comparison between array and linked list:**

| ARRAY | LINKED LIST |
|---|---|
| Size of an array is fixed | Size of a list is not fixed |
| Memory is allocated from stack | Memory is allocated from heap |
| It is necessary to specify the number of elements during declaration (i.e., during compile time). | It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time). |
| It occupies less memory than a linked list for the same number of elements. | It occupies more memory. |
| Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room. | Inserting a new element at any position can be carried out easily. |
| Deleting an element from an array is not possible. | Deleting an element is possible. |

## Single Linked List:

A linked list allocates space for each element separately called a "node". Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated memory using malloc( ), so the node memory continues to exist until it is explicitly de-allocated using free().

Single Linked list




The beginning of the linked list is stored in a "Head" pointer which points to the first node. The first node contains a pointer(*next) to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. We can access any node in the list by starting at the Head and following the next pointers.

**The basic operations in a single linked list are:**

- Creation.
- Insertion.
- Deletion.
- Traversing(Display).

**Creating a node for Single Linked List:**

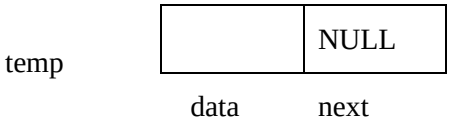
Creating a singly linked list starts with creating a node structure. Sufficient memory has to be allocated for creating a node using dynamic memory allocation.

A Node Structure

```
struct Node
{
        int data;
        struct Node *next;
};
struct Node * Head = NULL;
struct Node *temp = NULL:
```

For insertion, Memory is to be allocated for the new node. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL.

```
struct Node *temp  =  (struct Node *)malloc(sizeof(struct Node);
```




The newly created node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

## 1. Inserting a node at the beginning.

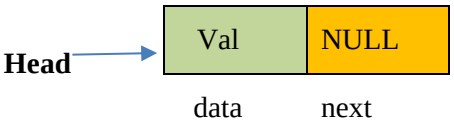
create a new node say, ptr

```
struct node *ptr = (struct node *) malloc(sizeof(struct node *));
sacnf(Val);
ptr → data = Val
```

**case 1:**

List is Empty

```
if(head==NULL)          // If  List is empty new node itself will be the head node
  head = ptr;
```




**Case 2:**

List is not empty

Allocate the space for the new node in the memory. This will be done by using the following statement.

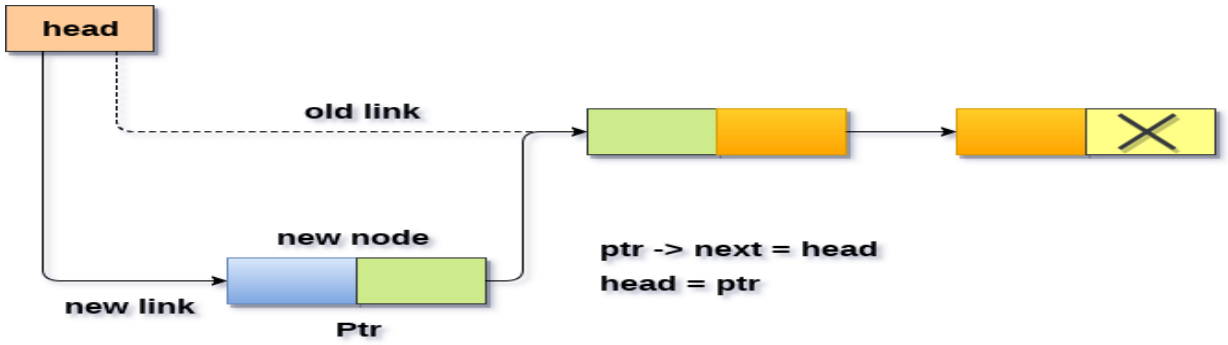
```
ptr = (struct node *) malloc(sizeof(struct node *));
scanf(val)
ptr → data = val;
```

make Head as next node for newly created ptr node

ptr->next = head;

Finally, we need to make the new node as the Head node

head = ptr;




## 2. Inserting a node at the end:

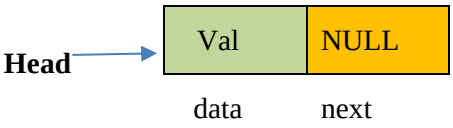
create a new node say, ptr

```
struct Node *ptr = (struct node *) malloc(sizeof(struct node *));
sacnf(Val);
ptr → data = Val;
ptr->next = NULL;
```

**case 1:**

List is Empty

```
if(head==NULL)
  head = ptr;
```




**Case 2:**

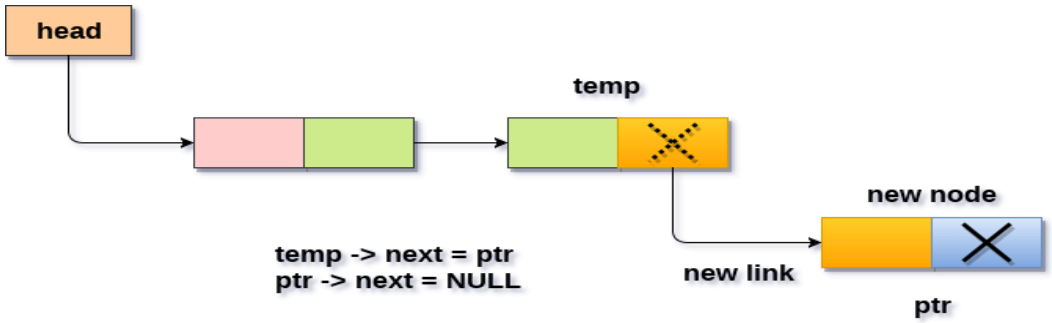
List is not empty

Step 1: Reach last Node in list, store this node addres in temp,

```
struct Node *temp;
temp = head
while (temp -> next != NULL)
{
        temp = temp -> next;
}
```

Step 2: now make last node next(temp->next) point to newly created node ptr

**temp->next = ptr;**




**Inserting node at the last into a non-empty list**

## 3. Inserting a node into the single linked list at a specified intermediate position other than beginning and end.

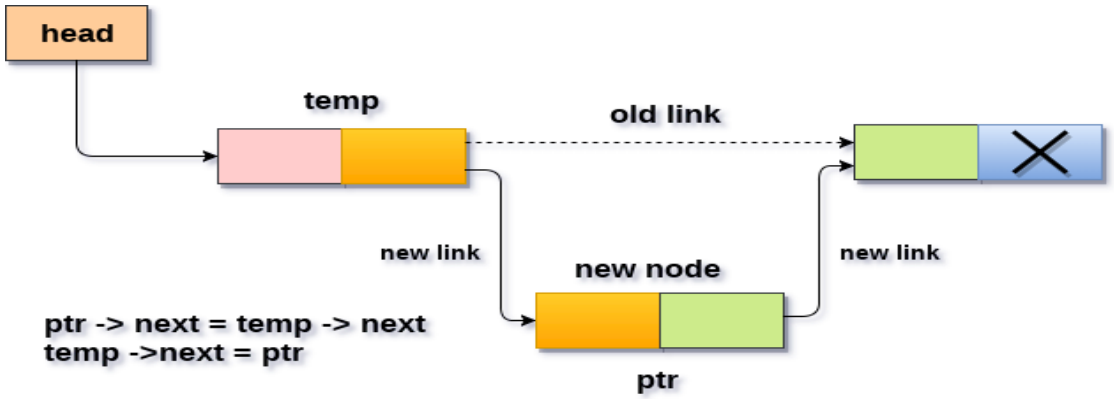
step 1: reach the node(temp) after which we want to add our newly created node(ptr)

```
temp=head;
for(i=0;i<loc;i++)
{
    temp = temp->next;
// temp will become NULL, when loc specified is is greater than no of nodes in in the list
    if(temp == NULL)
    {
        return;
    }
}
```

step 2: make newly created node point to temp next node, later make ptr as temp next node

**ptr->next = temp->next**

**temp -> next = ptr**




**Deletion of a node:**

A node can be deleted from the list from three different places namely.

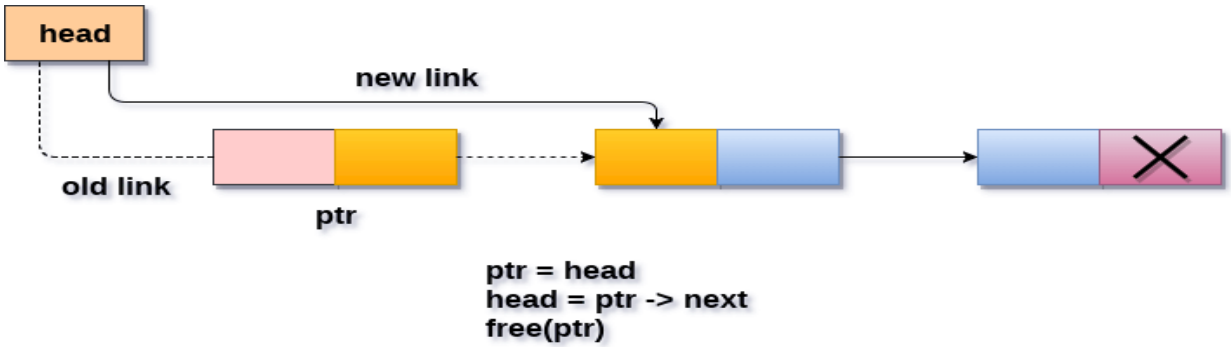
4. Deleting a node at the beginning.
5. Deleting a node at the end.
6. Deleting a node at intermediate position.

Note: Where ever the node is deleted, it is our Responsibility to dellocate/free the memory of that node(so that it can be used by other processes)

## 4. Deleting a node at the beginning:

**Step 1: store the address of very first node in ptr,**
**Step 2: make 2nd node of list as head**
**Step 3: free memory occupied by first node(ptr)**

```
ptr = head;
head = head->next;
free(ptr);
```

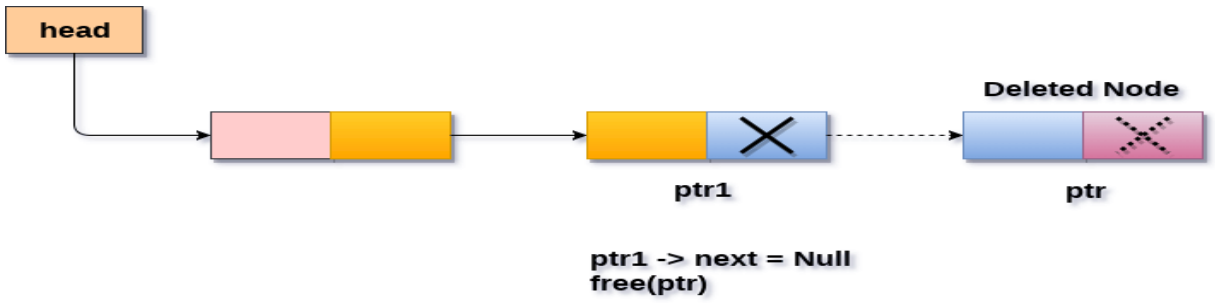



Deleting a node from the beginning

## 5. Deleting a node at the end:

**Step 1: reach 2nd last node of list and store its adress say ptr1**
**Step 2: store last node address at ptr**
**Step 3: free memory occupied by last node(ptr)**

```
ptr1 = head;
while(ptr1->next->next != NULL)
{
   ptr1 = ptr1 ->next;     // ptr1 is the second last node
}
ptr = ptr1->next;          // ptr is the last node
ptr1->next = NULL;
free(ptr);
```

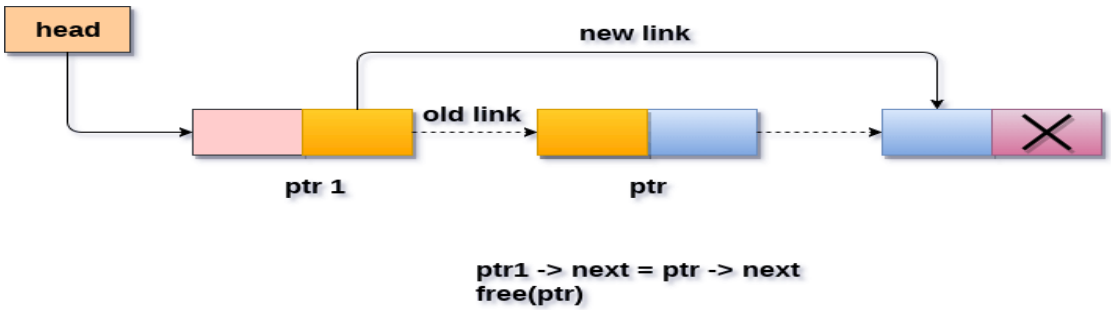



Deleting a node from the last

## 6. Deleting a node at Intermediate position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

Reach the node whose next node has to be deleted,
we stand at node called ptr1, and we want to delete its next node
ptr1->next is ptr(we want to delete ptr)




Deletion a node from specified position

To delete ptr

```
ptr1->next = ptr->next     // attach remaining list of nodes after ptr to ptr1
free(ptr);
```

**Traversal and displaying a list (Left to Right):**

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

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

## Double Linked List:

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi- directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

**The basic operations in a double linked list are:**

- Creation.
- Insertion.
- Deletion.
- Traversing.

The beginning of the double linked list is stored in a "Head" pointer which points to the first node. The first node‟s left link and last node‟s right link is set to NULL.

**Creating a node for Double Linked List:**

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for

creating a node.

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

## Inserting a node at the beginning:

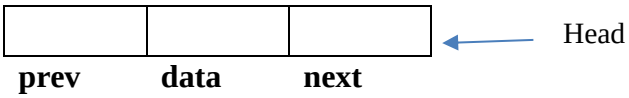
Allocate the space for the new node in the memory. This will be done by using the following statement.

```
ptr = (struct node *)malloc(sizeof(struct node));
ptr -> next = NULL;
ptr -> prev = NULL;
scanf(val);
ptr->data = val;
```

### Case 1: list is empty (head is NULL)

make newly created node itself as Head node

```
head=ptr;
```




### Case 2: List is not empty

If list is not empty, store adress of Head under newly created node, ptr->next

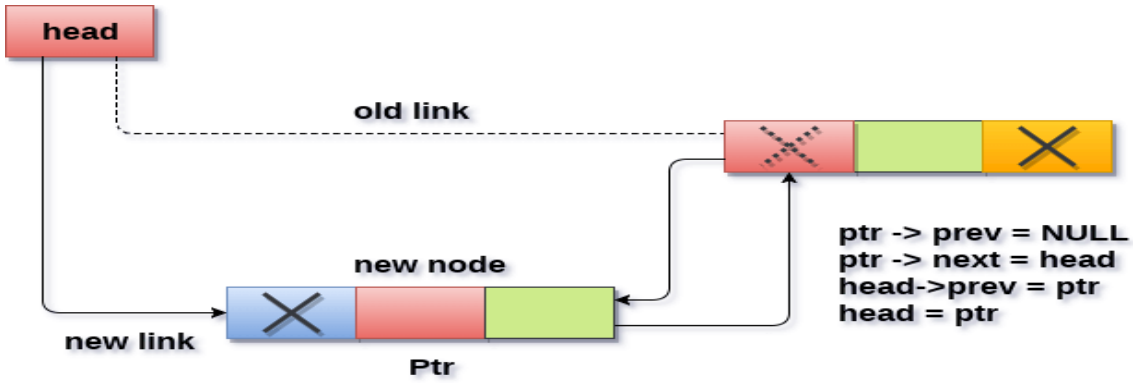
```
ptr->next = head;
```

store address of newly created node under Head->prev

```
head→prev=ptr;
```

Finally make the newly attached node as head of the list

```
head = ptr;
```




**Insertion into doubly linked list at beginning**

## Inserting a node at the end:

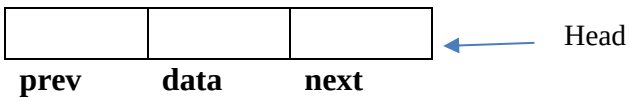
Allocate the space for the new node in the memory. This will be done by using the following statement.

```
ptr = (struct node *)malloc(sizeof(struct node));
ptr -> next = NULL;
ptr -> prev = NULL;
scanf(val);
ptr->data = val;
```

**Case 1: list is empty (head is NULL)**

make newly created node itself as Head node

head=ptr;




**Case 2: List is not empty**

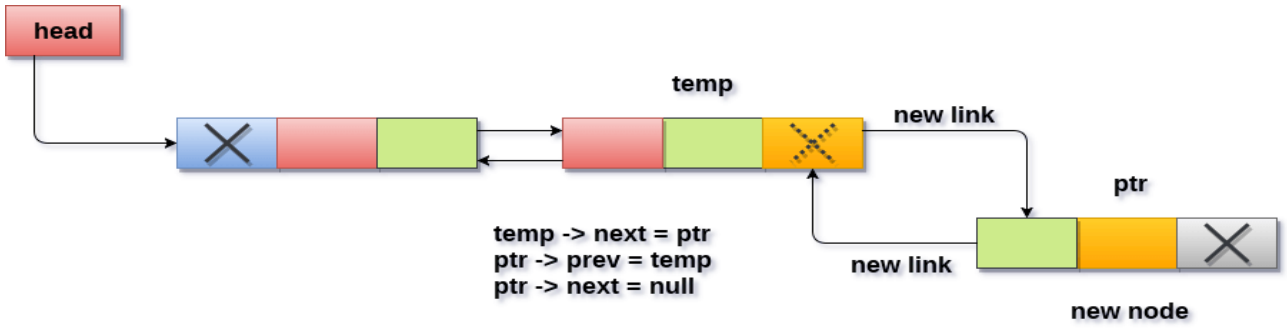
we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
temp = head;
while (temp->next != NULL)
```

```
{
        temp = temp → next;
}
```

After terminating from while loop temp will be pointing to last node of list, attach newly created node(ptr) as next node of temp. then ptr will be added at end of the list

```
temp→next =ptr;
ptr → prev = temp;
```




**Insertion into doubly linked list at the end**

## Inserting a node at an intermediate position:

Allocate the memory for the new node.

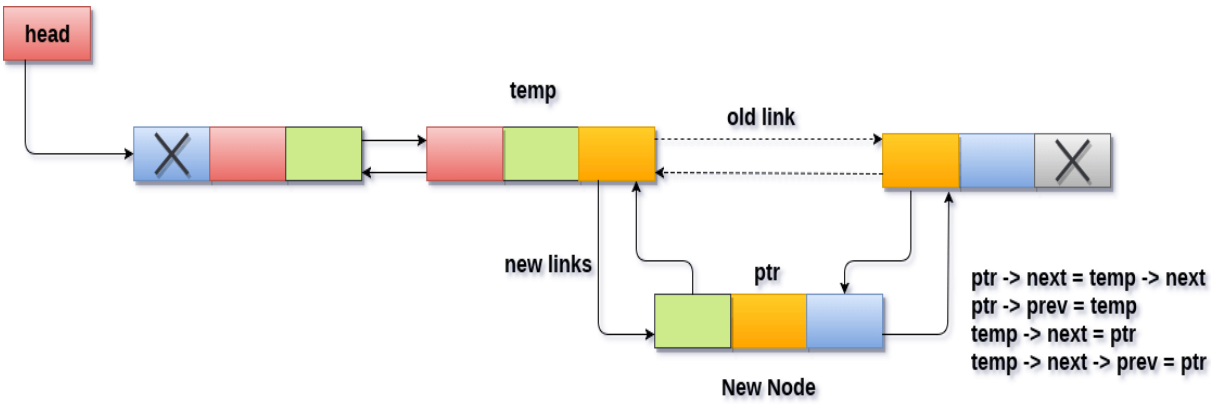
```
ptr = (struct node *)malloc(sizeof(struct node));
ptr -> next =NULL;
ptr -> prev = NULL;
```

reach the specified node, after which the new node has to be inserted using any pointer say temp.

```
temp=head;
for(i=0;i<loc;i++)
{
        temp = temp->next;
// temp will become NULL, when loc specified is is greater than no of nodes in in the list
        if(temp == NULL)
        {
                return;
        }
}
```

now add newly created node ptr, after temp

```
ptr → next = temp → next;   // make list after temp as next of new node ptr
ptr → prev = temp;          // new node prev list will be temp ny this
temp → next = ptr;          //temp next now onwards will start from new node ptr
temp → next → prev = ptr;   // make the previous pointer of the next node of temp
                               point to the new node.
```




**Insertion into doubly linked list after specified node**

## Deleting a node at the beginning:

Store the address under head pointer to another pointer say ptr, and shift the head pointer to its next node.

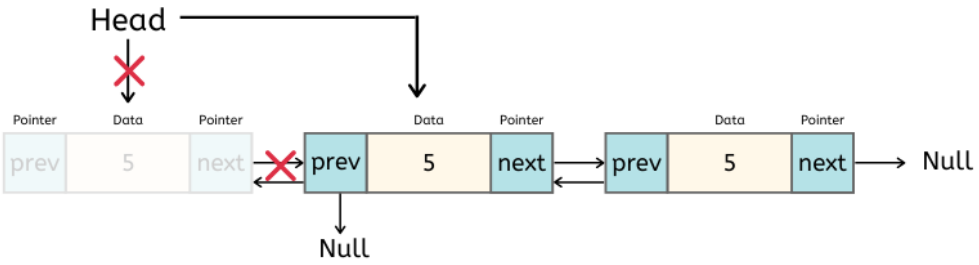
```
Ptr = head;
head = head → next;
```

make the prev of this new head node point to NULL

```
head → prev = NULL;
```

Now free the pointer ptr by using the **free** function.

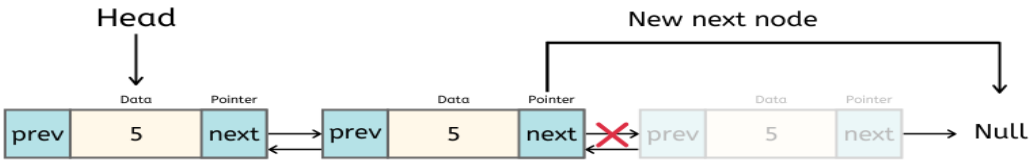
```
free(ptr);
```



## Deleting a node at the end:

**Step 1: reach 2nd last node of list and store its adress say ptr1**
**Step 2: store last node address at ptr**
**Step 3: free memory occupied by last node(ptr)**

```
ptr1 = head;
while(ptr1->next->next != NULL)
{
   ptr1 = ptr1 ->next;     // ptr1 is the second last node
}
ptr = ptr1->next;          // ptr is the last node
ptr1->next = NULL;
free(ptr);
```




## Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function traverse_left_right() is used for traversing and displaying the information stored in the list from left to right.

## Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function traverse_right_left() is used for traversing and displaying the information stored in the list from right to left.