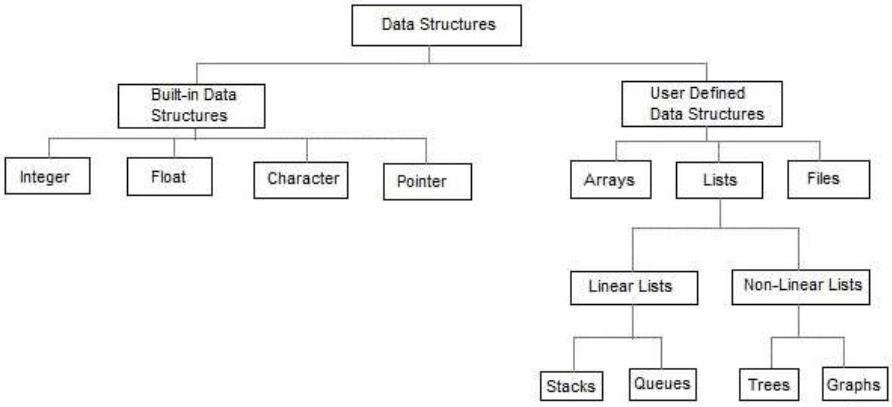
MODULE 1

Introduction: Concept of Data Structures, Types of Data Structures, Linear versus Non-Linear Data Structures, Data Structure Operations Array: Linear Array-Memory representation, insertion and deletion operation, **Multidimensional Arrays**-memory representation, Sparse Matrices. Linked List: Concept of Linked List, Memory representation, **Single Linked List** - Traversing, Searching, Insertion, Deletion, **Circular Header Linked List, Doubly Linked** List - Insertion, Deletion, Difference of Linked List and Array. (20hr)

Introduction to Data Structures

- Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.
- Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.
- Data Structure is a systematic way to organize data in order to use it efficiently.



What is an Algorithm?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as pseudocode or using a flowchart. An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

- **Time Complexity**
  Time Complexity is a way to represent the amount of time needed by the program to run to completion.
- **Space Complexity**
  It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components:

Instruction Space: It's the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.

Data Space: It's the space required to store all the constants and variables value.

Environment Space : It's the space required to store the environment information needed to resume the suspended function.

Data Structures

Data Structure is a way to organized data in such a way that it can be used efficiently.

Data Type

Data type is way to classify various types of data such as integer, string etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. Data type of two types –

1. **Built-in Data Type**
   Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provides following built-in data types.
   - Integers
   - Boolean (true, false)
   - Floating (Decimal numbers)
   - Character and Strings
2. **Derived Data Type**
   Those data types which are implementation independent as they can be implemented in one or other way are known as derived data types. These data types are normally built by combination of primary or built-in data types and associated operations on them. For example −
   - List
   - Array
   - Stack
   - Queue

Basic Operations in Data Structures

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing- Process of visiting each element in the structure
- Searching- Find a specified element
- Insertion- Insert a new element to the data structure
- Deletion- Deleting an element from the data structure
- Sorting- Arranging the elements in a specified manner
- Merging- Joining two set of elements

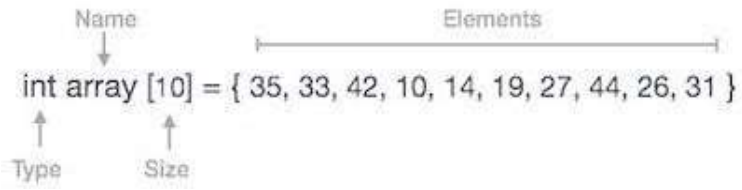
Review of Arrays

Array is a container which can hold fixed number of items and these items should be of same type. Most of the data structure make use of array to implement their algorithms. Following are important terms to understand the concepts of Array.

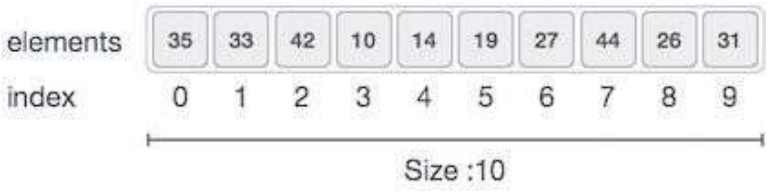
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per above shown illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 8 which means it can store 8 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 9.

Basic Operations in Array

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.
- Insertion − add an element at given index.
- Deletion − delete an element at given index.
- Search − search an element using given index or by value.
- Update − update an element at given index.

MULTIDIMENTIONAL ARRAY REPRESENTATION IN MEMORY

TWO DIMENTIONAL ARRAY (2D Array)

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

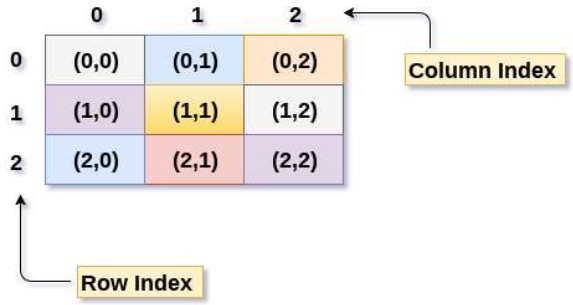
However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two-dimensional array is very much similar to that of a one-dimensional array, given as follows.

```
int arr[max_rows][max_columns];
```

however, it produces the data structure which looks like following.



The syntax to declare and initialize the 2D array is given as follows.

int arr[2][2] = {0,1,2,3};

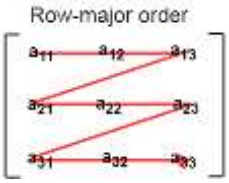
The number of elements that can be present in a 2D array will always be equal to (number of rows * number of columns).

There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

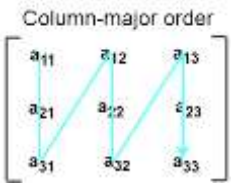
first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.



2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.

first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Size of multidimensional arrays
The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The array **int x[10][20]** can store total (10*20) = 200 elements.
Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

Three dimensional array:

```
int three_d[10][20][30];
```

Traversal operation

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

```
#include <stdio.h>
void main() {
  int Arr[5] = {18, 30, 15, 70, 12};
int i;
  printf("Elements of the array are:\n");
  for(i = 0; i<5; i++) {
    printf("Arr[%d] = %d,  ", i, Arr[i]);
  }
}
```

Output

```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12,
```

Insertion operation

This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array. Now, let's see the implementation of inserting an element into the array.

```
#include <stdio.h>
int main()
{
int arr[20] = { 18, 30, 15, 70, 12 };
int i, x, pos, n = 5;
```

```
printf("Array elements before insertion\n");
for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
printf("\n");

   x = 50; // element to be inserted
   pos = 4;
   n++;

   for (i = n-1; i >= pos; i--)
      arr[i] = arr[i - 1];
   arr[pos - 1] = x;
   printf("Array elements after insertion\n");
   for (i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
   return 0;
}
```

Output

```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion operation

As the name implies, this operation removes an element from the array and then reorganizes all of the array elements.

```
#include <stdio.h>
void main() {
int arr[] = {18, 30, 15, 70, 12};
int k = 30, n = 5;
int i, j;
printf("Given array elements are :\n");
for(i = 0; i<n; i++) {
        printf("arr[%d] = %d,  ", i, arr[i]);
}
j = k;
```

```
    while( j < n) {
      arr[j-1] = arr[j];
      j = j + 1;
    }

    n = n -1;
        printf("\nElements of array after deletion:\n");
        for(i = 0; i<n; i++) {
      printf("arr[%d] = %d,  ", i, arr[i]);
    }
}
```

output

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Elements of array after deletion:
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,
```

Sparse Matrix in Data Structure

- If a matrix has most of its elements equal to zero, then the matrix is known as a sparse matrix.
- In the case of a sparse matrix, we don't store the zeros in the memory to reduce memory usage and make it more efficient.
- We only store the non-zero values of the sparse matrix inside the memory.

For example, in the following 5*4 matrix, most of the numbers are zero. Only a few elements are non-zero which makes it a sparse matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 \\ 9 & 0 & 0 & 6 \\ 7 & 0 & 0 & 0 \end{bmatrix}$$

- A sparse matrix is a matrix in which the number of zeros is more than the number of non-zero elements.
- If we store this sparse matrix as it is, it will consume a lot of space. Therefore, we store only non-zero values in the memory in a more efficient way.

There are mainly two reasons for using sparse matrices. These are:

1. Computation time: If we store the sparse matrix in a memory-efficient manner, we can save a lot of computational time to perform operations on the matrix.
2. Storage: When we store only non-zero elements, we can save a lot of memory/space that we can use for storing other data structures or performing other operations.

Linked List

- A linked-list is a sequence of data structures which are connected together via links.
- Linked List is a sequence of links which contains items.
- Each node of linked list contains two fields.
    1. Data field: Contains actual data
    2. Address filed : Contains the address of next node

Following are important terms to understand the concepts of Linked List.

1. Link − Each Link of a linked list can store a data called an element.
2. Next − Each Link of a linked list contain a link to next link called Next.
3. LinkedList − A LinkedList contains the connection link to the first Link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per above shown illustration, following are the important points to be considered.

- LinkedList contains an link element called first.
- Each Link carries a data field(s) and a Link Field called next.
- Each Link is linked with its next link using its next link.
- Last Link carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various flavours of linked list.

1.Simple Linked List − Item Navigation is forward only.

2.Doubly Linked List − Items can be navigated forward and backward way.

3.Circular Linked List − Last item contains link of the first element as next and and first element has link to last element as prev.

Basic Operations in Linked List

Following are the basic operations supported by a list.

1. Insertion − add an element at the beginning of the list.
2. Deletion − delete an element at the beginning of the list.
3. Display − displaying complete list.
4. Search − search an element using given key.
5. Delete − delete an element using given key.

Implementing Structure Using Pointer /

How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Now, one would ask how are they connected or how they can be traversed? Well, they are connected through pointers. Usually a block in a linked list is represented through a structure like this :

```
Struct  list
{     int val;
struct
list*next;
};
```

So as you can see here, this structure contains a value ‘val’ and a pointer to a structure of same type. The value ‘val’ can be any value (depending upon the data that the linked list is holding) while the pointer ‘next’ contains the address of next block of this linked list. So linked list traversal is made possible through these ‘next’ pointers that contain address of the next node. The ‘next’ pointer of the last node (or for a single node linked list) would contain a NULL.
How a node is created?

A node is created by allocating memory to a structure (as shown in above point) in the following way struct list *ptr = (struct list*)malloc(sizeof(struct list));
How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Now, one would ask how are they connected or how they can be traversed? Well, they are connected through pointers. Usually a block in a linked list is represented through a structure like this :

```
struct list
{
int val;          struct
list*next;
};
```

So as you can see here, this structure contains a value ‘val’ and a pointer to a structure of same type. The value ‘val’ can be any value (depending upon the data that the linked list is holding) while the pointer ‘next’ contains the address of next block of this linked list. So linked list traversal is made possible through these ‘next’ pointers that contain address of the next node. The ‘next’ pointer of the last node (or for a single node linked list) would contain a NULL.
How a node is created?

A node is created by allocating memory to a structure (as shown in above point) in the following way :

struct list*ptr = (struct list*)malloc(sizeof(struct list));

Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

Step 1 - Include all the header files which are used in the program.

Step 2 - Declare all the user defined functions.

Step 3 - Define a Node structure with two members data and next

Step 4 - Define a Node pointer 'head' and set it to NULL.

Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

Inserting At Beginning of the list

Inserting At End of the list

Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode→next = NULL and head = newNode.

Step 4 - If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

Step 1 - Create a newNode with given value and newNode → next as NULL.

Step 2 - Check whether list is Empty (head == NULL).

Step 3 - If it is Empty then, set head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).

Step 6 - Set temp → next = newNode.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a newNode with given value.

Step 2 - Check whether list is Empty (head == NULL)

Step 3 - If it is Empty then, set newNode → next = NULL and head = newNode.

Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.

Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).

Step 6 - Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.

Step 7 - Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node
- Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Check whether list is having only one node (temp → next == NULL)

Step 5 - If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)

Step 6 - If it is FALSE then set head = temp → next, and delete temp.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Check whether list has only one Node (temp1 → next == NULL)

Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)

Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)

Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.

Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.

Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).

Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).

Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.

Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).

Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

Step 1 - Check whether list is Empty (head == NULL)

Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.

Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node

Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

……
……

```
/*Linked
List*/
        #include<stdio.h>
        #include<stdlib.h>
        int count=0;
        struct Node *start=NULL;
        struct Node
        {
                int data;
                struct Node *next;
        };
        void insert_at_begin(int x)
        {
                struct Node *t;
                t=(struct Node*)malloc(sizeof(struct Node));
                count++;
                if(start==NULL)
                {
                        start=t;
                        start->data=x;
                        start->next=NULL;
                        return;
                }
                t->data=x;
                t->next=start;
                start=t;
        }
        void insert_at_end(int x)
        {
                struct Node *t,*temp;
                t=(struct Node*)malloc(sizeof(struct Node));
                count++;
                if(start==NULL)
                {
                        start=t;
                        start->data=x;
                        start->next=NULL;
                        return;
                }
                temp=start;
                while(temp->next!=NULL)
                {
                        temp=temp->next;
```

```
        }
        temp->next=t;
        t->data=x;
        t->next=NULL;
}
void delete_from_begin()
{
        struct Node *t;
        int n;
        if(start==NULL)
        {
                printf("Linked List is empty!!!\n\n");
                return;
        }
        n=start->data;
        t=start->next;
        free(start);
        start=t;
        count--;
        printf("Deleted element is %d\n\n",n);
        return;
}
void delete_from_end()
{
        struct Node *t,*u;
        int n;
        if(start==NULL)
        {
                printf("Linked List is empty!!!\n\n");
                return;
        }
        count--;
        if(start->next==NULL)
        {
                n=start->data;
                free(start);
                start=NULL;
                printf("Deleted element is %d\n\n",n);
                return;
        }
        t=start;
        while(t->next!=NULL)
        {
                u=t;
                t=t->next;
        }
        n=t->data;
        u->next=NULL;
```

```
        free(t);
        printf("Deleted element is %d\n\n",n);
        return;
}
void display()
{
        struct Node *t;
        if(start==NULL)
        {
                printf("Linked List is empty!!!\n\n");
                return;
        }
        printf("No of elements: %d\n",count);
        printf("Elements are: ");
        t=start;
        while(t->next!=NULL)
        {
                printf("%d ",t->data);
                t=t->next;
        }
        printf("%d ",t->data);
        printf("\n\n");
}
int main()
{
        int ch,data;
        while(1)
        {
                printf("---LINKED LIST PROGRAMS---\n");
                printf("1. INSERT AT BEGINING\n");
                printf("2. INSERT AT END\n");
                printf("3. DELETE FROM BEGINING\n");
                printf("4. DELETE FROM END\n");
                printf("5. DISPLAY LIST\n");
                printf("6. EXIT\n\n");
                printf("Enter your choice: ");
                scanf("%d",&ch);
                if(ch==1)
                {
                        printf("Enter the insert value: ");
                        scanf("%d",&data);
                        printf("\n");
                        insert_at_begin(data);
                }
                else if(ch==2)
                {
                        printf("Enter the insert value: ");
                        scanf("%d",&data);
```

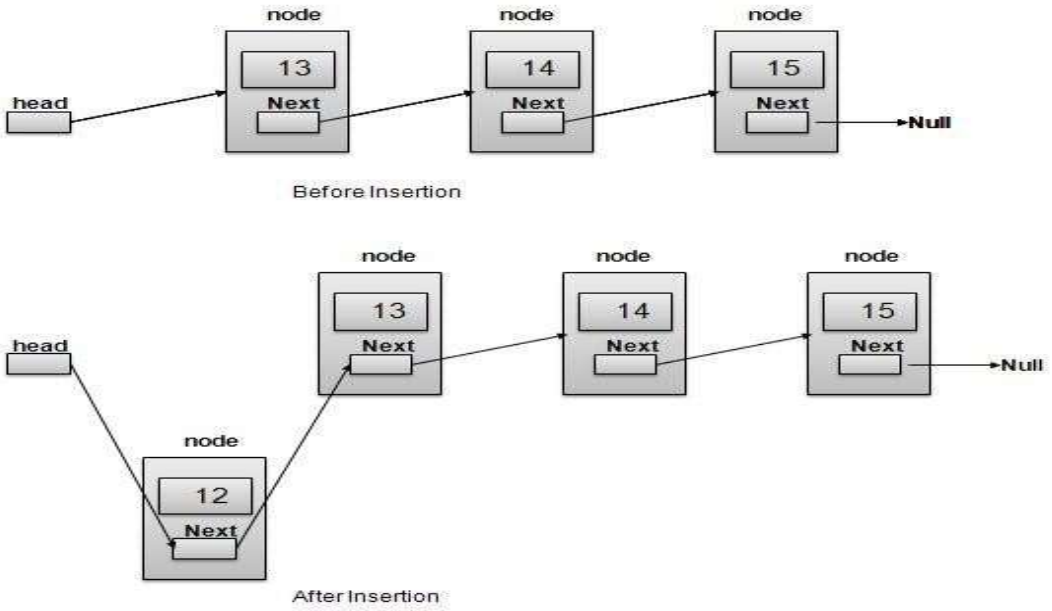
```
                printf("\n");
                insert_at_end(data);
            }
            else if(ch==3)
            {
                delete_from_begin();
            }
            else if(ch==4)
            {
                delete_from_end();
            }
            else if(ch==5)
            {
                display();
            }
            else if(ch==6)
            {
                break;
            }
            else
            {
                printf("Wrong choice!!!\n");
            }
        }
    }
```

Basic Operations on Linked List

1. Insertion Operation

Insertion is a three step process −

1. Create a new Link with provided data.
2. Point New Link to old First Link.
3. Point First Link to this New Link.

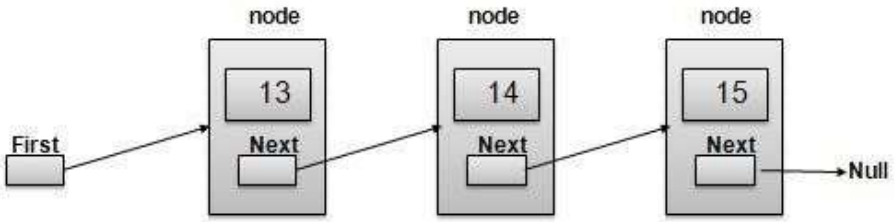



```
//insert link at the first location void
insertFirst(int key, int data)
{
  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));    link->key
= key;    link->data = data;

  //point it to old first node    link->next = head;

  //point first to new first node    head = link;
}
```

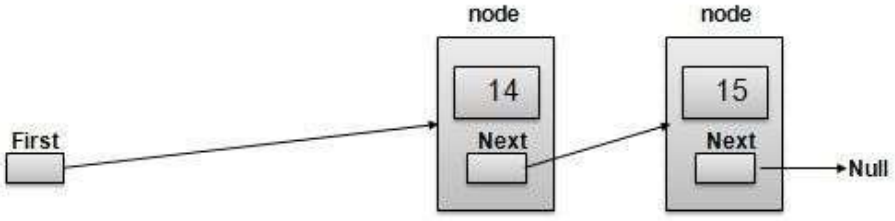
2. <u>Deletion Operation</u>

Deletion is a two step process −

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.



Before Deletion



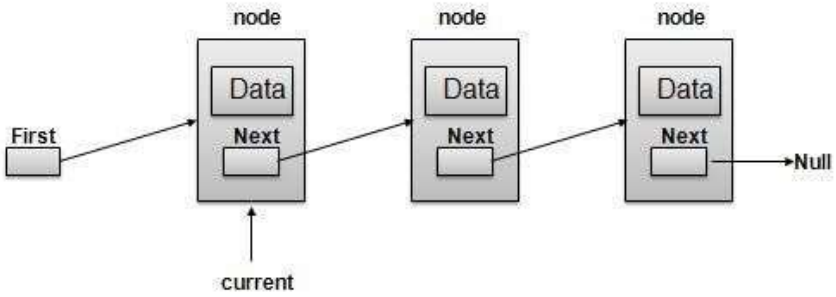
After Deletion

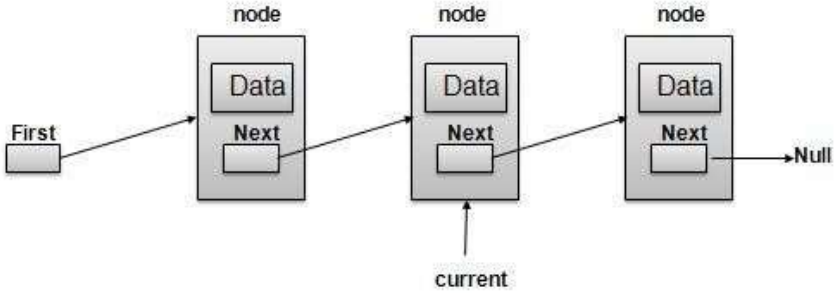
```
//delete first item struct
node* deleteFirst()
{
  //save reference to first link    struct
node *tempLink = head;

  //mark next to first link as first     head =
head->next;

  //return the deleted link    return
tempLink;
}
```

3. Traversing Operation

Navigation is a recursive step process and is basis of many operations like search, delete etc. −

- Get the Link pointed by First Link as Current Link.
- Check if Current Link is not null and display it.
- Point Current Link to Next Link of Current Link and move to above step.






```
//display the list void printList()
{
   struct node *ptr = head;    printf("\n[ ");

   //start from the beginning    while(ptr != NULL){
      printf("(%d,%d) ",ptr->key,ptr->data);       ptr = ptr-
>next;
   }

   printf(" ]");
……
……
……
}
```

Circular Linked List

A circular linked list is a type of linked list which the first and the last nodes are also connected to each other to form a circle.

Here, the address of the last node consists of the address of the first node.

Representation of Circular Linked List

Let's see how we can represent a circular linked list on an algorithm/code. Suppose we have a linked list:

Initial circular linked list

Here, the single node is represented as

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

Each struct node has a data item and a pointer to the next struct node.

Insertion on a Circular Linked List

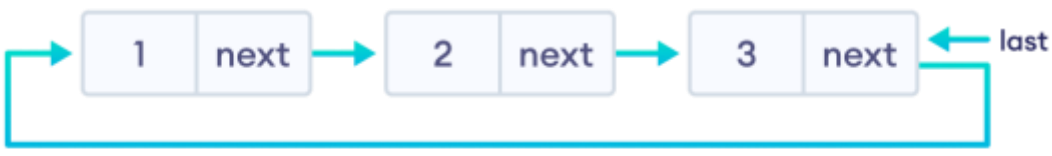
We can insert elements at 3 different positions of a circular linked list:

1. Insertion at the Beginning

2. Insertion in between two nodes

3. Insertion at the end

Suppose we have a circular linked list with elements 1, 2, and 3.



Initial circular linked list

Let's add a node with value 6 at different positions of the circular linked list we made above.
The first step is to create a new node.

- allocate memory for newNode
- assign the data to newNode

New Node

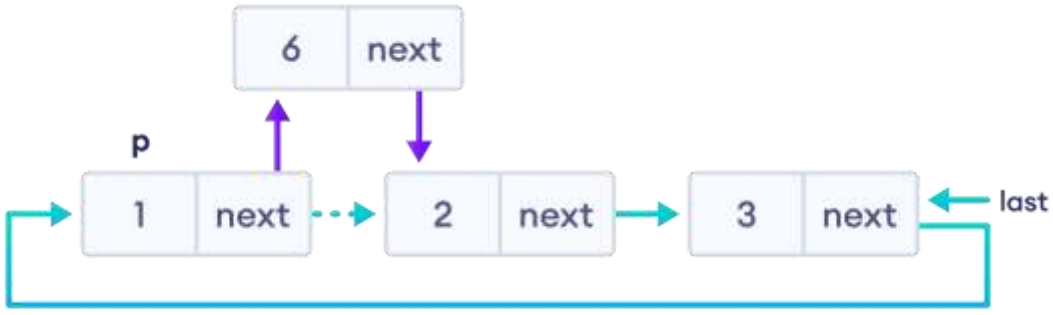
1. Insertion at the Beginning

- store the address of the current first node in the newNode (i.e. pointing the newNode to the current first node)
- point the last node to newNode (i.e making newNode as head)

2. Insertion in between two nodes

Let's insert newNode after the first node.

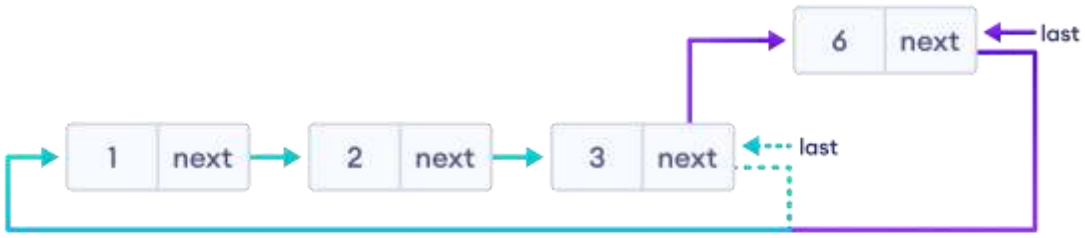
- travel to the node given (let this node be p)
- point the next newNode to the node next to p
- store the address newNode at next of p



3. Insertion at the end

- store the address of the head node to next of newNode (making newNode the last node)
- point the current last node to newNode

- make newNode as the last node



Deletion on a Circular Linked List

Suppose we have a double-linked list with elements 1, 2, and 3.

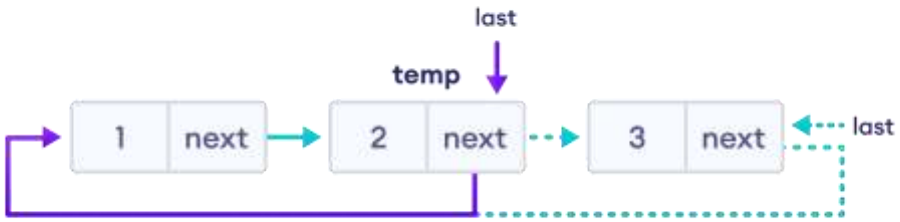
Initial circular linked list

1. If the node to be deleted is the only node

- free the memory occupied by the node
- store NULL in last

2. If last node is to be deleted

- find the node before the last node (let it be temp)
- store the address of the node next to the last node in temp
- free the memory of last
- make temp as the last node



Delete the last node

3. If any other nodes are to be deleted

- travel to the node to be deleted (here we are deleting node 2)
- let the node before node 2 be temp
- store the address of the node next to 2 in temp

- free the memory of 2

Delete a specific node

SOURCE CODE : :

```
/* C Program of implement circular linked list*/

#include<stdio.h>

#include<stdlib.h>

struct node

{

    int info;

    struct node *link;

};

struct node *create_list(struct node *last);

void display(struct node *last);

struct node *addtoempty(struct node *last,int data);

struct node *addatbeg(struct node *last,int data);

struct node *addatend(struct node *last,int data);

struct node *addafter(struct node *last,int data,int item);

struct node *del(struct node *last,int data);

int main( )

{

    int choice,data,item;
```

```
struct node     *last=NULL;
while(1)
{
        printf("\n\n1.Create List\n");
        printf("2.Display\n");
        printf("3.Add to empty list\n");
        printf("4.Add at beginning\n");
        printf("5.Add at end\n");
        printf("6.Add after \n");
        printf("7.Delete\n");
        printf("8.Quit\n");

        printf("\nEnter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
         case 1:
                last=create_list(last);
                break;
         case 2:
                display(last);
                break;
         case 3:
                printf("\nEnter the element to be inserted : ");
                scanf("%d",&data);
                last=addtoempty(last,data);
```

```
        break;
    case 4:
        printf("\nEnter the element to be inserted : ");
        scanf("%d",&data);
        last=addatbeg(last,data);
        break;
    case 5:
        printf("\nEnter the element to be inserted : ");
        scanf("%d",&data);
        last=addatend(last,data);
        break;
    case 6:
        printf("\nEnter the element to be inserted : ");
        scanf("%d",&data);
        printf("\nEnter the element after which to insert : ");
        scanf("%d",&item);
        last=addafter(last,data,item);
        break;
    case 7:
        printf("\nEnter the element to be deleted : ");
        scanf("%d",&data);
        last=del(last,data);
        break;
    case 8:
        exit(1);
    default:
```

```
                printf("\nWrong choice\n");
            }/*End of switch*/
        }/*End of while*/
        return 0;
}/*End of main( )*/
struct node *create_list(struct node *last)
{
        int i,n,data;
        printf("\nEnter the number of nodes : ");
        scanf("%d",&n);
        last=NULL;
        if(n==0)
            return last;
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        last=addtoempty(last,data);
        for(i=2;i<=n;i++)
        {
            printf("Enter the element to be inserted : ");
            scanf("%d",&data);
            last=addatend(last,data);
        }
        return last;
}/*End of create_list()*/
struct node *addtoempty(struct node *last,int data)
{
```

```
        struct node *tmp;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        last=tmp;
        last->link=last;
        return last;
}/*End of addtoempty( )*/
struct node *addatbeg(struct node *last,int data)
{
        struct node *tmp;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=last->link;
        last->link=tmp;
        return last;
}/*End of addatbeg( )*/
struct node *addatend(struct node *last,int data)
{
        struct node *tmp;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=last->link;
        last->link=tmp;
        last=tmp;
        return last;
}/*End of addatend( )*/
```

```
struct node *addafter(struct node *last,int data,int item)
{
        struct node *tmp,*p;
        p=last->link;
        do
        {
                if(p->info==item)
                {
                        tmp=(struct node *)malloc(sizeof(struct node));
                        tmp->info=data;
                        tmp->link=p->link;
                        p->link=tmp;
                        if(p==last)
                                last=tmp;
                        return last;
                }
                p=p->link;
        }while(p!=last->link);
        printf("%d not present in the list\n",item);
        return last;
}/*End of addafter()*/
struct node *del(struct node *last,int data)
{
        struct node *tmp,*p;
        if(last==NULL)
```

```
{
        printf("List is empty\n");
        return last;
}
/*Deletion of only node*/
if(last->link==last && last->info==data)
{
        tmp=last;
        last=NULL;
        free(tmp);
        return last;
}
/*Deletion of first node*/
if(last->link->info==data)
{
        tmp=last->link;
        last->link=tmp->link;
        free(tmp);
        return last;
}
/*Deletion in between*/
p=last->link;
while(p->link!=last)
{
        if(p->link->info==data)
        {
```

```
                tmp=p->link;
                p->link=tmp->link;
                free(tmp);
                return last;
            }
            p=p->link;
        }
        /*Deletion of last node*/
        if(last->info==data)
        {
            tmp=last;
            p->link=last->link;
            last=p;
            free(tmp);
            return last;
        }
        printf("\nElement %d not found\n",data);
        return last;
}/*End of del( )*/
void display(struct node *last)
{
        struct node *p;
        if(last==NULL)
        {
            printf("\nList is empty\n");
            return;
```

```
        }
        p=last->link;
        do
        {
                printf("%d ",p->info);
                p=p->link;
        }while(p!=last->link);
        printf("\n");
}/*End of display( )*/
```

OUTPUT : :

```
********** OUTPUT **********
1.Create List
2.Display
3.Add to empty list
4.Add at beginning
5.Add at end
6.Add after
7.Delete
8.Quit
Enter your choice : 1
Enter the number of nodes : 3
Enter the element to be inserted : 1
```

```
Enter the element to be inserted : 2

Enter the element to be inserted : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 2

1 2 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 3

Enter the element to be inserted : 2

1.Create List

2.Display

3.Add to empty list

4.Add at beginning
```

```
5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 2

2

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit



Enter your choice : 4

Enter the element to be inserted : 1

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit
```

```
Enter your choice : 2

1 2

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 5

Enter the element to be inserted : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 2

1 2 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning
```

```
5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 6

Enter the element to be inserted : 4

Enter the element after which to insert : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 2

1 2 3 4

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit
```

```
Enter your choice : 7

Enter the element to be deleted : 1

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 2

2 3 4

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Delete

8.Quit

Enter your choice : 8
```

Doubly Linked List

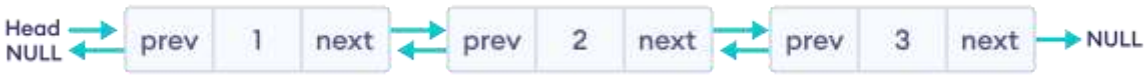
A doubly linked list is a type of linked list in which each node consists of 3 components:

- *prev - address of the previous node
- data - data item
- *next - address of next node

A doubly linked list node

Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

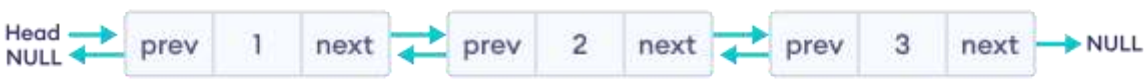
Here, the single node is represented as

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

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

Insertion on a Doubly Linked List

Suppose we have a double-linked list with elements **1**, **2**, and **3**.



Original doubly linked list

1. Insertion at the Beginning

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

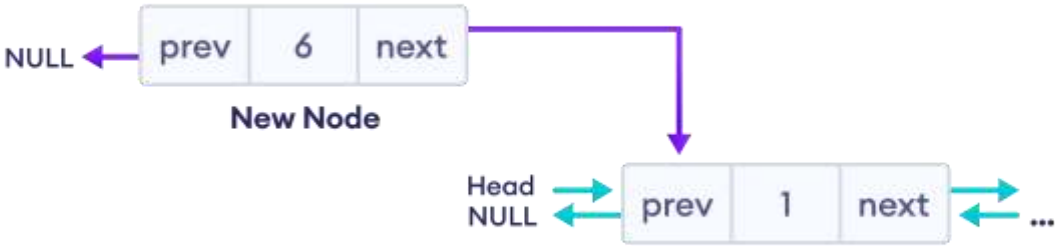
1. Create a new node

- allocate memory for newNode
- assign the data to newNode.

New node

2. Set prev and next pointers of new node

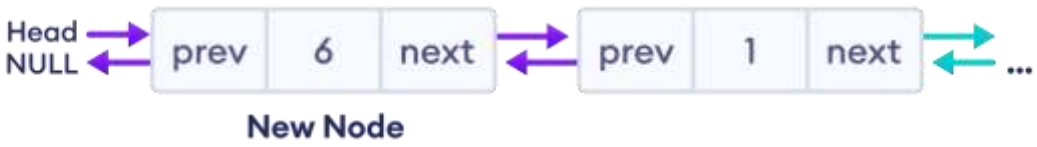
- point next of newNode to the first node of the doubly linked list
- point prev to null



Reorganize the pointers (changes are denoted by purple arrows)

3. Make new node as head node

- Point prev of the first node to newNode (now the previous head is the second node) d
- Point hea) newNode



Reorganize the pointers

2. Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

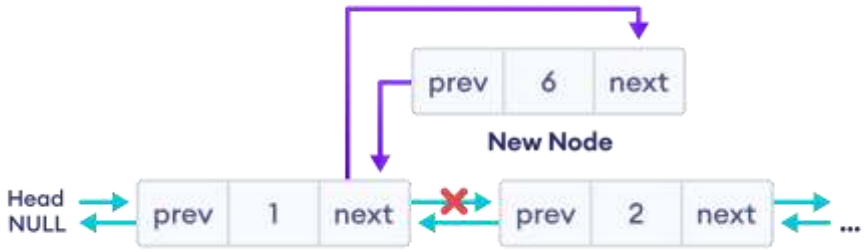
1. Create a new node

- allocate memory for newNode
- assign the data to newNode.

New node

2. Set the next pointer of new node and previous node

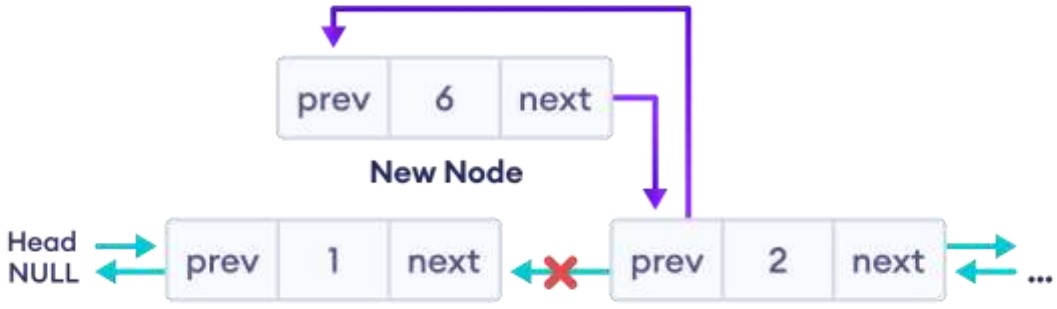
- assign the value of next from previous node to the next of newNode
- assign the address of newNode to the next of previous node



Reorganize the pointers

3. Set the prev pointer of new node and the next node

- assign the value of prev of next node to the prev of newNode
- assign the address of newNode to the prev of next node



Reorganize the pointers

The final doubly linked list is after this insertion is:

Final list

3. Insertion at the End

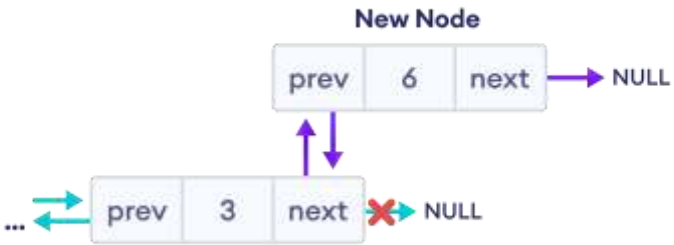
Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

New node

2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the newNode as the head node. Otherwise, traverse to the end of the doubly linked list and



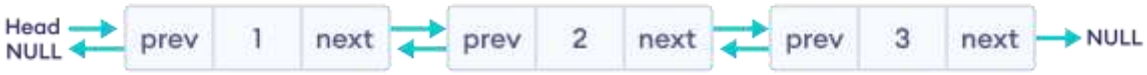
Reorganize the pointers

The final doubly linked list looks like this.

The final list

Deletion from a Doubly Linked List

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.
Suppose we have a double-linked list with elements **1**, **2**, and **3**.

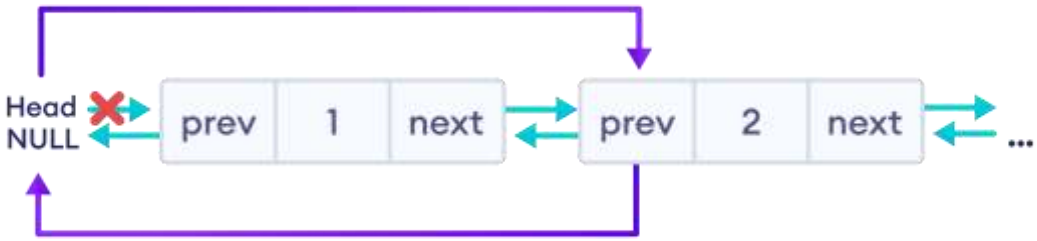


Original doubly linked list

1. Delete the First Node of Doubly Linked List

If the node to be deleted (i.e. `del_node`) is at the beginning **Reset value node**

after the del_node (i.e. node two)



Reorganize the pointers

Finally, free the memory of `del_node`. And, the linked will look like this

Final list

2. Deletion of the Inner Node

If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the del_node (i.e. first node)
Assign the value of `next` of `del_node` to the `next` of the `first` node.
For the node after the del_node (i.e. third node)
Assign the value of `prev` of `del_node` to the `prev` of the `third` node.

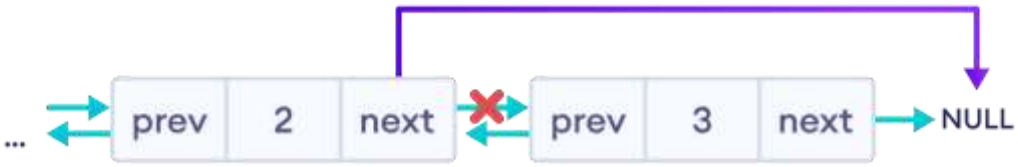
Reorganize the pointers

Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.

Final list

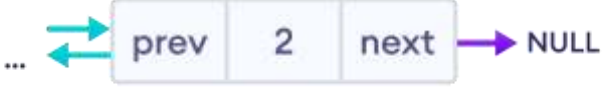
3. Delete the Last Node of Doubly Linked List

In this case, we are deleting the last node with value **3** of the doubly linked list.
Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.



Final list

```
/*C Program to implement double linked list operations */
#include<stdio.h>
#include<stdlib.h>
struct node
{
        struct node *prev;
        int info;
        struct node *next;
};
struct node *create_list(struct node *start);
void display(struct node *start);
struct node *addtoempty(struct node *start,int data);
struct node *addatbeg(struct node *start,int data);
struct node *addatend(struct node *start,int data);
struct node *addafter(struct node *start,int data,int item);
struct node *addbefore(struct node *start,int data,int item);
struct node *del(struct node *start,int data);
struct node *reverse(struct node *start);
int main()
{
        int choice,data,item;
        struct node *start=NULL;
        while(1)
        {
                printf("\n\n1.Create List\n");
```

```
printf("2.Display\n");
printf("3.Add to empty list\n");
printf("4.Add at beginning\n");
printf("5.Add at end\n");
printf("6.Add after\n");
printf("7.Add before\n");
printf("8.Delete\n");
printf("9.Reverse\n");
printf("10.Quit\n");
printf("\nEnter your choice : ");
scanf("%d",&choice);
switch(choice)
{
 case 1:
        start=create_list(start);
        break;
 case 2:
        display(start);
        break;
 case 3:
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addtoempty(start,data);
        break;
 case 4:
printf("Enter the element to be inserted : ");
```

```
        scanf("%d",&data);
        start=addatbeg(start,data);
        break;
    case 5:
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        start=addatend(start,data);
        break;
    case 6:
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        printf("Enter the element after which to insert : ");
        scanf("%d",&item);
        start=addafter(start,data,item);
        break;
    case 7:
        printf("Enter the element to be inserted : ");
        scanf("%d",&data);
        printf("Enter the element before which to insert : ");
        scanf("%d",&item);
        start=addbefore(start,data,item);
        break;
    case 8:
        printf("Enter the element to be deleted : ");
        scanf("%d",&data);
        start=del(start,data);
```

```
                break;
            case 9:
                start=reverse(start);
                break;
            case 10:
                exit(1);
            default:
                printf("Wrong choice\n");
        }/*End of switch*/
    }/*End of while*/
    return 0;
}/*End of main()*/
struct node *create_list(struct node *start)
{
    int i,n,data;
    printf("\nEnter the number of nodes : ");
    scanf("%d",&n);
    start=NULL;
    if(n==0)
        return start;
    printf("Enter the element to be inserted : ");
    scanf("%d",&data);
    start=addtoempty(start,data);
    for(i=2;i<=n;i++)
    {
        printf("Enter the element to be inserted : ");
```

```
            scanf("%d",&data);
            start=addatend(start,data);
        }
        return start;
}/*End of create_list()*/
void display(struct node *start)
{
        struct node *p;
        if(start==NULL)
        {
            printf("\nList is empty\n");
            return;
        }
        p=start;
        printf("\nList is :\n");
        while(p!=NULL)
        {
            printf("%d ",p->info);
            p=p->next;
        }
        printf("\n");
}/*End of display() */
struct node *addtoempty(struct node *start,int data)
{
        struct node *tmp;
        tmp=(struct node *)malloc(sizeof(struct node));
```

```
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=NULL;
        start=tmp;
        return start;
}/*End of addtoempty( )*/
struct node *addatbeg(struct node *start,int data)
{
        struct node *tmp;
        tmp = (struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=start;
        start->prev=tmp;
        start=tmp;
        return start;
}/*End of addatbeg( )*/
struct node *addatend(struct node *start,int data)
{
        struct node *tmp,*p;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        p=start;
        while(p->next!=NULL)
                p=p->next;
        p->next=tmp;
```

```
        tmp->next=NULL;
        tmp->prev=p;
        return start;
}/*End of addatend( )*/
struct node *addafter(struct node *start,int data,int item)
{
        struct node *tmp,*p;
        tmp=(struct node *)malloc(sizeof(struct node));
        tmp->info=data;
        p=start;
        while(p!=NULL)
        {
                if(p->info==item)
                {
                        tmp->prev=p;
                        tmp->next=p->next;
                        if(p->next!=NULL)
                                p->next->prev=tmp;
                        p->next=tmp;
                        return start;
                }
                p=p->next;
        }
        printf("\n%d not present in the list\n\n",item);
        return start;
}/*End of addafter()*/
```

```
struct node *addbefore(struct node *start,int data,int item)
{
        struct node *tmp,*q;
        if(start==NULL )
        {
                printf("\nList is empty\n");
                return start;
        }
        if(start->info==item)
        {
                tmp = (struct node *)malloc(sizeof(struct node));
                tmp->info=data;
                tmp->prev=NULL;
                tmp->next=start;
                start->prev=tmp;
                start=tmp;
                return start;
        }
        q=start;
        while(q!=NULL)
        {
                if(q->info==item)
                {
                        tmp=(struct node *)malloc(sizeof(struct node));
                        tmp->info=data;
```

```
                tmp->prev=q->prev;
                tmp->next = q;
                q->prev->next=tmp;
                q->prev=tmp;
                return start;
            }
            q=q->next;
        }
        printf("\n%d not present in the list\n",item);
        return start;
}/*End of addbefore()*/
struct node *del(struct node *start,int data)
{
        struct node *tmp;
        if(start==NULL)
        {
            printf("\nList is empty\n");
            return start;
        }
        if(start->next==NULL)       /*only one node in the list*/
            if(start->info==data)
            {
                tmp=start;
                start=NULL;
                free(tmp);
                return start;
```

```
        }
        else
        {
                printf("\nElement %d not found\n",data);
                return start;
        }
/*Deletion of first node*/
if(start->info==data)
{
        tmp=start;
        start=start->next;
        start->prev=NULL;
        free(tmp);
        return start;
}
/*Deletion in between*/
tmp=start->next;
while(tmp->next!=NULL )
{
        if(tmp->info==data)
        {
                tmp->prev->next=tmp->next;
                tmp->next->prev=tmp->prev;
                free(tmp);
                return start;
        }
```

```
            tmp=tmp->next;
        }
        /*Deletion of last node*/
        if(tmp->info==data)
        {
            tmp->prev->next=NULL;
            free(tmp);
            return start;
        }
        printf("\nElement %d not found\n",data);
        return start;
}/*End of del()*/

struct node *reverse(struct node *start)
{
        struct node *p1,*p2;
        p1=start;
        p2=p1->next;
        p1->next=NULL;
        p1->prev=p2;
        while(p2!=NULL)
        {
            p2->prev=p2->next;
            p2->next=p1;
            p1=p2;
            p2=p2->prev;
```

```
        }
        start=p1;
        printf("\nList reversed\n");
        return start;
}/*End of reverse()*/
```

OUTPUT : :

```
*************** OUTPUT ***************
1.Create List
2.Display
3.Add to empty list
4.Add at beginning
5.Add at end
6.Add after
7.Add before
8.Delete
9.Reverse
10.Quit
Enter your choice : 1
Enter the number of nodes : 4
Enter the element to be inserted : 1
Enter the element to be inserted : 2
Enter the element to be inserted : 3
```

```
Enter the element to be inserted : 4

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit



Enter your choice : 2

List is :

1 2 3 4

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 3

Enter the element to be inserted : 5

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit

Enter your choice : 2

List is :

5

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 4

Enter the element to be inserted : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit

Enter your choice : 2

List is :

3 5

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 5

Enter the element to be inserted : 6

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit

Enter your choice : 6

Enter the element to be inserted : 4

Enter the element after which to insert : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 7

Enter the element to be inserted : 2

Enter the element before which to insert : 3

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit

Enter your choice : 2

List is :

2 3 4 5 6

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 8

Enter the element to be deleted : 6

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit


Enter your choice : 9

List reversed

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit
```

```
Enter your choice : 2

List is :

5 4 3 2

1.Create List

2.Display

3.Add to empty list

4.Add at beginning

5.Add at end

6.Add after

7.Add before

8.Delete

9.Reverse

10.Quit

Enter your choice : 10
```

Singly Linked List Vs Doubly Linked List

Singly Linked List	Doubly Linked List
Each node consists of a data value and a pointer to the next node.	Each node consists of a data value, a pointer to the next node, and a pointer to the previous node.
Traversal can occur in one way only (forward direction).	Traversal can occur in both ways.
It requires less space.	It requires more space because of an extra pointer.
It can be implemented on the stack.	It has multiple usages. It can be implemented on the stack, heap, and binary tree.

Arrays and Linked List

Arrays	Linked Lists
An array is a collection of elements of a similar data type.	Linked List is an ordered collection of elements of the same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory.	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed.	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation).	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

Applications of Linked Lists

The most important drawback of array is that the array is a static structure; the allocation occurs during the compile time and hence cannot be extended or reduced. Linked list on the other hand have many advantages:

- **Linked list are dynamic data structure:** Linked list can grow or shrink during the execution of the program. Linked lists allocate memory for each element separately and only when necessary.
- **Efficient memory utilization:** Unlike array, where pre-allocation waste memory, linked list on the other hand allocates memory whenever it is required as it is a dynamic data structure.
- **Insertion and Deletion are easier and efficient:** Linked List provides flexibility of inserting and deleting a node from the list.
- Many complex applications can be easily carried out
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance

BASIS FOR COMPARISON	ARRAY	LINKED LIST
Basic	It is a consistent set of a fixed number of data items.	It is an ordered set comprising a variable number of data items.
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient