Types of Array operations:

- Traversal: Traverse through the elements of an array.
- Insertion: Inserting a new element in an array.
- Deletion: Deleting element from the array.
- Searching: Search for an element in the array.
- Sorting: Maintaining the order of elements in the array.

What is traversal operation in array?

- Traversing an array means accessing each and every element of the array for a specific purpose.
- Traversing the data elements of an array can include printing every element, counting the total number of elements, or performing any process on these elements.
- simply traversing an array means, Accessing or printing each element of an array exactly once .so that the *data items (values)* of the array can be checked or used as part of some other operation or process (This accessing and processing is sometimes called “**visiting**” the array).

Note: Accessing or printing of elements always takes place one by one.

Algorithm for Traversing an Array:

```
Step 01: Start
Step 02: [Initialize counter variable. ] Set i = LB.
Step 03: Repeat for i = LB to UB.
Step 04: Apply process to arr[i].
Step 05: [End of loop. ]
Step 06: Stop
```

Variables used:

1. **i :** Loop counter or counter variable for the `for` loop.
2. **arr :** Array name.
3. **LB :** Lower bound. [The *index value* or simply *index* of the first element of an array is called its lower bound]
4. **UB :** Upper bound. [The *index* of the last element is called its **upper bound**]

In step 2 Set i = LB means for(i=0;i<size;i++)

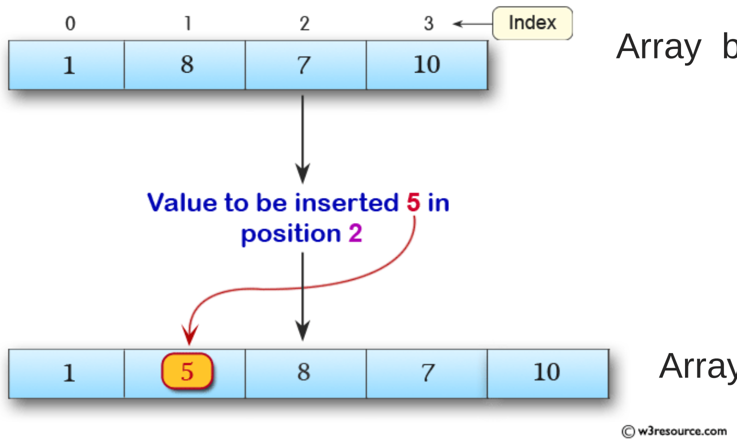
```
// writing a program in C to perform traverse operation in array

#include <stdio.h>
int  main()
{
   int i, size;
   int arr[] = {1, -9, 17, 4, -3};
   size = sizeof(arr)/sizeof(arr[0]);
   //sizeof(arr) will give 20 and sizeof(arr[0]) will give 4
   printf("The array elements are: ");
   for(i=0;i<size;i++)
      {
                        printf("\narr[%d]= %d", i, arr[i]);
      }
      return 0;
}
```

- write a program to read and print elements of array?

```
#include <stdio.h>
int  main()
{
    int arr[10];
    int i,n;
    printf("\n\nRead and Print elements of an array:\n");
    printf("-----------------------------------------\n");
    printf("Input size of array :  ");
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("\nElements in array are: ");
    for(i=0; i<n; i++)
    {
        printf("%d  ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

What is the insertion operation of array in data structure?

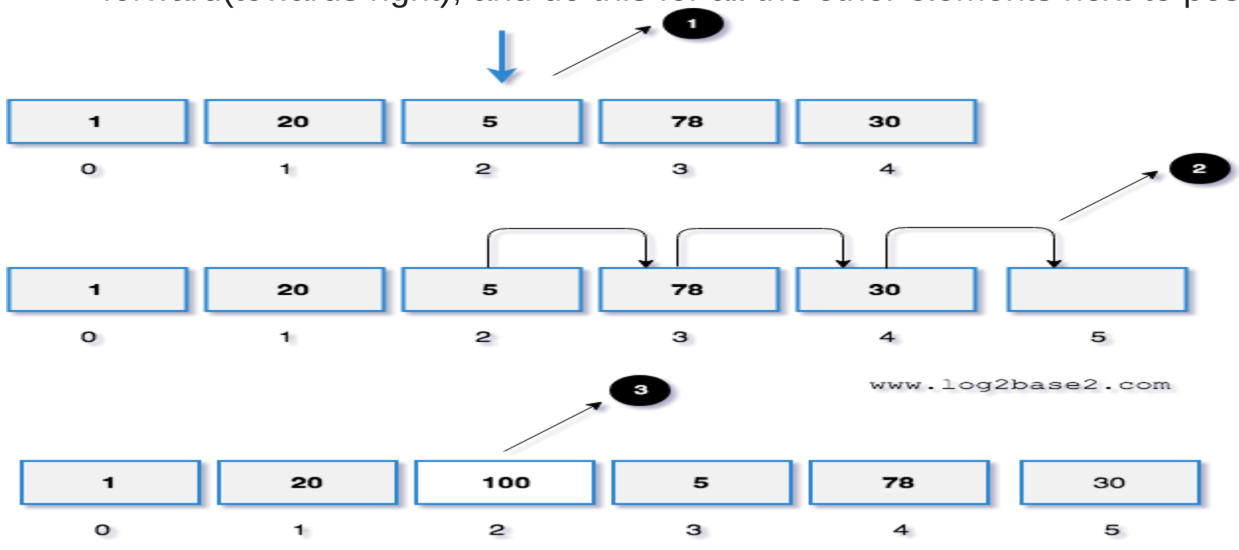
In the insertion operation, we are adding one or more elements to the array. Based on the requirement, a new element can be added at the beginning, end, or any given position of array.



Array before insertion

Array after insertion

1. First get the position at which this element is to be inserted, say pos
2. If position is valid get the element to be inserted, say x.
3. Then shift the array elements from this position to one position forward(towards right), and do this for all the other elements next to pos.



(Steps)

Problem Solution

1.Declare a one-dimensional array of some fixed capacity.
2.Take size of the array as input from users
3.Define array elements, taking each element as input from users.

4.Get the position at which this element is to be inserted, say pos

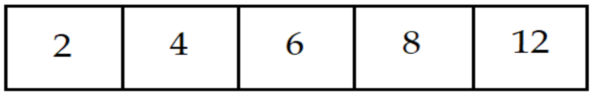
5.If position is valid get the element to be inserted, say x.
6.Then shift the array elements from this position to one position forward(towards right), and do this for all the other elements next to pos.

7.Insert the element in that position.
8.Exit.

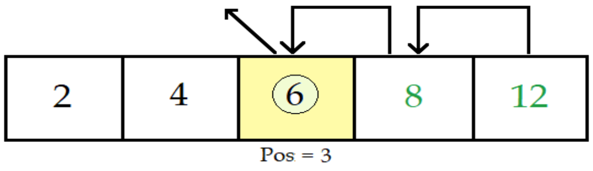
```
#include <stdio.h>
int main()
{
        int a[50],i,size,pos,num;
        printf("Enter array limit  :");
        scanf("%d",&size);

        printf("\nEnter array elemnts\n");
        for(i=0;i<size;i++)
        scanf("%d",&a[i]);

        printf("\nEnter position at which element has to be inserted :");
        scanf("%d",&pos);

        if(pos<=0 || pos>size+1)
                printf("\nInvalid Position\n");
        else
                {
                        printf("\nEnter element is to be inserted :");
                        scanf("%d",&num);
                       size++;
                        for(i=size;i>=pos;i--)
                                {
                                        a[i]=a[i-1];
                                }
                                a[i]=num;

                           printf("\nArray after insertion :\t");
                                for(i=0;i<size;i++)
                                        {
                                                printf("%d\t",a[i]);
                                        }
                }
                return 0;
}
```

What is deletion operation in array?

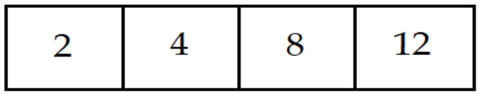
Deletion operation in Array or delete an element from an Array simply means, Removing an existing element from a specific position of an array.



Initial Array



Move each element backward by one place whose position is greater than the element you wish to delete.



Array with 6 deleted from 3rd position

Algorithm to Delete an element from an Array:

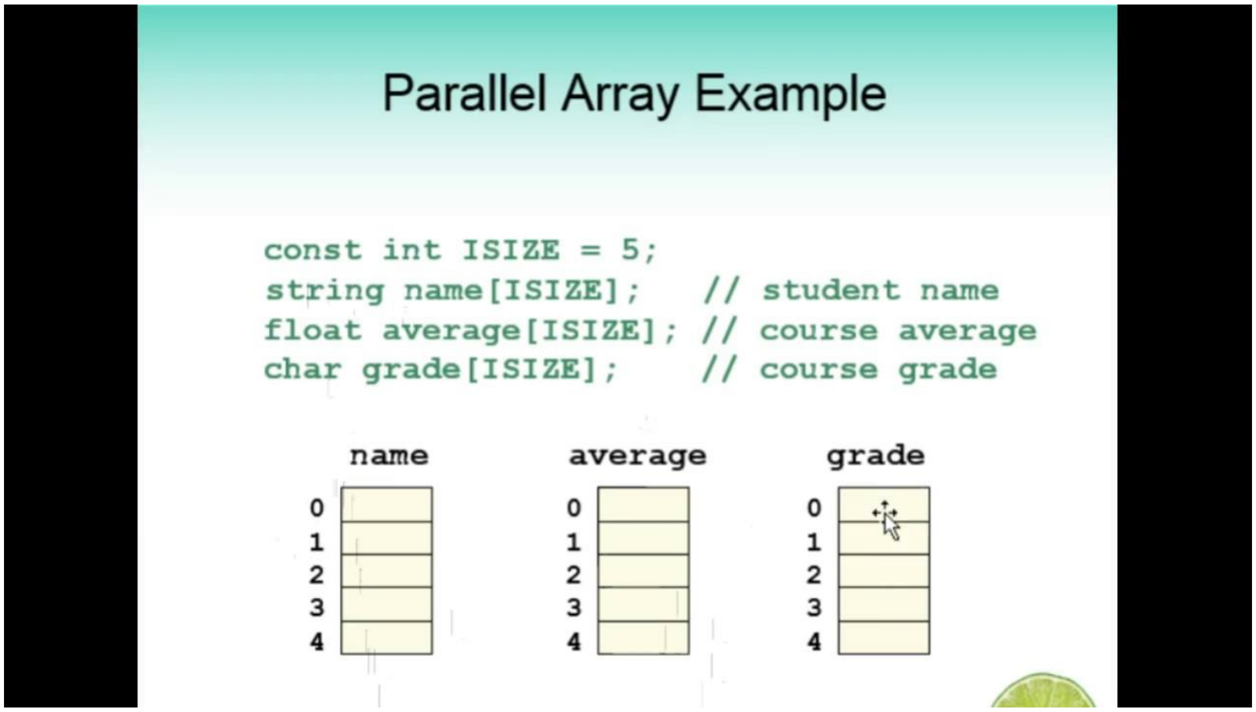
```
Step 01: Start
Step 02: [Initialize counter variable. ] Set i = pos - 1
Step 03: Repeat Step 04 and 05 for i = pos - 1 to i < size
Step 04: [Move i-th element backward (left). ] set a[i] = a[i+1]
Step 05: [Increase counter. ] Set i = i + 1
Step 06: [End of step 03 loop. ]
Step 07: [Reset size of the array. ] set size = size - 1
Step 08: Stop
```

Program:

```
#include <stdio.h>
int main()
{
        int a[50], i, size, pos;
        printf("Enter array limit  :");
        scanf("%d",&size);

        printf("\nEnter array elemnts\n");
        for(i=0;i<size;i++)
        scanf("%d",&a[i]);

        printf("\nEnter position at which element is to be deleted :");
        scanf("%d",&pos);

        if(pos<=0 || pos>size)
                printf("\nInvalid Position\n");
        else
        {
                size--;
                for(i=pos-1;i<size;i++)
                        {
                                a[i]=a[i+1];
                        }
                        printf("\nArray after deletion :\t");
                        for(i=0;i<size;i++)
                        printf("%d\t",a[i]);
        }
        return 0;
}
```

parallel arrays

- A parallel array is a structure of an array. It contains multiple arrays of the same size, in which the i-th element of each array is related to each other.
- **parallel array** is a data structure for representing **arrays of records**.
- **Values** located at the **same index** in **each array** are **implicitly** the **fields of the *same* record**.



Parallel Array Example

```
const int ISIZE = 5;
string name[ISIZE];    // student name
float average[ISIZE]; // course average
char grade[ISIZE];     // course grade
```

Program

```
#include<stdio.h>
#include<conio.h>
#include <stdbool.h>


int main()
{
int rollno[10];
char name[2][10];
int Phy[10];
int Mal[10];
int Eng[10];
float ave[10];
char grade[10];
int i,record_no,std_total;
bool found = false;
printf("\nEnter no of students : " );
scanf("%d",&std_total);
printf("\nEnter student details\n " );
for(i=0;i<std_total;i++)
{
printf("\nEnter roll no            :");
scanf("%d",&rollno[i]);
printf("\nEnter name               :");
scanf("%s",name[i]);
printf("\nEnter marks in Physics   :");
scanf("%d",&Phy[i]);
printf("\nEnter marks in Malayalam :");
scanf("%d",&Mal[i]);
printf("\nEnter marks in English   :");
scanf("%d",&Eng[i]);
}
for(i=0;i<2;i++)
{
   ave[i]=(float)(Phy[i]+Mal[i]+Eng[i])/3;

   if(ave[i]<=50.00&&ave[i]>=40.00)
   grade[i]='A';
   else if(ave[i]<=39.00&&ave[i]>=30.00)
   grade[i]='B';
   else if(ave[i]<=29.00&&ave[i]>=20.00)
   grade[i]='C';
   else
   grade[i]='F';
}


printf("\nRoll no\tName           Average  Grade  \n");
printf("--------------------------------------------");
for(i=0;i<2;i++)
{
printf("\n%d\t%s             %.2f\t%c",rollno[i],name[i],ave[i],grade[i]);
}

printf("\nEnter roll no of student to serach  : ");
scanf("%d",&record_no);
for(i=0;i<2;i++)
   {
      if(rollno[i]==record_no)
      {
         printf("\nRoll no\tName           Average  Grade  \n");
         printf("--------------------------------------------");
         printf("\n%d\t%s             %.2f\t%c",rollno[i],name[i],ave[i],grade[i]);
         found= true;
      }
 }


      if(found != true)

      printf("\n\nStudent details of Roll no %d does not exist.",record_no);


return 0;

}
```

Array contains following limitations:

- Inserting or deleting an element from an array can be inefficient and time-consuming because all the elements after the insertion or deletion point must be shifted to accommodate the change.
- The size of array must be known in advance before using it in the program.
- It is almost impossible to expand the size of the array at run time.
- Arrays have a fixed size that is determined at the time of creation.
  This means that if the size of the array needs to be increased, a new array must be created and the data must be copied from the old array to the new array, which can be time-consuming and memory-intensive.

In short

Array limitations are:-

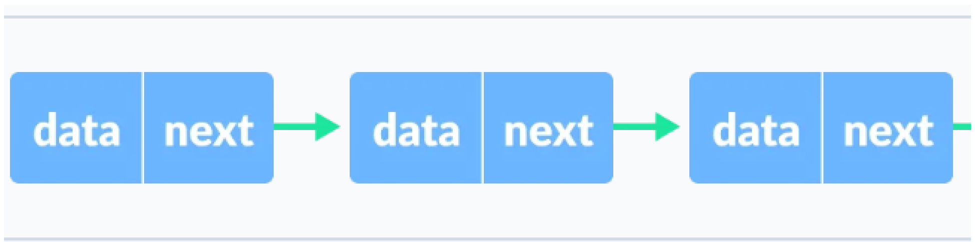
- Fixed size(static)
- Memory allocation issues
- Insertion and deletion issues
- Wasted space

To overcome these issues linked lists are used

Linked lists are used for dynamic memory allocation.

Memory efficient: Memory consumption of a linked list is efficient as its size can grow or shrink dynamically according to our requirements, which means effective memory utilization hence, no memory wastage.

Linked lists



Linked list Data Structure

Linked List

- Linked List can be defined as collection nodes that are randomly stored in the memory
- Linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers.
- Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.

Head and Tail:

The linked list is accessed through the head node, which points to the first node in the list.

The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

Why linked list data structure needed?(Advantages)

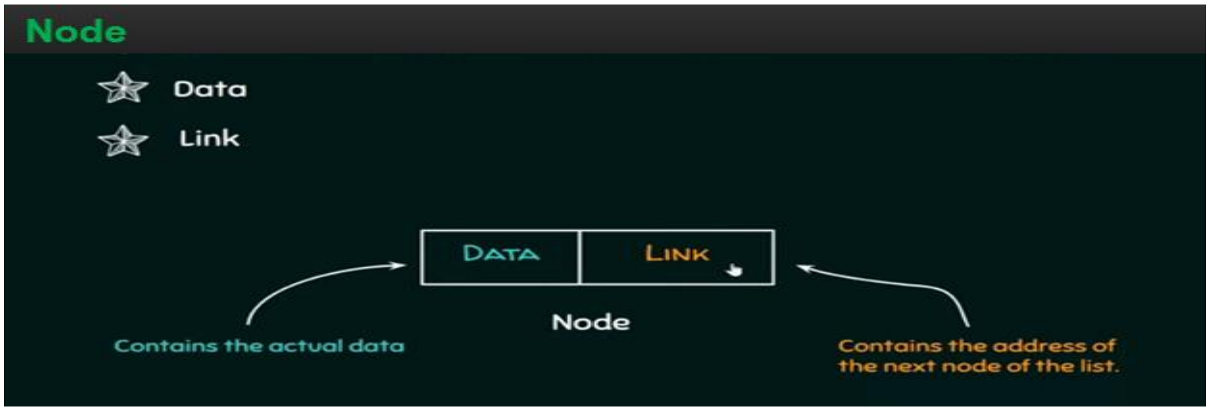
Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.

Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

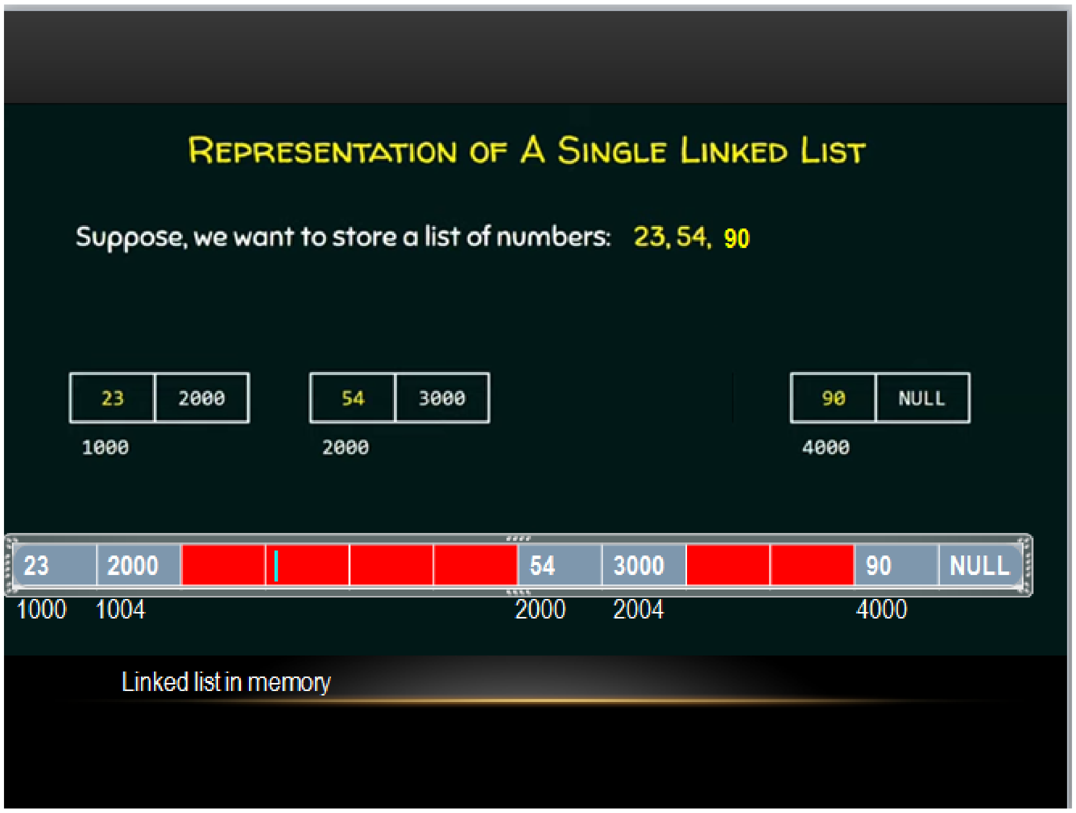
Node Structure:



A node in a linked list typically consists of two components:

Data part : ***stores actual information that is to be represented by the node***

Address part/Link part: It stores the memory address (reference) of the next node in the sequence.

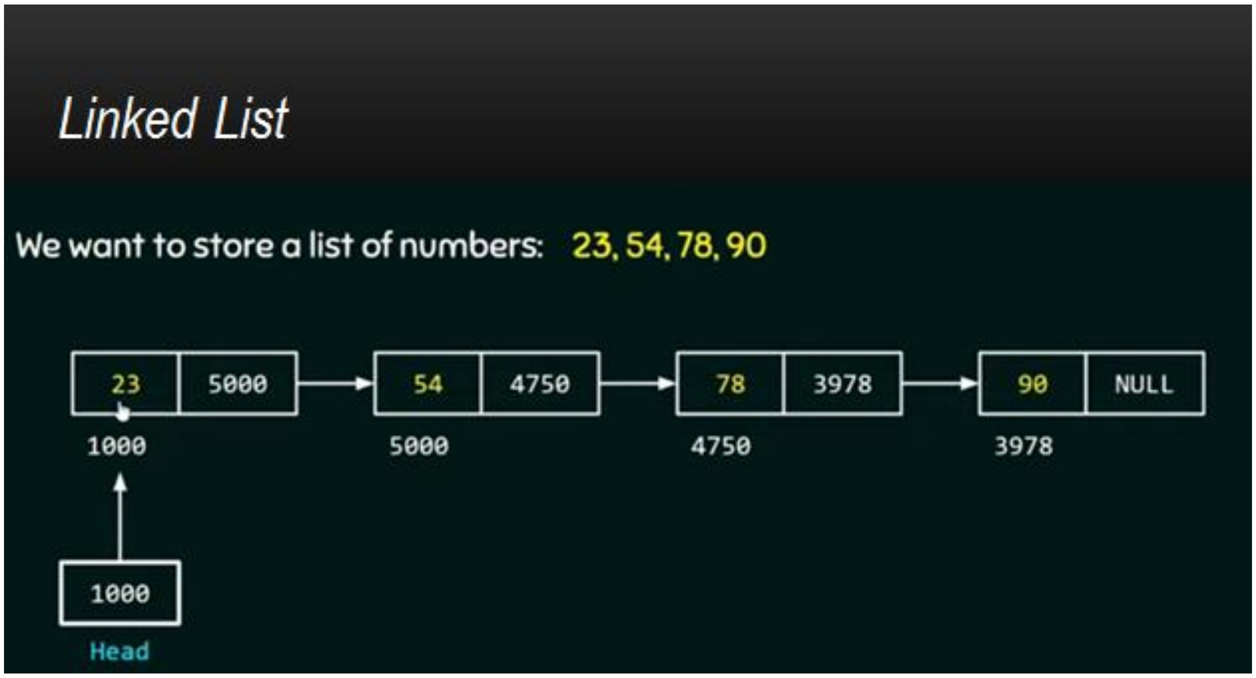


REPRESENTATION OF A SINGLE LINKED LIST

Suppose, we want to store a list of numbers: 23, 54, 90

Linked list in memory

Red blocks are not free



Linked List

We want to store a list of numbers: 23, 54, 78, 90

There are four types of linked lists:

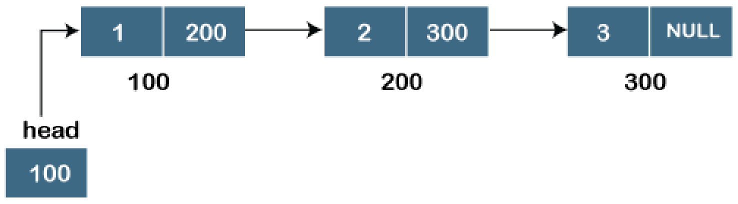
- Singly linked lists.
- Doubly linked lists.
- Circular linked lists.
- Circular doubly linked lists.

1. Singly Linked List

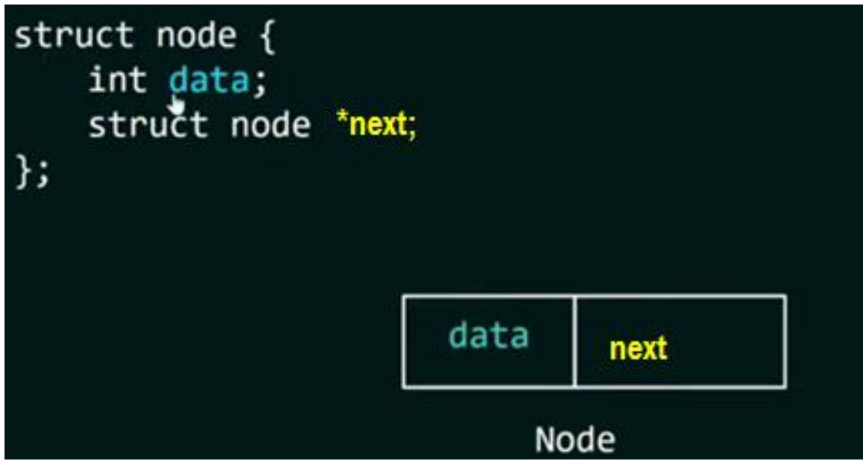
- *It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.*

- The node contains a pointer to the next node means that the node stores the address of the next node in the sequence.

- A single linked list allows the traversal of data only in one way.



Representation of the node in a singly linked list



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

Write a C program to create 3 nodes and connect them manually and display the data.
Or Create simple linked list?

```
#include<stdio.h>
#include<stdlib.h>            //for malloc()
struct node
{
int data;
struct node *next;
};
int main()
{
struct node *N1 = (struct node *)malloc(sizeof(struct node)); //memory allocation
struct node *N2 = (struct node *)malloc(sizeof(struct node));
struct node *N3 = (struct node *)malloc(sizeof(struct node));
struct node *head = (struct node *)malloc(sizeof(struct node));

N1->data = 45 ;//assign data
N2->data = 55 ;
N3->data = 48 ;

N1->next = N2;//Connect nodes
N2->next = N3;
N3->next = NULL;

head = N1;   //save address of Ist node  in head

struct node *temp = (struct node *)malloc(sizeof(struct node));

temp = head;
while(temp!=NULL)
{
  printf("\t%d",temp->data);
  temp = temp->next;
}
return 0;}
```

There are various linked list operations that allow us to perform different actions on linked list .

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

Traverse a Linked List

Display/ access the contents of a linked list is very simple.

We keep moving the temp node to the next one and display its contents.

When `temp` is `NULL`, we know that we have reached the end of the linked list so we get out of the while loop.

```
ALGORITHM

Step 1: [INITIALIZE] SET TEMP = HEAD

Step 2: Repeat Steps 3 and 4 while TEMP!= NULL

Step 3: Apply Process to TEMP->  DATA

Step 4: SET TEMP = TEMP-> NEXT

[END OF LOOP]

Step 5: EXIT
```

```
temp = head;
while(temp!=NULL)
{
  printf("\t%d",temp->data);
  temp = temp->next;
}
```

……………………………………………………………………………………
………………

Using for loop :

```
printf("\n Created Linked list \t:");
             for(temp=head;temp!=NULL;temp=temp->next)
             {
             printf("\t%d",temp->data);
              }
```

……………………………………………………………………………………

create a singly linked list of n nodes ?

Algorithm:

```
Step 1. Set HEAD = NULL, TAIL =NULL ,N=0
Step 2. Repeat steps 3 to 8 until (N<=LIMIT)
Step 3. Create NEW NODE
Step 4. Get VALUE
Step 5. Set DATA (NEWNODE) = VALUE
Step 6. Set NEXT (NEWNODE)= NULL
Step 7. If HEAD= NULL Then
    Set HEAD = NEW NODE,
    TAIL= NEW NODE
Else Set
```

```
    NEXT (TAIL) = NEW NODE ,TAIL =NEWNODE
Step.SET N=N+1
```

create a singly linked list of n nodes and display it.?

```
#include<stdio.h>
#include<stdlib.h>
struct node
        {
        int data;
        struct node *next;
        };

int main()
{
        struct node *newNode;
        struct node *head=(struct node*)malloc(sizeof(struct node));
        struct node *tail=(struct node*)malloc(sizeof(struct node));
        struct node *temp=(struct node*)malloc(sizeof(struct node));
        int i,n=0;
        head=NULL;
        tail=NULL;
        temp=NULL;
        printf("\nEnter no of nodes : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                newNode=(struct node*)malloc(sizeof(struct node));
                printf("\n\nEnter data(integer) for node %d :  ",i);
                scanf("%d",&newNode->data);
                newNode->next=NULL;
                if(head==NULL)
                {
                        head=newNode;
                        tail=newNode;
                }
                else
                {
                        tail->next=newNode;
                        tail=newNode;
                }
        }
                printf("\n Created Linked list \t:");
                for(temp=head;temp!=NULL;temp=temp->next)
                {
                printf("\t%d",temp->data);
             }
                return 0;
        }
```

Singly Linked List :Insertion

Inserting a New Node in a Linked List

When a new node is added into an already existing linked list there are four cases

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

Case 4: The new node is inserted before a given node

Case 1: The new node is inserted at the beginning.
Algorithm

1. Allocate memory for new node
2. Store data
3. If the linked list's head is null, set the new node as the linked list's head and return.
4. Set the new node's next pointer to the current head of the linked list.

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

#include <stdio.h>
#include <stdlib.h>
struct node
{
        int data;
        struct node *next;

}*head,*tail,*temp,*newNode;
void addNodeBegin()  //function

{
        newNode = (struct node*)malloc(sizeof(struct node));
        printf("\tEnter data");
        scanf("%d",&newNode->data);
        newNode->next = NULL;
        if(head==NULL)
        {
                head=newNode;
                tail=newNode;
        }
        else
        {
                newNode->next = head;
                head = newNode;
        }
}
```

Case 2: The new node is inserted at the end.

Algorithm

1. Create a new node with the given data.

2.And make the new node => next as NULL. (Because the new node is going to be the last node.)

3. If the head node is NULL (Empty Linked List), make the new node as the head.
4. If the head node is not null, (Linked list already has some elements),

   find the last node.

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
        int data;
        struct node *next;

}*head,*tail,*temp,*newNode;

void addNodeEnd() //function
{
newNode = (struct node*)malloc(sizeof(struct node));
printf("\n\tEnter data :");
scanf("%d",&newNode->data);
newNode->next = NULL;
if(head==NULL)
          {
            head=newNode;
            tail=newNode;
          }
else
     {
      tail->next=newNode;
      tail = newNode;
     }
}
```

Deleting a Node from a Linked List

There are three cases in deletion.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 1: The first node is deleted.

Point head to the next node i.e. second node

```
temp = head
head = head->next
```

Make sure to free unused memory

```
free(temp);
```

Case 2: The last node is deleted.

Traverse to second last element

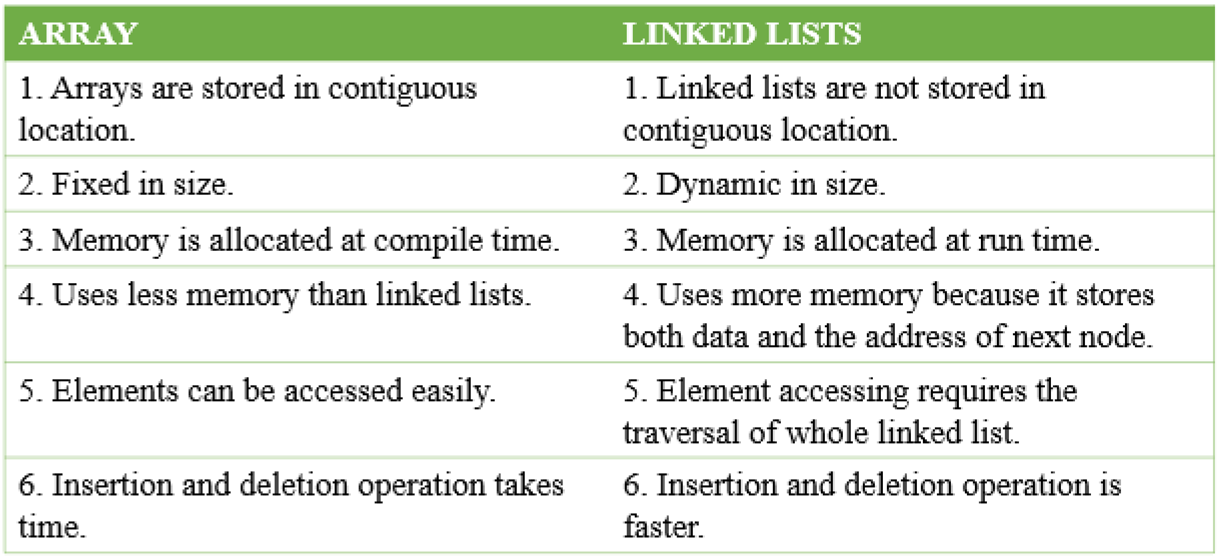
Change its next pointer to null

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

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
	int data;
	struct node *next;

}*head,*tail,*temp,*newNode;

int main()
{
	int n,i;
	printf("\nEnter no of nodes \t : ");
	scanf("%d",&n);
	for(i=1;i<=n;i++)
	{
		newNode=(struct node*)malloc(sizeof(struct node));
		printf("\n\nEnter data(integer) for node %d :  ",i);
		scanf("%d",&newNode->data);
		newNode->next=NULL;
		if(head==NULL)
		{
			head=newNode;
			tail=newNode;
		}
		else
		{
			tail->next=newNode;
			tail=newNode;
```

```
                }
        }
        printf("List before delete:");
        temp=head;
                while(temp!=NULL)
                {
                        printf("\t%d",temp->data);
                        temp=temp->next;

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

       printf("\nList After delete:");
           temp=head;
                while(temp!=NULL)
                {
                        printf("\t%d",temp->data);
                        temp=temp->next;

                }
}
```

Linked List vs Array



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Disadvantages Of Linked List:

- Memory usage: More memory is required in the linked list as compared to an array

  require additional memory cost to store the pointers connecting each node to the next node..

- Traversal: In a Linked list traversal is more time-consuming as compared to an array.

  To reach a particular node, we need to traverse the nodes present before it; hence traversing in the linked List is a time taking process.

  We cannot directly access a specific node in the Linked List because of the absence of an index approach.

- In a singly linked list reverse traversing is not possible
- more complex implementation.