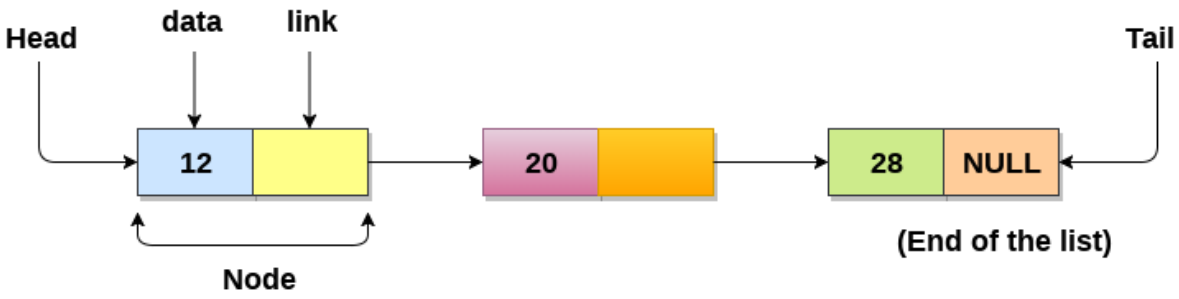
SECTION-III
Linked List

Single Linked List:

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.
Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while

the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structure	Time Complexity	Space Compleity

	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
```

```
5. };
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different

		logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just

		need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Linked List in C: Menu Driven Program

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
```

```
8. struct node *head;
9.
10.     void beginsert ();
11.     void lastinsert ();
12.     void randominsert();
13.     void begin_delete();
14.     void last_delete();
15.     void random_delete();
16.     void display();
17.     void search();
18.     void main ()
19.     {
20.         int choice =0;
21.         while(choice != 9)
22.         {
23.             printf("\n\n*********Main
    Menu*********\n");
24.             printf("\nChoose one option from the following
    list ...\n");
25.             printf("\n==========================
    ====================\n");
26.             printf("\n1.Insert in begining\n2.Insert at
    last\n3.Insert at any random location\n4.Delete from
    Beginning\n
27.             5.Delete from last\n6.Delete node after
    specified location\n7.Search for an
    element\n8.Show\n9.Exit\n");
28.             printf("\nEnter your choice?\n");
29.             scanf("\n%d",&choice);
30.             switch(choice)
31.             {
32.                 case 1:
33.                 beginsert();
```

```
34.             break;
35.             case 2:
36.             lastinsert();
37.             break;
38.             case 3:
39.             randominsert();
40.             break;
41.             case 4:
42.             begin_delete();
43.             break;
44.             case 5:
45.             last_delete();
46.             break;
47.             case 6:
48.             random_delete();
49.             break;
50.             case 7:
51.             search();
52.             break;
53.             case 8:
54.             display();
55.             break;
56.             case 9:
57.             exit(0);
58.             break;
59.             default:
60.             printf("Please enter valid choice..");
61.         }
62.     }
63. }
64. void beginsert()
65. {
66.     struct node *ptr;
```

```
67.         int item;
68.         ptr = (struct node *) malloc(sizeof(struct node
    *));
69.         if(ptr == NULL)
70.         {
71.             printf("\nOVERFLOW");
72.         }
73.         else
74.         {
75.             printf("\nEnter value\n");
76.             scanf("%d",&item);
77.             ptr->data = item;
78.             ptr->next = head;
79.             head = ptr;
80.             printf("\nNode inserted");
81.         }
82.
83.     }
84.     void lastinsert()
85.     {
86.         struct node *ptr,*temp;
87.         int item;
88.         ptr = (struct node*)malloc(sizeof(struct
    node));
89.         if(ptr == NULL)
90.         {
91.             printf("\nOVERFLOW");
92.         }
93.         else
94.         {
95.             printf("\nEnter value?\n");
96.             scanf("%d",&item);
97.             ptr->data = item;
```

```
98.             if(head == NULL)
99.             {
100.               ptr -> next = NULL;
101.               head = ptr;
102.               printf("\nNode inserted");
103.            }
104.            else
105.            {
106.               temp = head;
107.               while (temp -> next != NULL)
108.               {
109.                  temp = temp -> next;
110.               }
111.               temp->next = ptr;
112.               ptr->next = NULL;
113.               printf("\nNode inserted");
114.
115.            }
116.         }
117.      }
118.      void randominsert()
119.      {
120.         int i,loc,item;
121.         struct node *ptr, *temp;
122.         ptr = (struct node *) malloc (sizeof(struct node));
123.         if(ptr == NULL)
124.         {
125.            printf("\nOVERFLOW");
126.         }
127.         else
128.         {
129.            printf("\nEnter element value");
130.            scanf("%d",&item);
```

```
131.            ptr->data = item;
132.            printf("\nEnter the location after which you
    want to insert ");
133.            scanf("\n%d",&loc);
134.            temp=head;
135.            for(i=0;i<loc;i++)
136.            {
137.               temp = temp->next;
138.               if(temp == NULL)
139.               {
140.                  printf("\ncan't insert\n");
141.                  return;
142.               }
143.
144.            }
145.            ptr ->next = temp ->next;
146.            temp ->next = ptr;
147.            printf("\nNode inserted");
148.         }
149.      }
150.      void begin_delete()
151.      {
152.         struct node *ptr;
153.         if(head == NULL)
154.         {
155.            printf("\nList is empty\n");
156.         }
157.         else
158.         {
159.            ptr = head;
160.            head = ptr->next;
161.            free(ptr);
```

```
162.            printf("\nNode deleted from the begining
    ...\n");
163.        }
164.    }
165.    void last_delete()
166.    {
167.        struct node *ptr,*ptr1;
168.        if(head == NULL)
169.        {
170.            printf("\nlist is empty");
171.        }
172.        else if(head -> next == NULL)
173.        {
174.            head = NULL;
175.            free(head);
176.            printf("\nOnly node of the list deleted ...\n");
177.        }
178.
179.        else
180.        {
181.            ptr = head;
182.            while(ptr->next != NULL)
183.            {
184.                ptr1 = ptr;
185.                ptr = ptr ->next;
186.            }
187.            ptr1->next = NULL;
188.            free(ptr);
189.            printf("\nDeleted Node from the last ...\n");
190.        }
191.    }
192.    void random_delete()
193.    {
```

```
194.        struct node *ptr,*ptr1;
195.        int loc,i;
196.        printf("\n Enter the location of the node after
   which you want to perform deletion \n");
197.        scanf("%d",&loc);
198.        ptr=head;
199.        for(i=0;i<loc;i++)
200.        {
201.           ptr1 = ptr;
202.           ptr = ptr->next;
203.
204.           if(ptr == NULL)
205.           {
206.              printf("\nCan't delete");
207.              return;
208.           }
209.        }
210.        ptr1 ->next = ptr ->next;
211.        free(ptr);
212.        printf("\nDeleted node %d ",loc+1);
213.     }
214.     void search()
215.     {
216.        struct node *ptr;
217.        int item,i=0,flag;
218.        ptr = head;
219.        if(ptr == NULL)
220.        {
221.           printf("\nEmpty List\n");
222.        }
223.        else
224.        {
```

```
225.          printf("\nEnter item which you want to
   search?\n");
226.          scanf("%d",&item);
227.          while (ptr!=NULL)
228.          {
229.             if(ptr->data == item)
230.             {
231.                printf("item found at location %d ",i+1);
232.                flag=0;
233.             }
234.             else
235.             {
236.                flag=1;
237.             }
238.             i++;
239.             ptr = ptr -> next;
240.          }
241.          if(flag==1)
242.          {
243.             printf("Item not found\n");
244.          }
245.       }
246.
247.    }
248.
249.    void display()
250.    {
251.       struct node *ptr;
252.       ptr = head;
253.       if(ptr == NULL)
254.       {
255.          printf("Nothing to print");
256.       }
```

```
257.        else
258.        {
259.            printf("\nprinting values . . . . .\n");
260.            while (ptr!=NULL)
261.            {
262.                printf("\n%d",ptr->data);
263.                ptr = ptr -> next;
264.            }
265.        }
266.    }
267.
```

Output:

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
2

Enter value?
2

Node inserted

*********Main Menu*********

Choose one option from the following list ...

```
==========================================
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
3

Enter element value1

Enter the location after which you want to insert 1

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========================================
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
```

4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
8

printing values . . . . .

1
2
1

**********Main Menu**********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

```
Enter your choice?
2

Enter value?
123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========================================
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
1

Enter value
1234

Node inserted
```

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
4

Node deleted from the begining ...

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning

5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
5

Deleted Node from the last ...

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
6

Enter the location of the node after which you want to perform deletion

```
1

Deleted node 2

*********Main Menu*********

Choose one option from the following list ...

==========================================
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
8

printing values . . . . .

1
1

*********Main Menu*********

Choose one option from the following list ...
```

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
7

Enter item which you want to search?
1
item found at location 1
item found at location 2

*********Main Menu*********

Choose one option from the following list ...

==========
===

1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last

6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
9

Representation in memory:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head of the linked list. If the linked list is empty, then the value of the head points to NULL.
Each node in a list consists of at least two parts:

- A Data Item (we can store integer, strings, or any type of data).
- Pointer (Or Reference) to the next node (connects one node to another) or An address of another node

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.
In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

- C
- C++
- Java
- Python
- C#
- Javascript

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

Construction of a simple linked list with 3 nodes:

- C
- C++
- Java
- Python
- C#
- Javascript

```
// C program to implement a
// linked list
#include <stdio.h>
#include <stdlib.h>

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

// Driver's code
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    /* Three blocks have been allocated dynamically.
     We have pointers to these three blocks as head,
     second and third
      head        second          third
       |           |           |
       |           |           |
    +---+-----+  +----+----+     +----+----+
    |# |# |     |# |# |   | #| #|
    +---+-----+  +----+----+     +----+----+
```

```
# represents any random value.
Data is random because we haven't assigned
anything yet  */

 head->data = 1; // assign data in first node
 head->next = second; // Link first node with
 // the second node

 /* data has been assigned to the data part of the first
  block (block pointed by the head). And next
  pointer of first block points to second.
  So they both are linked.

    head        second          third
     |           |            |
     |           |            |
  +---+---+    +----+----+     +-----+----+
  | 1  | o----->| #  | #  |    |  #  | #  |
  +---+---+    +----+----+     +-----+----+
*/

 // assign data to second node
 second->data = 2;

 // Link second node with the third node
 second->next = third;

 /* data has been assigned to the data part of the second
  block (block pointed by second). And next
  pointer of the second block points to the third
  block. So all three blocks are linked.
```

```
  head        second          third
   |           |              |
   |           |              |
+---+---+    +---+---+ +----+----+
| 1  | o----->| 2 | o-----> |  # |  # |
+---+---+    +---+---+ +----+----+      */

third->data = 3; // assign data to third node
third->next = NULL;

/* data has been assigned to data part of third
block (block pointed by third). And next pointer
of the third block is made NULL to indicate
that the linked list is terminated here.

 We have the linked list ready.

      head
        |
        |
   +---+---+ +---+---+       +----+------+
   | 1  | o----->|  2  | o-----> |  3 | NULL |
   +---+---+ +---+---+       +----+------+


   Note that only head is sufficient to represent
   the whole list.  We can traverse the complete
   list by following next pointers.   */

   return 0;
}
```