:: UNIT – 4 ::

TOPIC ➔ Link List

CONTENTS:

❖ **Introduction:**

- Same as an array, we can use linked list to store similar type of elements in memory. An array is simple to understand and elements of an array are easily accessible. But there are some limitations of an array as follows:

❖ **Limitations of an Array:**

- Size is fixed: Once the size of an array is decided it cannot be increased or decreased during execution. So, array is known as Static Data Structure.
- Array Items are stored contiguously: The array elements are stored in sequential manner. That means it stores data in computer memory in contiguous memory location.
- Operations like insertion of a new element in an array or deletion of an existing form the array are complex and tedious.

:: UNIT – 4 ::

❖ Linked List:

A linked list is a collection of nodes with various fields. It contains ***Data Field*** and ***Address Field***. It is a linear type data structure. The Linked list is defined as an ordered collection of homogenous data elements. The each element of linked list is known as NODE. The linked list overcomes all these disadvantages. A linked list can grow and reduce in size during its lifetime. For this reason linked list is known as Dynamic Data Structure. That means there is no maximum size of linked list. The advantages of linked list are:

- Size is not fixed.
- Data can be stored at any place.
- Insertions and deletions are simple and faster.

The Linked List types are:

- Singly Linked List
- Circular Singly Linked List
- Doubly Linked List
- Circular Doubly Linked List

❖ Singly Linked List:

- The singly linked list is the most basic of all the linked data structures.
- A sequence of dynamically allocated nodes (items/objects) is known as singly linked list.
- Each node refers to its successor in the list.
- A singly linked list is a collection of nodes and each node contains a pointer to the next element.
- In singly linked list, we can move forward only. That means we can move to the next node but return back is not possible.
- The node can be of integer, character, float, structure etc.
- Each node in list contains two parts: Information Part and Pointer to Next Node.

Information Part	Pointer to next node

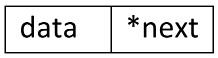
- Building / Creating a linked list:

  To create a linked list, following steps are required:
  - Declare the structure that defines the node parts (elements).
  - Declare the variables *start or *temp or *node as per requirement.
  - Assign start = NULL which denotes that linked list is empty.
  - Find each node.
    - Find end of the list so that temp->next = NULL.
    - Allocate memory for the new entry by assigning it to temp->next.
    - Assign the member values to node.
    - Assign temp->next, the value NULL to indicate the end of the list.

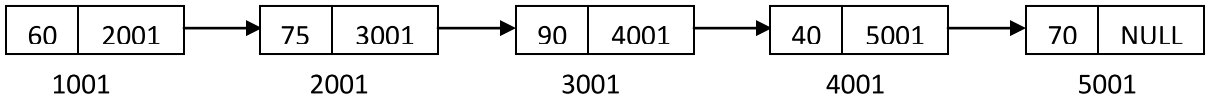
The linked list can be defined as:

```
struct node
{
        int data;                   // The Information Part
        struct node *next;        // Pointer to next node.
} *start;
```

We can describe the above definition of structure (node) in graphical way as follow:



Here, the data is used to store the information or actual value and *next stores the address of the successor (next) node in the list. The *next is used to maintain the link of the node with another node. The following figure shows the linked list which stores the marks obtained by the students:



A linked list also contains a list pointer variable called "start" which contains the address of the first node in the list. The $\longrightarrow$ is used to link between the two nodes. If there is no node in the list, the list is called NULL list or empty list.

- **Advantages:**
  - Memory efficiency is very high.
  - Insertion and deletion is very easy and fast.

- **Disadvantages:**
  - Traversal is very slow.
  - The singly linked list is one directional only. That means we can move forward only.
  - If one link is corrupted remaining data will be lost.

:: UNIT – 4 ::

Menu Driven Singly Linked List Program which performs following operations:

1) Create 2) Display / Traverse 3) Count 4) Quit / Exit

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

int count = 0;
void create(struct node *);
void display(struct node *);

void main()
{
        int ch;
        start = NULL;
        clrscr();
        while(1)
        {
                clrscr();
                printf("\n\t1. => Create ");
                printf("\n\t2. => Display ");
                printf("\n\t3. => Count ");
                printf("\n\t0. => Exit ");
                printf("\n\t====> Enter Your Choice : ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                create(start);
                                break;
                        case 2:
                                display(start);
                                break;
                        case 3:
                                printf("\n\t Total No. of Nodes are : %d",count);
                                break;
                        case 0:
```

```
                        exit(0);
                default:
                        printf("\n\t Invalid Choice : ");
                        break;
            }
        getch();
        }
}
void create (struct node *temp)
{
        if(temp == NULL)
        {
                temp = (struct node*) malloc(sizeof(struct node));
                start = temp;
        }
        else
        {
                while(temp->next != NULL)
                {
                        temp = temp->next;
                }
                temp->next =(struct node*) malloc(sizeof(struct node));
                temp = temp->next;
        }
        printf("\n\tEnter Value : ");
        scanf("%d",&temp->data);
        temp->next = NULL;
        count++;
}

void display(struct node *temp)
{
        printf("\n\nTHE LINKED LIST IS :: \n");
        while(temp != NULL)
        {
                printf("\n\t%d",temp->data);
                temp = temp->next;
        }
}
```

FUNCTION TO SEARCH THE NODE FROM THE LIST

```
void search(struct node *temp)
{
        int key,pos=1,flag=0;
        printf("\n\tEnter number to search from the list : ");
        scanf("%d",&key);
        if(temp == NULL)
                printf("\n\tList is empty...");
        else
        {
                while(temp != NULL)
                {
                        //printf("%d\n",temp->data);
                        if(key == temp->data)
                        {
                                printf("\n\tElement %d is found at %d position",key,pos);
                                flag = 1;
                        }
                        temp = temp->next;
                        pos++;
                }
                if(!flag)
                {
                        printf("\n\tElement is not available in the list");
                }
        }
}
```

FUNCTIONS TO CREATING OF NODE WITH DIFFERENT WAYS:

1) Insert / Create *at First Position*
2) Insert / Create *at Last Position*
3) Insert / Create *at Specific Position*
4) Insert / Create *after Specific Position*

```
void insert_at_first()
{       struct node *temp;
        temp = (struct node *) malloc (sizeof(struct node));
        printf("\n\tEnter Value : ");
        scanf("%d",&temp->data);
        temp->next =start;
        start = temp;
}
```

```
void insert_at_last()
{
	struct node *temp;
	temp = start;
	while(temp->next != NULL)
	{		temp = temp->next;		}
	temp->next =(struct node*) malloc(sizeof(struct node));
	temp = temp->next;

	printf("\n\tEnter Value : ");
	scanf("%d",&temp->data);
	temp->next = NULL;
	count++;
}
void insert_at_spec()
{
	int pos,i;
	struct node *temp,*new1;
	temp = start;
	printf("\n\tEnter Position: ");
	scanf("%d",&pos);
	i=1;
	while(temp != NULL)
	{
		if(pos == i+1)
		{
			new1 = (struct node*)malloc(sizeof(struct node));
			printf("\n\tEnter Data : ");
			scanf("%d",&new1->data);
			new1->next = temp->next;
			temp->next = new1;
			break;
		}
		i++;
		temp = temp->next;
	}
}
void insert_aft_spec()
{
	int pos,i;
	struct node *temp,*new1;
	temp = start;
```

```
        printf("\n\tEnter Position: ");
        scanf("%d",&pos);
        i=1;
        while(temp != NULL)
        {
                if(pos == i)
                {
                        new1 = (struct node*)malloc(sizeof(struct node));
                        printf("\n\tEnter Data : ");
                        scanf("%d",&new1->data);
                        new1->next = temp->next;
                        temp->next = new1;
                }
                i++;
                temp = temp->next;
        }
}
```

FUNCTIONS TO DISPLAYING THE NODES WITH DIFFERENT WAYS:

1) Display *from Last Position*
2) Display *from Specific Position*
3) Display *after Specific Position*
4) Display *odd number of nodes Value*
5) Display *odd Values of nodes*
6) Display *from Specific Value*
7) Display *from Specific Range*

```
void disp_frm_last(struct node *temp)
{
        int node[100],i=0,j;
        while(temp!=NULL)
        {
                node[i]  = temp ->data;
                temp = temp->next;
                i++;
        }

        for(j=i-1;j>=0;j--)
                printf("\n\t%d",node[j]);
}

void disp_spec(struct node *temp)
{
        int pos,cnt=1;
        printf("\n\tEnter Position : ");
        scanf("%d",&pos);
        while(temp != NULL)
        {
                if(cnt >= pos)
                        printf("\n\t%d",temp->data);
                temp = temp->next;
                cnt++;
        }
}

void disp_aft_spec(struct node *temp)
{
        int pos,cnt=0;
        printf("\n\tEnter Position : ");
        scanf("%d",&pos);
```

```
		while(temp != NULL)
		{
			if(cnt >= pos)
				printf("\n\t%d",temp->data);
			temp = temp->next;
			cnt++;
		}

	}
	void odd_node(struct node *temp)
	{
		int cnt=1;
		while(temp!=NULL)
		{
			if(cnt%2 != 0)
				printf("\n\t%d",temp->data);
			temp=temp->next;
			cnt++;
		}
	}
	void odd_value(struct node *temp)
	{
		while(temp!=NULL)
		{
			if(temp->data%2 != 0)
				printf("\n\t%d",temp->data);
			temp=temp->next;
		}
	}
	void disp_frm_spec_value(struct node *temp)
	{
		int val;
		struct node *t1;
		printf("\n\tEnter Value : ");
		scanf("%d",&val);
		while(temp != NULL)
		{
			if(val == temp->data)
			{
				t1 = temp;
				break;
			}
```

```
                temp = temp -> next;
        }
        while(t1 != NULL)
        {
                printf("\n\t%d",t1->data);
                t1 = t1->next;
        }
}

void disp_range(struct node*temp)
{
        int sr,er;
        printf("\n\tEnter Starting Range : ");
        scanf("%d",&sr);
        printf("\n\tEnter Ending Range : ");
        scanf("%d",&er);
        while(temp!=NULL)
        {
                if(temp->data >=sr && temp->data <= er)
                        printf("\n\t%d",temp->data);
                temp=temp->next;
        }
}
```

FUNCTIONS TO DELETING THE NODES WITH DIFFERENT WAYS:

1) Delete *First Node*
2) Delete *Last Node*
3) Delete Node *By Position*
4) Delete Node *By Value*
5) Delete Node *By Range*

```
void del_first(struct node *temp)
{
        if(temp == NULL)
                printf("\n\tLinked List is Empty...");
        else
        {
                start = temp -> next;
                printf("\n\tDeleted Node Value is %d",temp->data);
                count--;
        }
}
```

```
void del_last(struct node*temp)
{
        int i;
        while(temp->next!=NULL)
                temp = temp->next;
        printf("\n\tDeleted Node Value is %d",temp->data);
        count--;
        temp = start;
        for(i=1;i<count;i++)
                temp=temp->next;
        temp->next = NULL;
}

void del_by_pos(struct node *temp)
{
        int pos,cnt=1;
        struct node *prev;
        printf("\n\tEnter Position : ");
        scanf("%d",&pos);
        if(pos == 1)
        {
                printf("\n\tDeleted Node is %d",temp->data);
                start = temp->next;
        }
        else
        {
                while(temp!=NULL)
                {
                        if(cnt == pos)
                        {
                            printf("\n\tDeleted Node is %d",temp->data);
                            prev ->next = temp;
                            break;
                        }
                        prev = temp;
                        temp = temp->next;
                        cnt++;
                }
        }
}
```

```
void del_by_val(struct node *temp)
{
	int no,i;
	struct node *prev;
	prev=temp;
	printf("\n\n\t\tEnter Number you want to delete:");
	scanf("%d",&no);
	if(temp->data == no && temp->next !=NULL)
	{
		printf("\n\n\t\tDeleted element is %d",temp->data);
		start = start->next;
	}
	else if(temp->data == no)
	{
		printf("\n\n\t\tDeleted element is %d",temp->data);
		start = NULL;
	}
	else
	{
		while(temp!=NULL)
		{
			if(temp->data==no)
			{
				printf("\n\n\t\tDeleted element is %d",temp->data);
				prev->next = temp->next;
				break;
			}
			prev = temp;
			temp=temp->next;
		}
		temp=start;
		display(start);
	}
}

void del_bet_range(struct node *temp)
{
	int sr,er;
	printf("\n\tEnter Starting Range : ");
	scanf("%d",&sr);
	printf("\n\tEnter Ending Range : ");
	scanf("%d",&er);
```

```
        while(temp != NULL)
        {
                if(temp->next->data >=sr && temp->next->data <= er)
                {
                        printf("\n\tDeleted Node is %d",temp->next->data);
                        temp->next = temp->next->next;
                }
                else
                        temp = temp->next;
        }
}
```

<u>FUNCTIONS TO SORT THE NODES WITH DIFFERENT WAYS:</u>

1) Sorting *<u>By Ascending Order</u>*
2) Sorting *<u>By Descending Order</u>*

```
void asc_sort(struct node*temp)
{
        int b,i,j,cnt=count;
        struct node *a;
        for(i=1;i<=count;i++)
        {
                a=temp->next;
                for(j=1;j<cnt;j++)
                {
                        if(temp->data > a->data)
                        {
                                b=temp->data;
                                temp->data=a->data;
                                a->data=b;
                        }
                        a=a->next;
                }
                printf("\n\t%d",temp->data);
                temp=temp->next;
                cnt--;
        }
}
void desc_sort(struct node *temp)
{
        int b,i,j,cnt=count;
        struct node *a;
        for(i=1;i<=count;i++)
        {
                a=temp->next;
                for(j=1;j<cnt;j++)
                {
                        if(temp->data < a->data)
                        {
                                b=temp->data;
                                temp->data=a->data;
                                a->data=b;
                        }
                        a=a->next;
                }
                printf("\n\t%d",temp->data);
                temp=temp->next;
                cnt--;
        }
}
```

<u>FUNCTIONS TO UPDATE THE NODES</u>

1) Update *<u>First Node</u>*
2) Update *<u>Last Node</u>*
3) Update Node *<u>By Value</u>*
4) Update Node *<u>By Position</u>*

```
void update_first()
{
        if(start == NULL)
                printf("\n\tLIST IS EMPTY....");
        else
        {
                printf("\n\tEnter New Value for First Node : ");
                scanf("%d",&start->data);
        }
}
void update_last(struct node *temp)
{
        if(temp == NULL)
                printf("\n\tLIST IS EMPTY....");
        else
        {
                while(temp->next!=NULL)
                        temp=temp->next;
                printf("\n\tEnter New Value for Last Node : ");
                scanf("%d",&temp->data);
        }
}
void update_by_value(struct node *temp)
{
        int val;
        if(temp ==NULL)
                printf("\n\tLIST IS EMPTY...");
        else
        {
                printf("\n\tEnter Old value that you want to update : ");
                scanf("%d",&val);
                while(temp!= NULL)
                {
                        if(temp->data == val)
                        {
                                printf("\n\tThe Old Value of this node is %d", temp->data);
                                printf("\n\tEnter New Value : ");
                                scanf("%d",&temp->data);
                        }
                        else
                                temp = temp->next;
                }
        }
}

void update_by_pos(struct node *temp)
{
        int pos,i=1;
        if(temp == NULL)
                printf("\n\tLIST IS EMPTY...");
        else
        {
                printf("\n\tEnter Position : ");
                scanf("%d",&pos);
                while(temp != NULL)
                {
                        if(pos == i)
                        {
                                printf("\n\tThe Old Value at pos %d is %d",pos,temp->data);
                                printf("\n\tEnter New Value : ");
                        scanf("%d",&temp->data);
                        break;
                    }
                    else
                    {
                        temp = temp->next;
                        i++;
                    }
                }
            }
}
```

<u>FUNCTION TO REVERSE THE LINKED LIST</u>

```
void reverse(struct node* temp)
{
        struct node *previous,*t1;
        previous = NULL;
        while(temp !=NULL)
        {
                t1 = temp ->next;
                temp->next = previous;
                previous = temp;
                temp = t1;
        }
        if(temp == NULL)
                start = previous;
}
```

❖ **<u>Doubly Linked List:</u>**

- In singly linked list, the traversing is possible only in one direction which is the limitation sometime when we want to traverse the list in both directions.
- The Doubly Linked List provides this facility which increases the performance and efficiency of algorithms.
- To traverse the nodes in both directions requires a node with two pointers, in which one pointer points to the previous node and another pointer points to the next node.
- So, each node is having three fields:
    - Pointer to Previous Node
    - Information Part
    - Pointer to Next Node

Pointer to previous node	Information Part	Pointer to next node

- <u>Building / Creating a doubly linked list:</u>

  To create a doubly linked list, following steps are required:
  - Declare the structure that defines the node parts (elements).
  - Declare the variables *start or *temp or *node as per requirement.
  - Assign start = NULL which denotes that linked list is empty.
  - Find each node.
    - Find end of the list so that temp->next = NULL.
    - Assign Previous node address (or NULL if List is empty) to temp -> previous.
    - Allocate memory for the new entry by assigning it to temp->next.
    - Assign the member values to node.
    - Assign temp->next, the value NULL to indicate the end of the list.

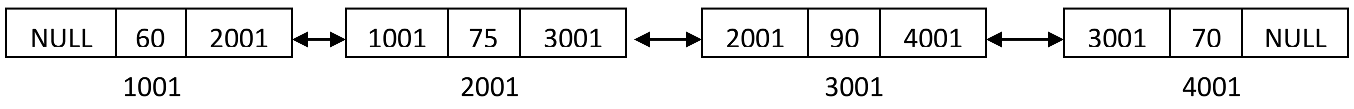
The linked list can be defined as:

```
struct node
{
        int data;                       // The Information Part
        struct node *next, *prev;       //Pointer to next node & Pointer to previous node.
} *start;
```

We can describe the above definition of structure (node) in graphical way as follow:

*prev	data	*next

Here, the data is used to store the information or actual value and *next stores the address of the successor (next) node in the list. The *next is used to maintain the link of the node with another node. The *prev stores the address of the predecessor (previous) node in the list. If the Node is first node then *prev has NULL. The following figure shows the linked list which stores the marks obtained by the students:



❖ **<u>Circular Singly Linked List:</u>**
- A linked list in which the last node points to the first node is known as CIRCULAR LINKED LIST.
- The Circular Singly Linked List is modified version of Singly Linked List.

- It works same as singly linked list except it has no end – that means the there isn't any node which contains NULL.
- Each node in list contains two parts: Information Part and Pointer to Next Node.

Information Part	Pointer to next node

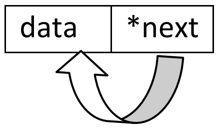
- Building / Creating a linked list:

  To create a linked list, following steps are required:
  - Declare the structure that defines the node parts (elements).
  - Declare the variables *start or *temp or *node as per requirement.
  - Assign start = NULL which denotes that linked list is empty.
  - Find each node.
    - Find end of the list so that temp->next != start (as this is the circular and last node contains the address of first node).
    - Allocate memory for the new entry by assigning it to temp->next.
    - Assign the member values to node.
    - Assign temp->next, the value *start* to indicate the end of the list.

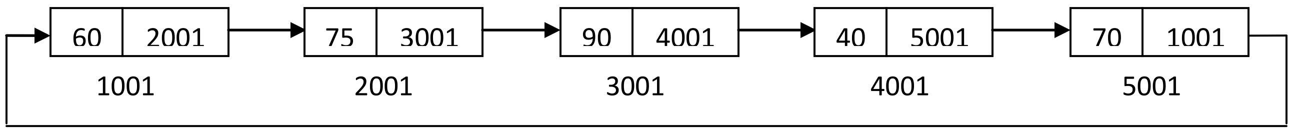
  The linked list can be defined as:

```
struct node
{
        int data;                    // The Information Part
        struct node *next;         // Pointer to next node.
} *start;
```

  We can describe the above definition of structure (node) in graphical way as follow:



- The *next can be used to point the last / next node in the list.
- In this type of linked list, *next of last node has START.



- Advantages of Circular Linked List over Singly Linked List:
    - In circular list every node is accessible from given node.
- Disadvantages:
    - It is possible to get into an infinite loop if care is not taken.

❖ **Circular Doubly Linked List:**

- The circular doubly linked list is a modified version of doubly linked list and works in same fashion but it has no end, compare to doubly linked list.
- In circular doubly linked list, the first node's previous has last node's address and last node's next has first node's address.
- To traverse the nodes in both directions requires a node with two pointers, in which one pointer points to the previous node and another pointer points to the next node.
- So, each node is having three fields:
    - Pointer to Previous Node
    - Information Part
    - Pointer to Next Node

Pointer to previous node	Information Part	Pointer to next node

- Building / Creating a circular doubly linked list:

  To create a doubly linked list, following steps are required:
    - Declare the structure that defines the node parts (elements).
    - Declare the variables *start or *temp or *node as per requirement.
    - Assign start = NULL which denotes that linked list is empty.
    - Find each node.
        - Find end of the list so that temp->next != start.
        - Assign Previous node address (or NULL if List is empty) to temp -> previous.
        - Allocate memory for the new entry by assigning it to temp->next.
        - Assign the member values to node.
        - Assign temp->next, the value start to indicate the end of the list.

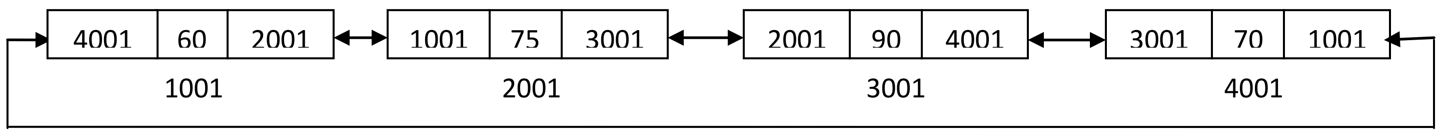
  The linked list can be defined as:

```
struct node
{
        int data;                          // The Information Part
        struct node *next, *prev;     //Pointer to next node & Pointer to previous node.
} *start;
```

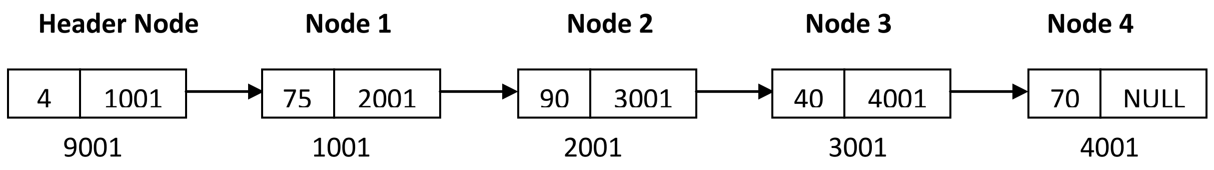
  We can describe the above definition of structure (node) in graphical way as follow:

*prev	data	*next

The following figure shows the circular linked list.



- **Header Linked List:**
  - A linked list which contains an additional node – header node is known as Header Linked List.
  - A header node is a special node which is found always at the front of the list.
  - A header node contains the total number of nodes in the list. And the header node's *next (i.e. header -> next) stores the address of the first node (i.e. *start).



  - This type of linked list is similar to the singly or doubly linked list except that it contains one additional node that is header node.
  - All the possible operations that we can perform in singly or doubly linked list, is possible in header linked list.
  - The Header node is created when the first node of the list is created.
  - Header Linked List are of two types: 1) Grounded Header Linked List and 2) Circular Header Linked List.

- **Applications of Linked List:**

  The linked lists are used in different areas. The some of are:
  - In Operating System while allocating block of memory.
  - In multi-user system the OS may keep track of user jobs waiting to execute through linked queue of control block.
  - With the use of linked list, it is easy to implement stack and queue.

- **DIFFERENCES:**

  Also check the differences of:
  1. Doubly Linked List Vs. Singly Linked List
  2. Circular Linked List Vs. Singly Linked List
  3. Linked List Vs. Header Linked List