# 9. Linked List:

- A *linked list*, or *one-way list*, is a linear collection of data elements, called *nodes*, where the linear order is given by means of *pointers*.
- Each node is divided into two parts: the first part contains the information of the element, and the second part, called the *link field* or next-*pointer field*, contains the address of the next node in the list.

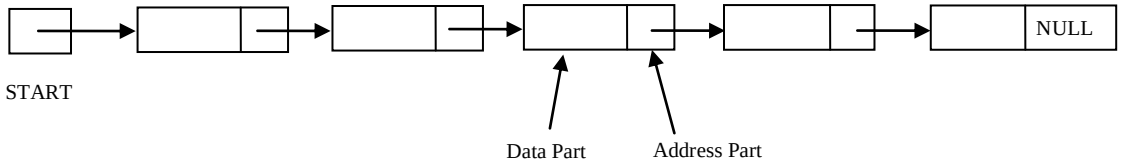


**Fig.1 Dynamic Implementation of Singly Linked List with 5 nodes**

- In *figure 1*, each node is pictured with two parts. The left part represents the information part of the node and second part represents the next-pointer field of the node.

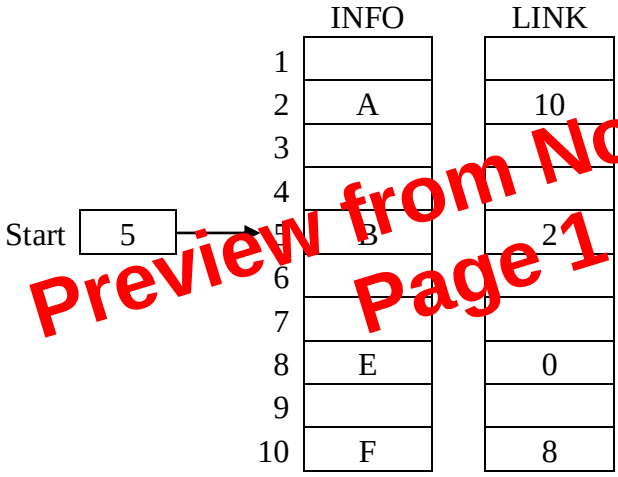




**Fig.2 Array Implementation of Singly Linked List with 4 elements.**

- In *figure 2*, there is array representation of LIST. LIST requires two linear arrays (INFO array and LINK array). INFO array stores the elements of LIST and LINK array stores the position of next element. There is a *start* variable which stores the position of first element of LIST. In this figure, *start* has value 5 i.e. it points the 5$^{th}$ location of Array INFO of linked list. In INFO[5] there is element B, and LINK[5] stores 2 i.e. the next element is stored in 2$^{nd}$ position of INFO array. If LINK[k] stores 0 this means INFO[k] is the last element of the LIST.
- There are some **disadvantages of Array representation** of Linked List.
  - It is relatively expensive to insert and delete elements in an array.
  - Since an array occupies a block of memory space, one cannot simply double or triple the size of array when additional space is required. (For this reason, arrays are called *dense lists* and are said to be *static* data structures.

```
                start=n;
                printf("\n%d has been inserted",item);
        }
        else
        {
                p=start;
                t=start->next;
                while(t!=NULL)
                {
                        if(t->data==key)
                        {
                                n->next=t;
                                p->next=n;
                                printf("\n%d has been inserted",item);
                                break;
                        }
                        p=t;
                        t=t->next;
                }
                if(t==NULL)
                        printf("\n%d is not present in the list",key);
        }
}

void Reverse()
{
        node *p, *q, *t;
        p=NULL;
        t=start;
        while(t!=NULL)
        {
                q=p;
                p=t;
                t=t->next;
                p->next=q;
        }
        start=p;
        printf("\nList has been reversed");
}
```



**9.1.7 Write a C function for inserting nodes in singly linked list in increasing order.**

```
void SortedList(int item)
{
        node *n, *t, *p;
        n=(node*)malloc(sizeof(node));
        n->data=item;
        n->next=NULL;

        if(start==NULL)
```

4. N→Data= ITEM
5. N→Next= NULL
6. If(START==NULL)
    a. START= N
    b. Return
7. START→Prev= N
8. N→Next= START
9. START= N
10. Return

**INSERT_LAST(LIST, START, ITEM)**

Where LIST is a doubly linked list, START is pointer variable which stores the address of first element, and ITEM is an element to be inserted.

1. Make a new node N
2. If (N==NULL) then
    a. Write(Memory is not available)
    b. Return
3. N→Prev= NULL
4. N→Data= ITEM
5. N→Next= NULL
6. If (START== NULL)
    a. START= N
    b. Return
7. T= START
8. while(T→Next != NULL) do
    a. T= T→Next
9. T→ Next= N
10. N→ Prev= T
11. Return



**INSERT_AFTER(LIST, START, ITEM, KEY)**

Where LIST is a doubly linked list, START is pointer variable which stores the address of first element, ITEM is an element to be inserted, KEY is an element after which new element ITEM will be inserted.

1. Make a new node N
2. If (N= = NULL) then
    a. Write (Memory is not available)
    b. Return
3. N→Prev= NULL
4. N→Data= ITEM
5. N→Next= NULL
6. T= START
7. while (T!=NULL) do
    a. if (T→Data = = KEY) then
        i. N→Prev=T
        ii. N→Next= T→Next
        iii. T→Next→Prev= N
        iv. T→Next= N

```
stack *top;

void Push(int);
void Pop();
void Traverse();

void main()
{
        int choice, item;
        clrscr();

        do
        {
                printf("\n\nStack");
                printf("\n1.Push\n2.Pop\n3.Traverse\n4.Exit\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("\nEnter one number ");
                                scanf("%d",&item);
                                Push(item);
                                break;
                        case 2: Pop();
                                break;
                        case 3: Traverse();
                }
        }while(choice!=4);
}

void Push(int item)
{
        stack *n;
        n=(stack*)malloc(sizeof(stack));
        n->data=item;
        n->next=top;
        top=n;
        printf("\n%d has been pushed",item);
}

void Pop()
{
        stack *t;
        if(top==NULL)
        {
                printf("\nStack is empty");
        }
        else
        {
                t=top;
```

```
                start=n;
        }
        else if(start->next==NULL)
        {
                start->next=n;
        }
        else
        {
                p=start;
                t=start->next;
                while(t!=NULL)
                {
                        if(t->pr > priority)
                        {
                                n->next=t;
                                p->next=n;
                                break;
                        }
                        p=t;
                        t=t->next;
                }
                if(t==NULL)
                        p->next=n;
        }
        printf("\n%d has been inserted",item);
}

void QDelete()
{
        node *t;
        t=start;
        if(t==NULL)
                printf("\nPriority Queue is Empty");
        else
        {
                start=start->next;
                printf("\n%d has been deleted",t->data);
                free(t);
        }
}

void QTraverse()
{
        node *t;
        t=start;
        if(t==NULL)
                printf("\nPriority Queue is empty");
        else
        {
                printf("\nPriority Queue is\n");
```

```
                start=n;
        }
        else
        {
                t=start;
                while(t->next!=NULL)
                {
                        t=t->next;
                }
                t->next=n;
        }
        printf("\nTerm is inserted");
    }
}

void Add()
{
    node *n, *t, *p, *q;

    t=p1;
    p=p2;

    while(t!=NULL || p!=NULL)
    {
        n=(node*)malloc(sizeof(node));
        n->next=NULL;
        if(t!=NULL && p!=NULL && t->expo==p->expo)
        {
                n->coeff=t->coeff+ p->coeff;
                n->expo=t->expo;
                t=t->next;
                p=p->next;
        }
        else if(p==NULL || t->expo> p->expo)
        {
                n->coeff= t->coeff;
                n->expo= t->expo;
                t=t->next;
        }
        else if(t==NULL || t->expo < p->expo)
        {
                n->coeff= p->coeff;
                n->expo= p->expo;
                p=p->next;
        }
        if(start==NULL)
                start=n;
        else
        {
                q=start;
```