UNIT-II

LINKED LIST

INTRODUCTION

An Array is a very useful data structure provided in programming languages. It has some limitations such as

(a) Memory storage space is wasted, as the memory remains allocated to the array throughout the program execution even if small elements are stored.

(b) The size of array can't be changed after its declaration i.e its size has to be known at compilation time.

(c) The addition (insertion) and deletion of elements requires shifting other data in this array.

These limitations of array can be overcome by using Linked list data structure.

* What is Linked List : (Definition)

A linked list is a set of nodes where each node has two fields "data" and "link". The "data" field stores actual piece of information and "link" field is used to point to next node. Basically the link field is nothing but the address only.

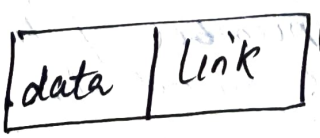


fig: Structure of node.

Linked list is also known as Single Linked list. It is a list which uses non-contiguous memory. Logically it is linear in structure.

* **Structure of linked list**

```
struct node
{
  int info;
  struct node * link;
};
struct node * first;
```

* **Advantages of Linked list:**

1) **Dynamic data structure:** Linked list are dynamic data structure. i.e they can grow or shrink during the execution of a program.

2) **Efficient memory utilization:** Memory is not pre-allocated. Memory is allocated whenever it is required and it is deallocated when it is no longer needed.

3) **Insertion & deletion are easier & efficient:** Linked lists provide flexibility in inserting a data item at a specified position and deletion of data item from the given position.

(4) Many complex applications can be easily carried out with linked list.

* **Disadvantages of Linked list**

(1) **More Memory:** If the number of fields are more than more memory space is needed.

(2) Access to an arbitrary data item is little bit cumbersome and also time consuming.

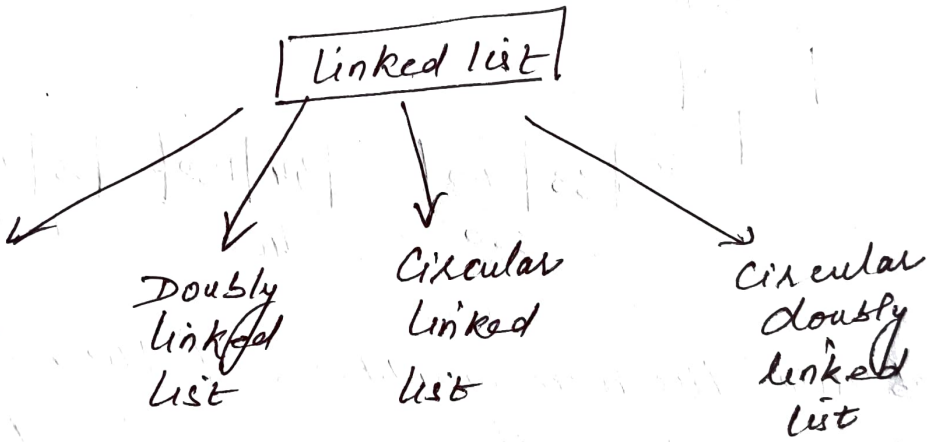
LINKED LISTS

List is a collection of similar type of elements. There are two ways of maintaining a list in memory.

① The first way is to store the elements of the list in an array. But arrays have some restrictions and disadvantages.

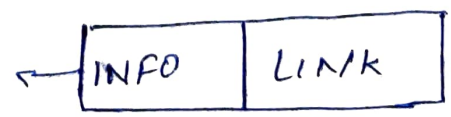
② The second way of maintaining a list in memory is through linked list.

Types of Linked list



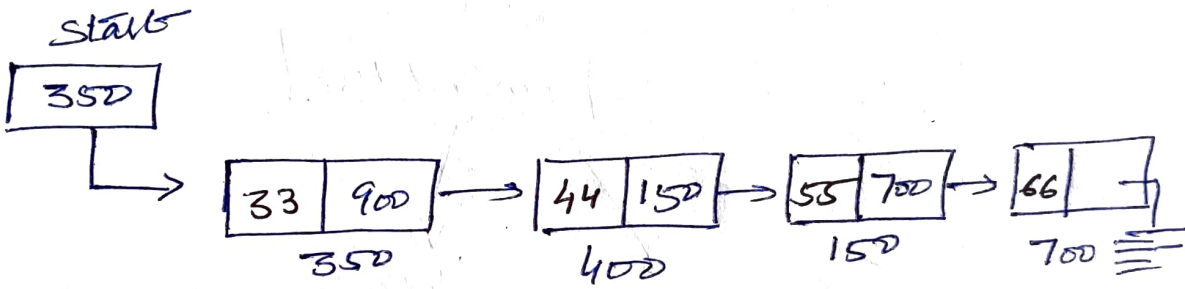
① Single linked list

A single linked list is made up of nodes where each node has two parts, the first one is the info part that contains the actual data of the list and the second one is the link part that points to the next node of the list as it contains the address of the next node.



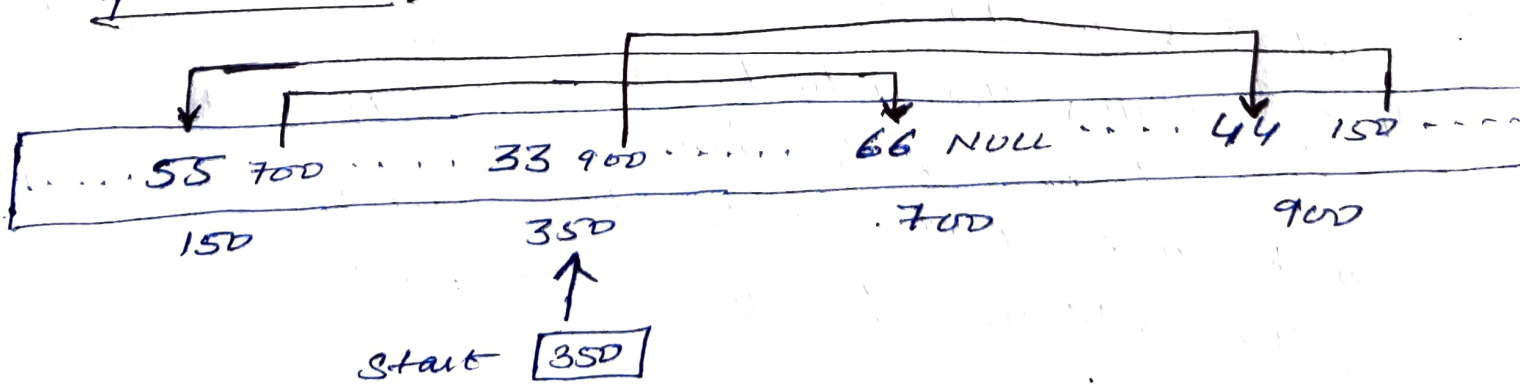
- The beginning of the list is marked by a special pointer named start. This pointer points to the first node of the list.
- The link part of the each node points to the next node in the list, but the link part of last node has no next node to point to, so it is made NULL.

eg Suppose, we have a list of four integers 33, 44, 55, 66. how we will represent it through linked list.



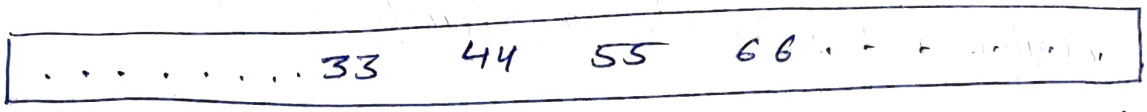
If we observe the memory address of the nodes, we find that the nodes are not necessarily located adjacent to each other in the memory.

The following figure shows the position of nodes of the above list in memory.



- The nodes are scattered here and there in memory. but still they are connected to each other through the link part, which also maintains their linear order.

The picture of memory if the same list of integers is implemented though array.



The elements are stored in consecutive memory locations in the same order as they appear in the list.

Array	Linked list
• Array is sequential representation of list.	• Linked list is the linked representation of list.
• The elements are stored in consecutive memory locations in the same order as they appear in the list.	• In a linked list, nodes are not stored contiguously. but they are linked through pointers (links).

How we can represent a node of a single linked list in C language.

The nodes in linked list is represented by self-referential structure.

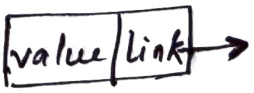
- A self referential structure is a structure which contains a pointer to a structure of the same type.

The general form of a node of linked list is —

```
struct node
{
    type1 member1;
    type2 member2;
    ....
    ....
    struct node *link;   // pointer to next node.
};
```

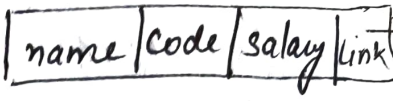
Examples

(1)
```
struct node {
    int value;
    struct node *link;
};
```



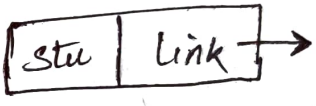
(2). struct node {

```
char name[10];
int code;
float salary;
struct node *link;
};
```



(3) struct node {

```
struct student stu;
struct node *link;
};
```



In array we could perform all the operations using the array name and index.

In the case of linked list, we will perform all the operations with help of the pointer start because it is the only source through which we can access our linked list.

The list will be considered empty if the pointer start contain NULL value.

```
struct node *start;
start = NULL;
```

Operations on a Single linked list

(1) Traversal of a linked list
(2) Searching an element
(3) Insertion of an element.
(4) Deletion of an element.
(5) Creation of a linked list
(6) Reversal of a linked list.

Traversal a Single Linked List

Traversal means visiting each node, starting from the first node till we reach the last node..

p = start;

- we will take a structure pointer p which will point to the node that is currently being visited.
- we have to visit the first node so p is assigned the value of start.
- By writing p = start it means, now p points to the first node of linked list.
- p → info with this statement we can access the info part of first node.

- To shift the pointer p forward so that it points to the next node. This can be done by assigning the address of the next node to p as -

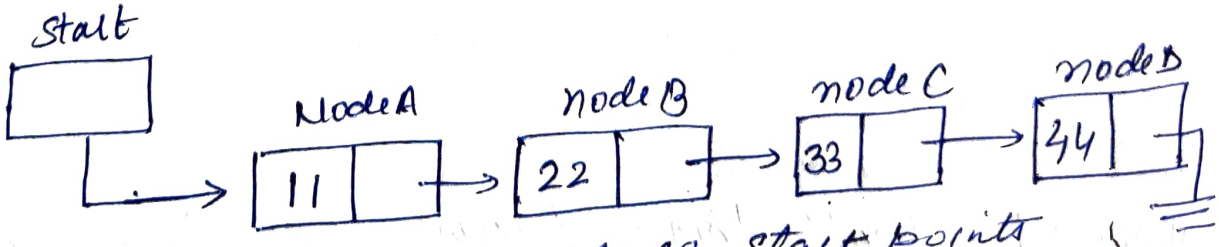
`p = p -> link` or `p = p -> next`

By writing this statement, now p has address of the next node.

Similarly we can visit each node of linked list through this assignment until p has NULL value. which is link part value of last element.

So, the linked list can be traversed as -

```
while (p != NULL)
{
    printf ("%d", p->info);
    p = p -> next;
}
```



- node A is the first node so start points to it, node D is the last node so its link is NULL.

Algorithms of Linked List

1. Single linked list

To insert an element into the following three things should be done.

(1) Allocating a node.
(2) Assigning the data
(3) Adjusting the pointers

Inserting a new node into the linked list has the following three instances:

a) Insertion at the beginning of the list.
b) Insertion at the end of the list.
c) Insertion at the specified position within the list.

Insertion in a single linked list

There can be four cases while inserting a node in a linked list.

a) Insertion at the beginning.
b) Insertion in an empty list.
c) Insertion at the end.
d) Insertion in between the List nodes.

To insert a node, initially we will dynamically allocate space for that node using malloc(). Suppose tmp is a pointer that points to this dynamically allocated node. In the info part of the node we will put the data value.

```
tmp = (struct node *) malloc(sizeof(
                              struct node));

tmp->info = data;
```

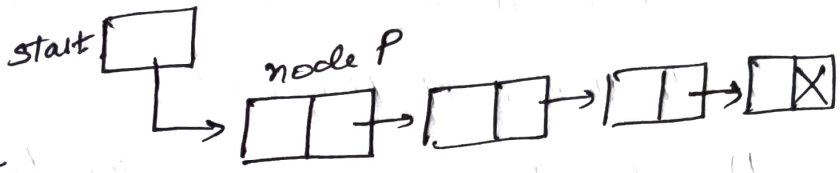
The link part of the node contains garbage value. we will assign address to it separately in the different cases.

(a) Insertion at the beginning of the list

We have to insert node T at the beginning of the list. Suppose the first node of the list is node P. So the new node T would be inserted before it.

Before insertion

Node P is the first node start points to node P



After insertion

Node T is first node
Node P is second node
start points to node T
Link of node T points to node P

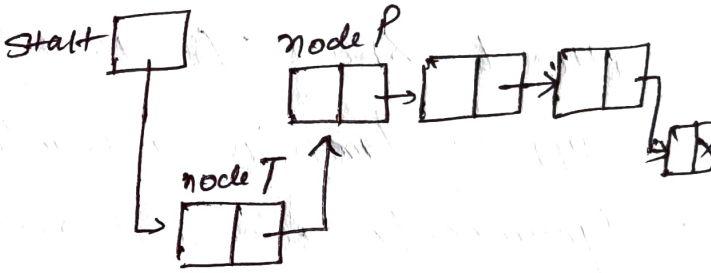


Fig: Insertion at the beginning of the list

(1) Link of node T should contain the address of node P and we know that start has address of node P.

```
tmp -> next = start;
```

(2) To make node T the first node, hence we should update start so that now it points to node T.

```
start = tmp;
```

Note : What happens if the order of these two statements is reversed

```
start = tmp;
```

start points to tmp and we lost the address of node P.

```
tmp -> next = start;
```

Link of tmp will points to itself because start has address of tmp, and we will stuck in an infinite loop when the list is processed.

a) Inserting a Node at the Beginning :

Insert first (start, item)

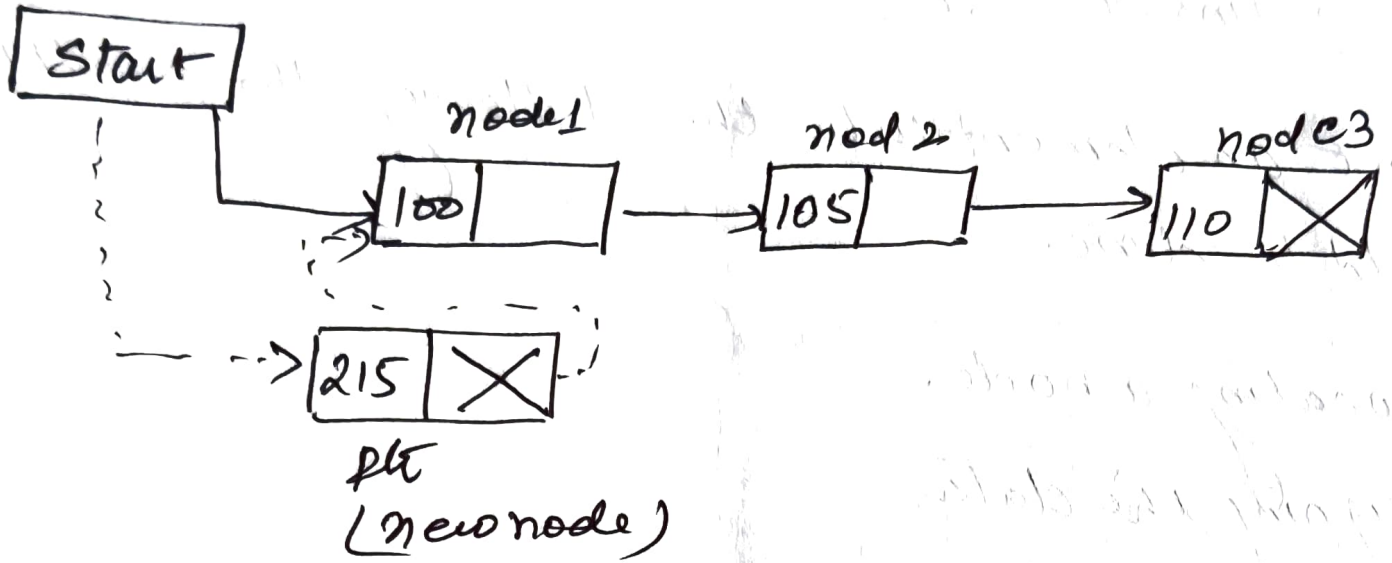
(1) [Check for overflow?]

```
if ptr = NULL then
    print "Overflow"
    exit
else
    ptr = (NODE*) malloc (size of (node));
endif
```

(2) Set Ptr → info = item

(3) Set Ptr → next = Start

(4) Set Start = Ptr



After insertion