Linked List

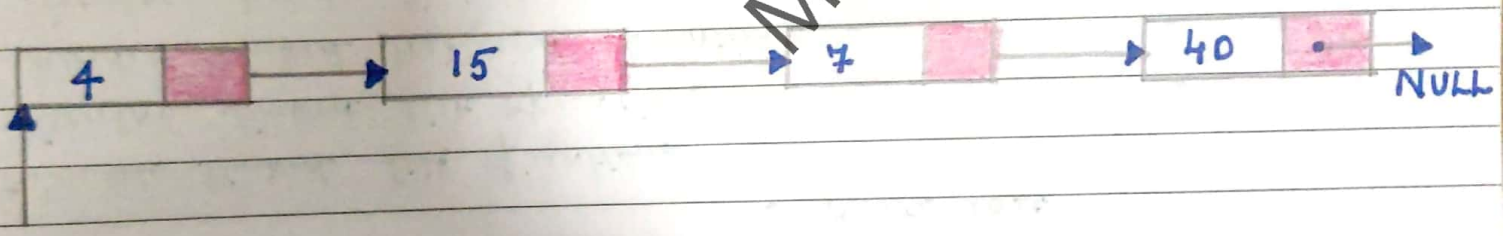
Topics

1. Introduction to linked list
2. Advantages, disadvantages, applications.
3. Type of linked list representation
4. Single linked list operation :- a) traversal
   b) insertion
   c) deletion
   a) insert last
   e) delete last
5. Double linked list operations
6. Circular linked list operations
7. Header linked list and operations
8. Circular header linked list and operations
9. Polynomial
10. Double linked list and operations
11. Sparse matrix.

① What is a linked list?

Ans a linked list is a data structure used for storing collections of data. A linked list has the following properties.

a) Successive elements are connected by pointers.
b) The last element points to NULL.
c) Can grow or shrink in size during execution of a program.
d) Can be made just as long as required (until systems memory exhausts)
e) Does not waste memory space (but takes some extra memory for pointers.) It allocates memory as list grows.



What is list and linked list?

Ans list	linked list
① a sequence of data structures, which are connected together via links.	① a data linear structure, which the elements are not stored at contiguous memory locations.
② Uses a dynamic array to store the elements	② Uses a double linked list to store the elements

③ list class can act as a list only	③ linked list class can act as list and queue both.
④ list is better for storing and accessing data	④ linked list is better for manipulating data.

Linked List ADT :-

The following operations make linked lists an ADT :-

Main linked lists operations

a) Insert :- inserts an element into the list
b) Delete :- removes and returns the specified position element from the list.

Auxiliary linked lists operations

a) Delete list : removes all elements of the list (disposes the list)

b) Count :- returns the number of elements in the list

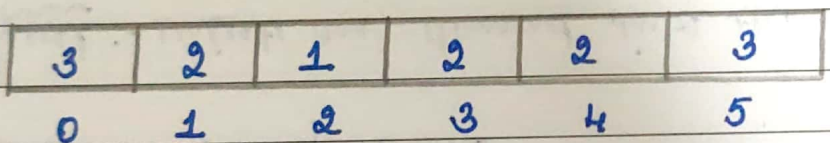
c) find n^{th} node from the end of the list.

Why linked lists?

Ans There are many other data structures that do the same thing as linked lists. Before discussing linked list it is important to understand the difference between linked lists and arrays. Both linked list and arrays are used to store collections of data, and since both are used for the same purpose, we need to differentiate their usage. That means in which cases arrays are suitable and in which cases linked lists are suitable.

Arrays overview :-

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.



Why constant time for accessing array elements?

Ans To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

two operations take constant time, we can say the array access can be performed in constant time.

advantages of arrays :-

a) Simple and easy to use.
b) Faster access to the elements (constant access)

disadvantage of arrays :-

a) Preallocate all needed memory up front and wastes memory space for indices in the array that are empty.

b) Fixed Size :- the size of the array is static (specify the array size before using it.)

b) One block allocation :- to allocate the array itself at the beginning sometimes it may not be possible to get the memory for the complete array (if array size is big)

c) Complex position-based insertion :- to insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

Dynamic Arrays :-

also called as growable array, resizable array, dynamic table or array list. is a random access, variable size list data structure that allows elements to be added or removed.

One simple way of implementing dynamic arrays is to initially start with some fixed size array. As soon as that array becomes full, create the new array double the size of the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

Note :- We will see the implimentation for dynamic arrays in the Stacks, Queues and hashing.

Advantages of linked lists

linked lists have both advantages and disadvantages. The advantage of linked list is that they can be expanded in constant time. To create an array, we must allocate memory for a certain number of elements. To add more elements to the array when full, we must create a new array and copy the old array into the new array. This can take a lot of time.

We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element and add on new elements easily without the need to do any copying and reallocating.

Issues with linked lists (disadvantages) :-

There are a number of issues with linked lists. The main disadvantage of linked lists is access time to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists take O(n) for access to an element in the list in the worst case.

Another advantage of arrays in access time is spacial locality in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the overhead with storing and retrieving data can make a big difference. Sometimes linked lists are hard to manipulate. If the last item is deleted, the last but one must then have its pointer changed to hold a 'NULL' reference. This requires that the list is traversed to find the last but one link, and its pointer set to a 'Null' reference.

Finally, linked lists waste memory in terms of extra reference points.

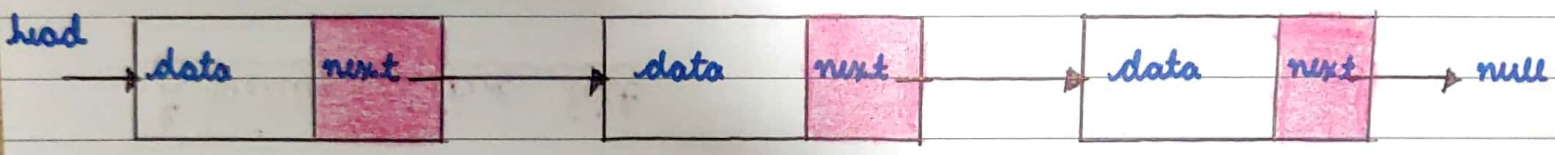
Comparison of linked list with arrays and dynamic arrays

Paramter	linked list	Array	Dynamic array
Indexing	O(n)	O(1)	O(1)
Inserting / deletion at beginning	O(1)	O(n), if array is not full (for the shifting the elements)	O(n)
insertion at ending	O(n)	O(1) if array is not full.	O(1), if array is not full. O(n) if array is full.
deletion at ending	O(n)	O(1)	O(n)
insertion in middle	O(n)	O(n), if array is not full (for shifting the elements)	O(n)
deletion in middle	O(n)	O(n), if array is not full (for shifting the elements)	O(n)
wasted space	O(n) (for pointers)	0	O(n)

Types of linked list

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list :-



1. Each node has data and a pointer to the next node.

Node represented as :-

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

A three member singly L.L can be created as :-

```
struct node * head;
struct node * one = NULL;
struct node * two = NULL;
struct node * three = NULL;
```

} initialize nodes

```
One = malloc (sizeof (struct node));
two = malloc (sizeof (struct node));
three = malloc (sizeof (struct node));
```
} allocate memory

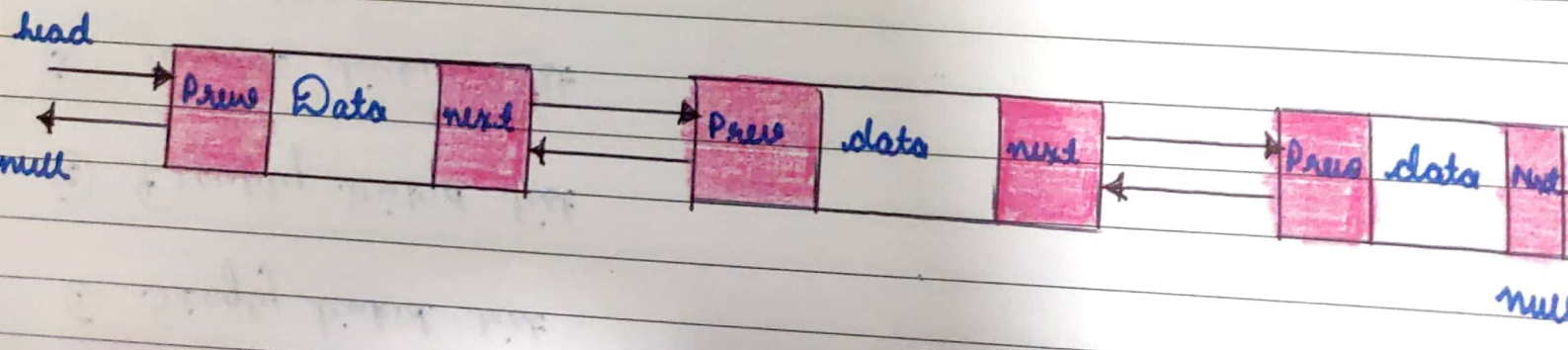
```
one → data = 1;
two → data = 2;
three → data = 3;
```
} assign data values

```
one → next = two;
two → next = three;
three → next = Null;
```
} connect nodes

```
head = one;
```
} save address of first node in head.

Doubly linked list

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction :- forward or backward.



node representation :-

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

A three member doubly linked list can be created as :-

```
struct node * head;
struct node * one = NULL;
struct node * two = NULL;
struct node * three = NULL;
```
} Initialize nodes.

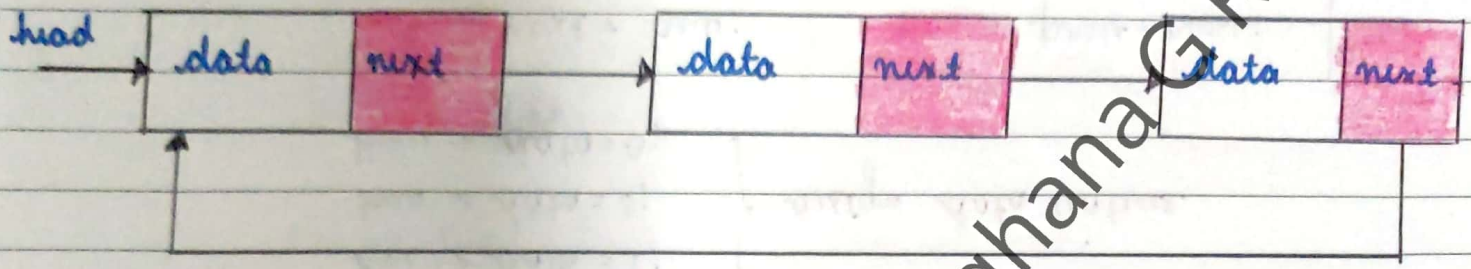
```
one = malloc (sizeof (struct node));
two = malloc (sizeof (struct node));
three = malloc (sizeof (struct node));
```
} allocate memory

```
one -> data = 1;
two -> data = 2;
three -> data = 3;
```
} assign data values.

```
one -> next = two;        one -> prev = NULL;
two -> next = three       two -> prev = one;
three -> next = Null      three -> prev = two;
```
} connect nodes.

```
head = one;
```
} same address of first node in head.

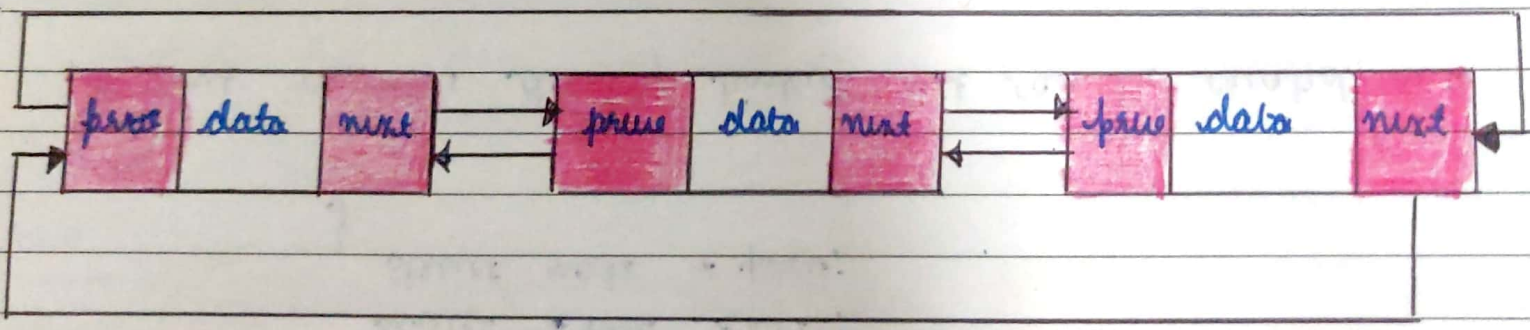
Circular linked list :- is a variation of L.L in which the last element is linked to the first element. This forms a circular loop.



A circular linked list can be either singly linked or doubly linked.

a) for singly linked list, next pointer of last item points to the first item.

b) in doubly linked list, prev pointer of first item point to last item as well.



A three member circular singly linked list can be created as :-

```
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```
} initialize nodes

```
one = malloc (sizeof (struct node));
two = malloc (sizeof (struct node));
three = malloc (sizeof (struct node));
```
} allocate memory

```
one -> data = 1;
two -> data = 2;
three -> data = 3;
```
} assign data values.

```
one -> next = two;
two -> next = three;
three -> next = one;
```
} connect nodes.

```
head = one;
```
} save address of first node in head.

What are lists?

Ans Fundamental data structure in computer Science.

LISP (list processing) was built upon lists as the only data structure in the language.

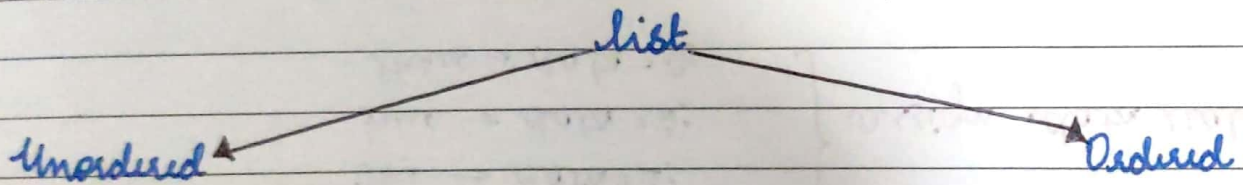
Eg of list are :-

integer list = { 0, 1, -1, 2, -2, 3, -3 ..... }

days of week = { S, M, T, W, Th, F, Sa }

instances may or may not be related

{ apple, chair, 2, 5.2, red, green, jack }

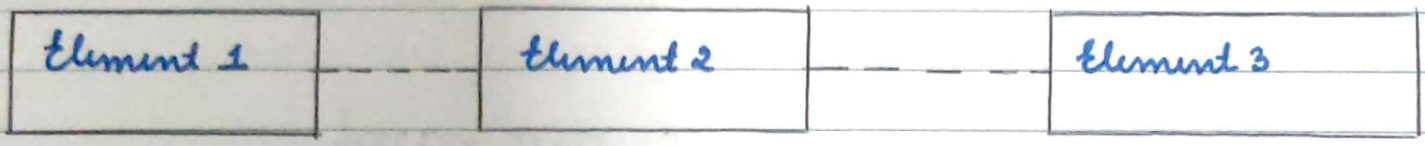


lists are lists in which the items of the list are not in any particular order.

lists are very similar to the alphabetical list of employee names for the XYZ company.

Since order has to be maintained whenever an item is added to the list, it is placed in the correct sorted position so that the entire list is always sorted

Linear List



① A sequence of elements
② There is first and last element
③ Each element has previous and next

→ nothing before first
→ nothing after last.

instances are of the form $(e_0, e_1, e_2 \ldots e_{n-1})$
where $n \geq 0$ (finite)
list size → n

relationships :- e_0 is the zeroth (or front) element
e_{n-1} is the last element
e_i immediately precedes e_{i+1}

Example:- Students in comp 2210 = (Jack, Jill ... Judy)

List Operations

① Create an empty list
② Determine whether a list is empty.
③ Get the item at a given position in a list (retrieve)
④ Determine the number of items on a list
⑤ Determine the index of an element.

① create (list)

description :- create new, empty list
input :- type of elements in list.
output :- none
result :- a new empty list is created.
we need to specify the size, type.

② isEmpty()

description :- checks to see if a list is empty
Input :- none.
Output :- true if the list is empty
false if the list is not empty } boolean output.
result :- the list is unchanged.

③ GetIndex ※

get (Index) → get element with given index.

L = (a, b, c, d, e)
→ get(0) = a , get (2) = c , get(4) = e
get (-1) = error
get(9) = error

discription :- get an item from list
Input :- index : the item to get
Output :- returns the item at the specified index
throws an exception if index is out of range.

result : the list is unchanged.

④ retrieval operation :- Specify the index and retrieve the value.

Linear List Operations

(size, index) :-

① traverse :- get-first (list) → returns first element if it exists.

get-next (list) → returns next element if it exists

both functions return NULL otherwise.

Calling get-next in a loop we will get one by one all elements of the list.

Insert :- ① add (theIndex, the element) - add an element so that the new element has a specified index.
② add (L, item) : add an element item anywhere in a list

Function : adds item to list

Precondition :- a) List has been initialized,
b) list is not full
c) item is not in list (to avoid duplication)

Post condition :- item is in list.

Ex:- L = (a, b, c, d, e, f, g)

add (0, h) ⇒ L (h, a, b, c, d, e, f, g)
index of a, b, c, d, e, f & g increased by 1.

Delete Operation

Delete (the index) :- remove and return element with given index.

Delete (L, item) : remove an element item anywhere in a list L.

Function :- Delete the element whose key matches item's key.

Precondition :- a) list has been initialized,
b) key member of item has been initialized
c) there is only one element in list which has a key matching item's key.

Postcondition :- No element in list has a key matching item's key.

L = (a, b, c, d, e, f, g)
remove (2) return c

L becomes (a, b, d, e, f, g)
index of d, e, f and g decrease by 1.

RemoveAll() Operation

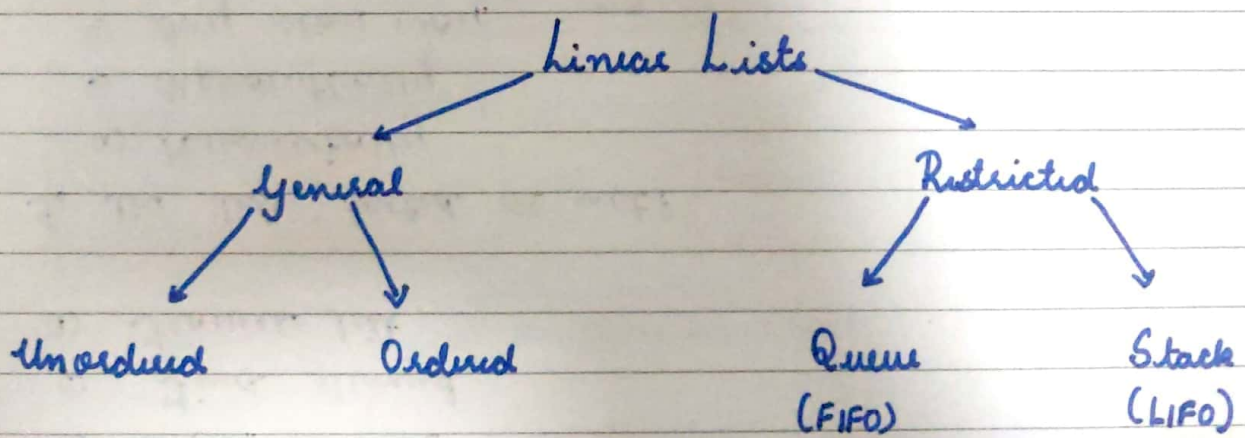
Description :- Removes all items from a list

Input :- None

Output :- None

Result :- the list is now empty.

Classification of Lists



What makes a special kind of list?

What we can do with a linear list?

a) Delete element
b) Insert element
c) Find element
d) Traverse list

Is the list sorted or not?

a) Numerically
b) Alphabetically
c) Any other way.

Operations that we want to do, depends on the (list) we choose.
↳(application)

How we can implement a list?
Ans Array, linked list, doubly linked list

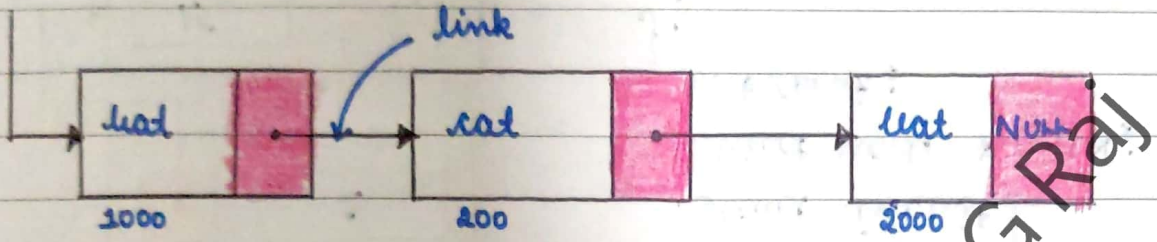
How we can implement a list?

Lists vs Linked list

→ A list is just (a list of names, phone numbers - nothing to do with implementation) a list can be implemented using different data structures (arrays, linked list)

→ Linked-List data structure can be used to implement many data structures (eg:- stacks, queues, lists, bits)

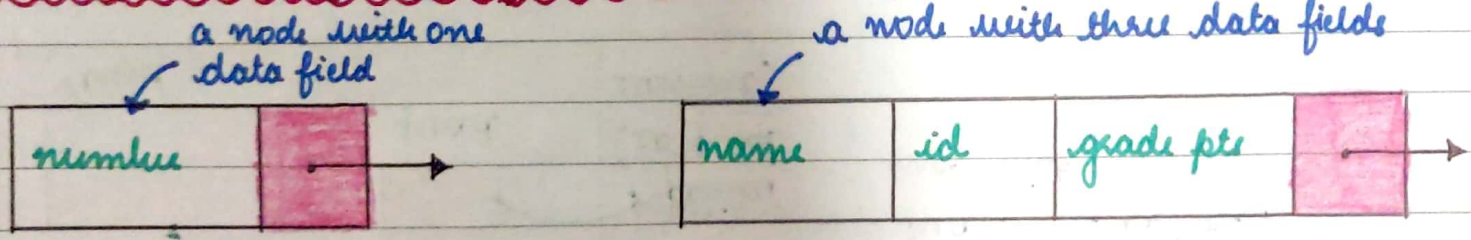
Linked List Introduction

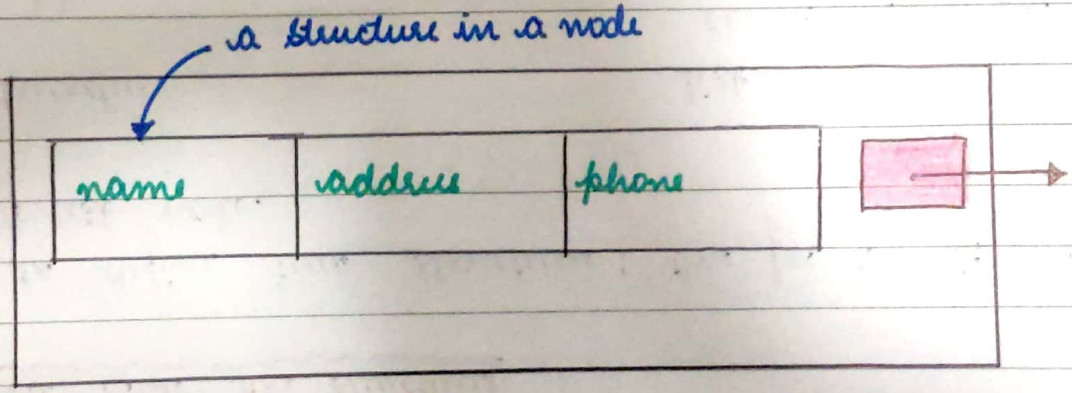


→ these are lists connected logically to each other, physically they are not next to each other.

→ Every linked list will have 'head' which is the first element (node), if list is empty, then head would be NULL.

A node - an element of the linked list

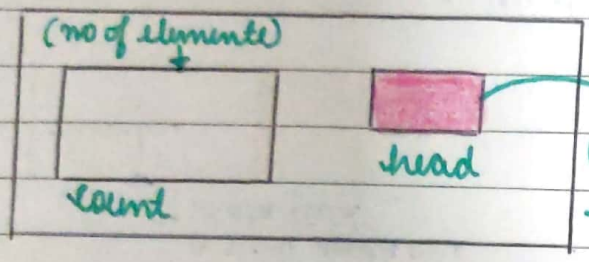




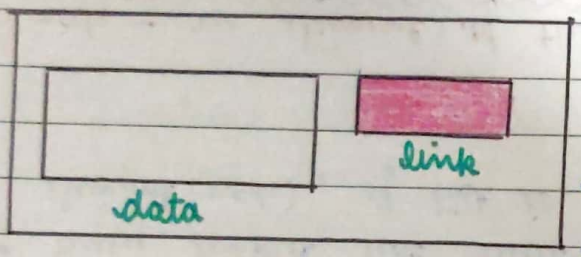
Head and Node are Structures

We need to define two structures: one for a list head, one for a list node.

head Structure



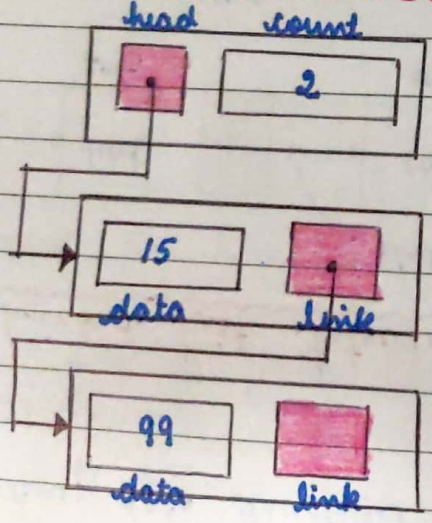

```
list
    count   <integer>
    head    <pointer>
end list
```



(can be anything even a structure)

```
node
    data    <data-type>
    link    <pointer>
end node.
```

Head and Node: Simple int list




```
struct intlist {
    struct node *head;
    int count;
};

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

Head and Node : Structure as data

```
struct intlist {
    struct node *head;
    int count;
};

struct node {
    struct person *data;
    struct node *link};

struct person {
    char *name;
    int age;
}
```

Implimentation (allocating memory)

Declaration :-

```
typedef struct node, *pnode;
typedef struct node {
        char data[4];
        pnode next; };
```

Creation :-

```
pnode ptr = NULL;
```

testing

```
#define IS_EMPTY(ptr) (!(ptr))
```

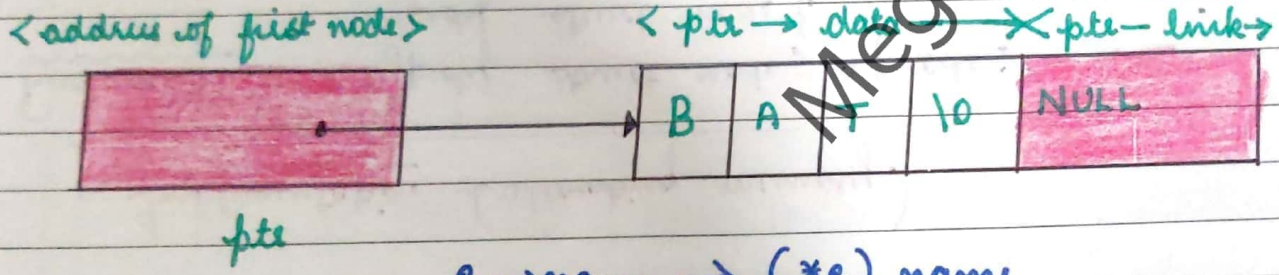
Allocation

```
ptr = (pnode) malloc (sizeof (node));
```

Main Operations

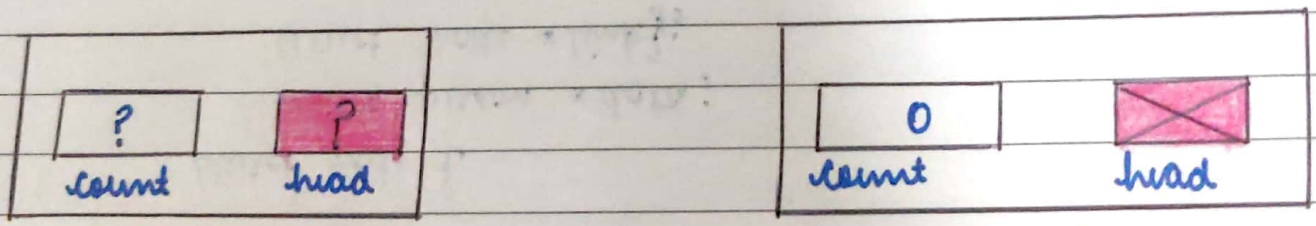
1. Create list
2. Add node (beginning, middle or end)
3. Delete node (beginning, middle or end)
4. Find node
5. Traverse list

Create one Node



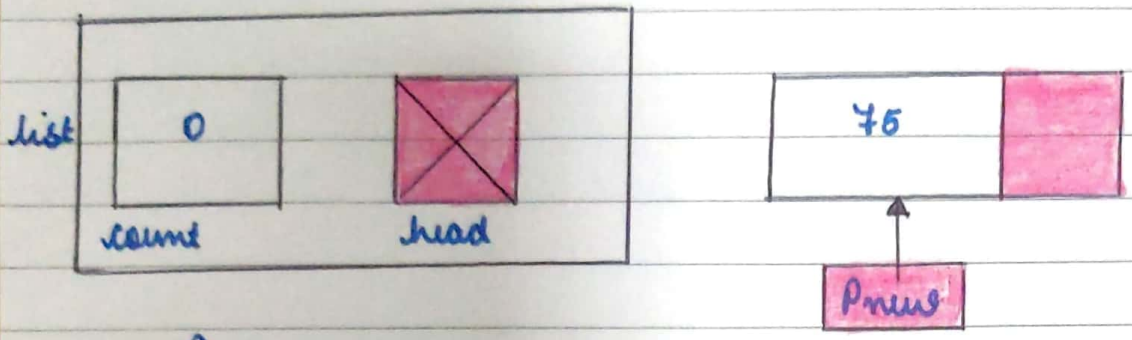

```
e->name  ⇒  (*e).name
Strcpy (ptr->data, "Cat");
ptr->link = NULL;
```

Create list



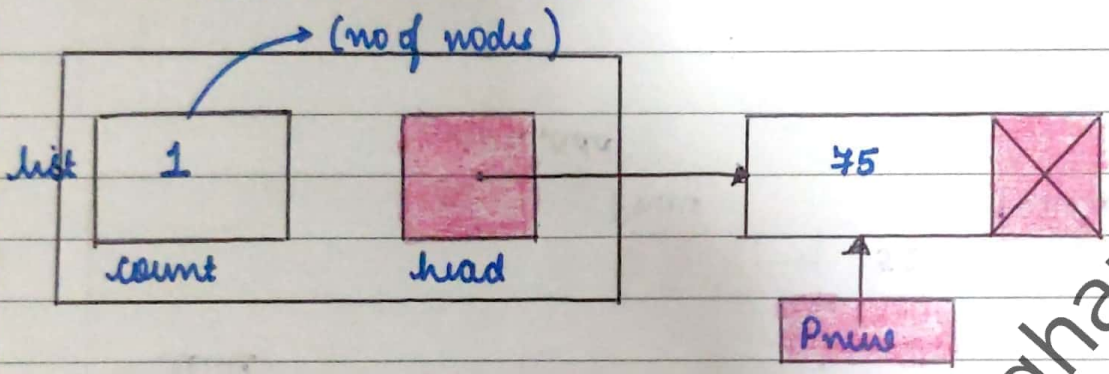

```
list.head = NULL
list.count = 0
```

Add: Empty list



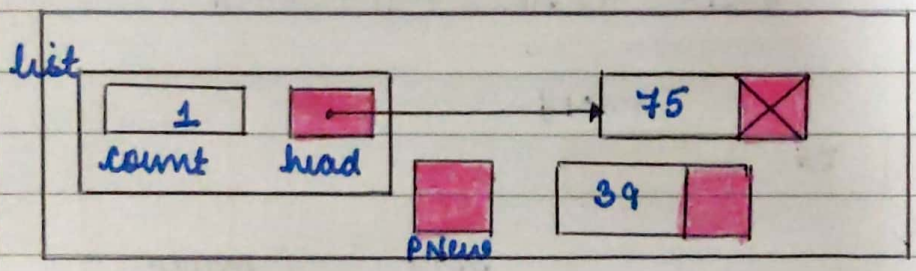
PNew → link = list.head

list.head = PNew



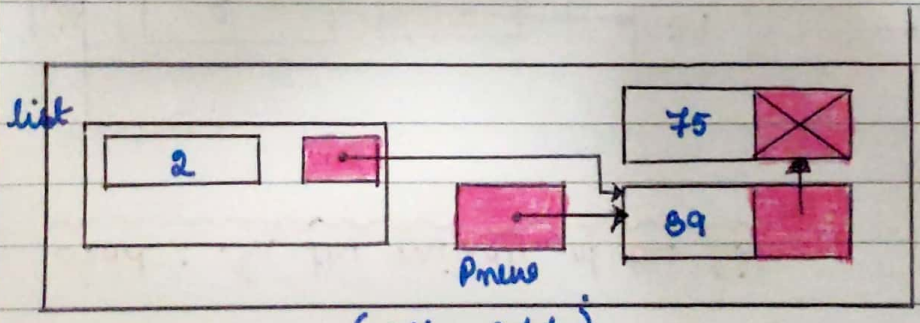
after add

Add: At the begining of the list



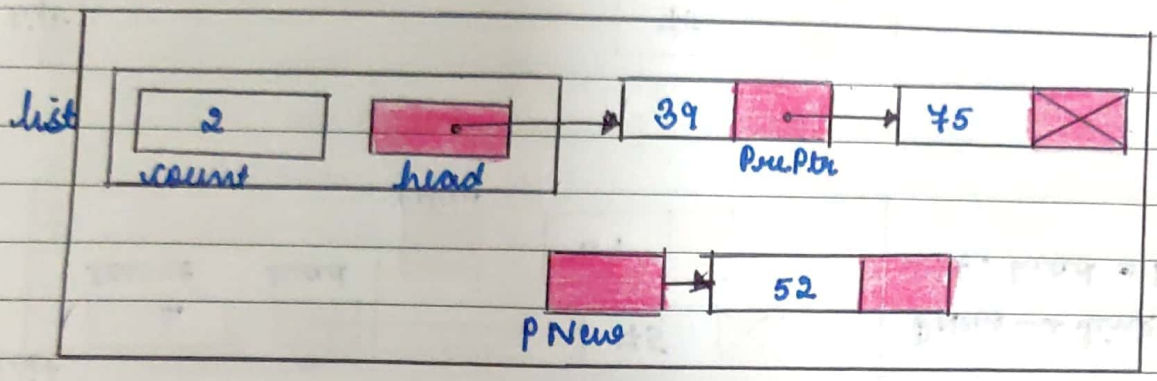
PNew → link = link.head

list.head = PNew.



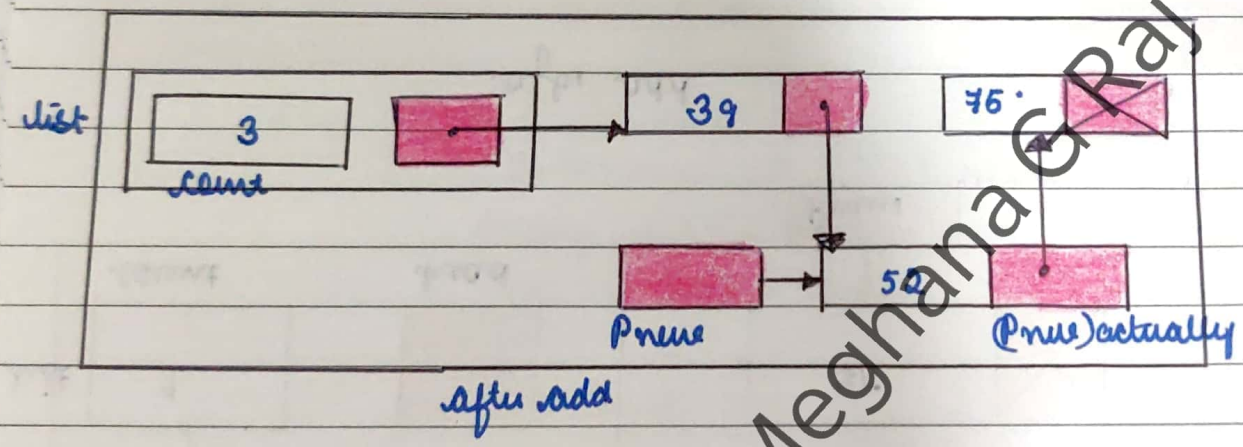
(after add)

Add : In the middle of the list



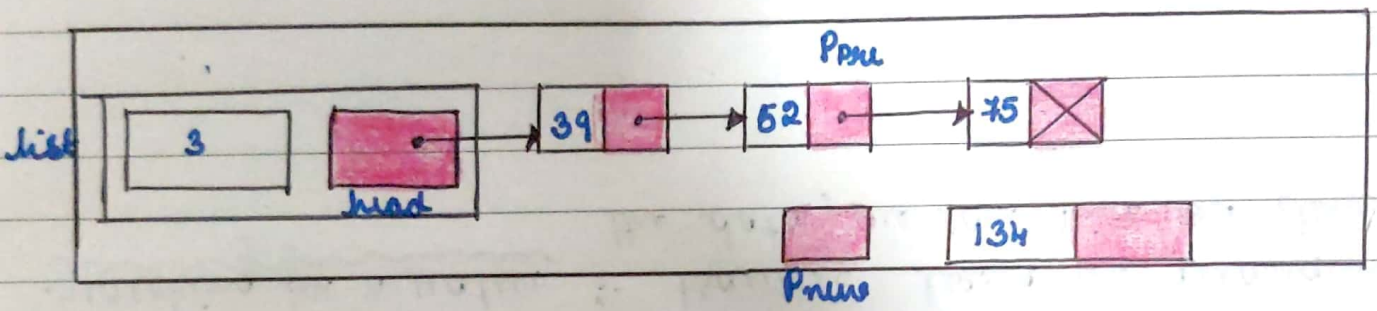
Pnew → link = PrePtr → Link

PrePtr → link = Pnew.



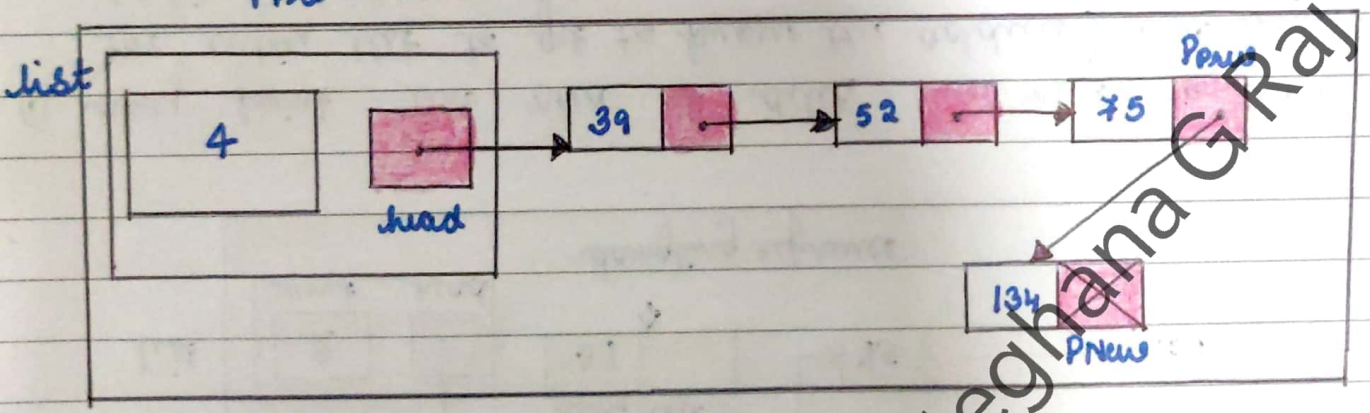
Add: At the end of the list

* this is the disadvantage, we have (assume) 1 lakh nodes, all nodes should be surfed and arrive at the end, collect the addres of the previous node address.



PNew → link = PRe → link

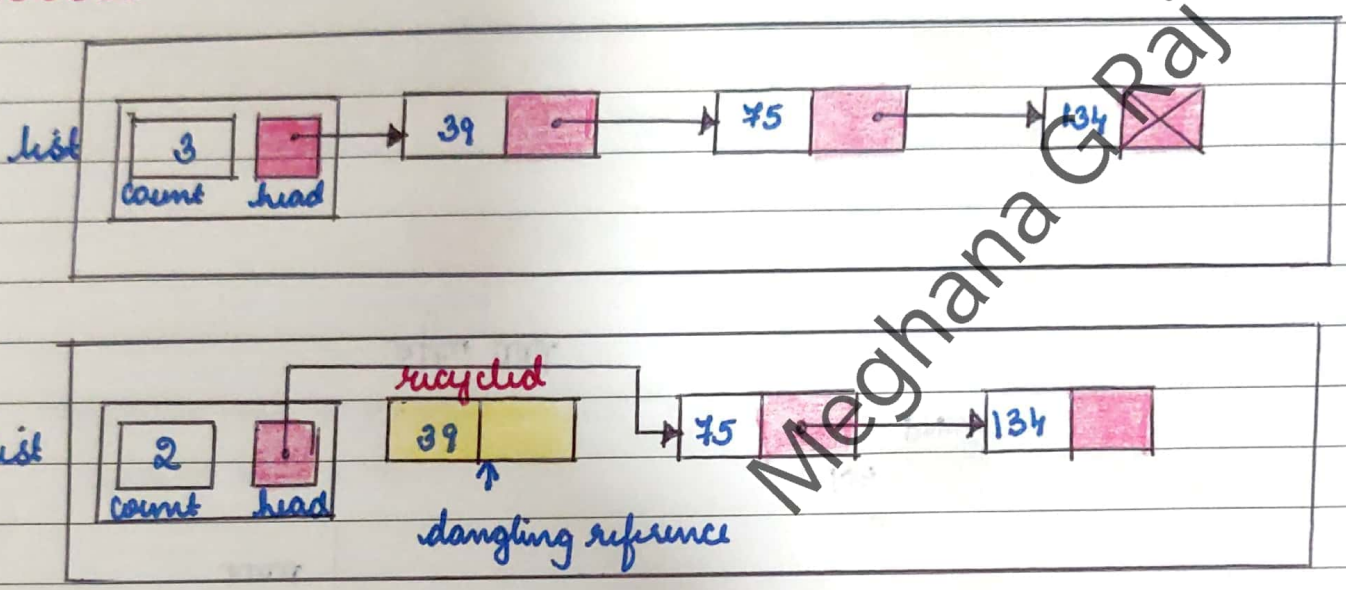
PPre → link = PNew



after add

Deletion

Delete First :-



① any point we need to delete (we need to traverse the entire list to get to know the address) and then go for deletion operation.

Searching for a value :- traverse based on arrangement of the data, we can find the element.

Single Linked List Operation

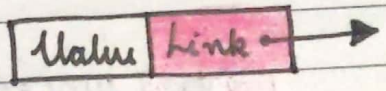
Note:- Array is sequential representation of list while linked list is the linked representation of list.

The general form of a node:-

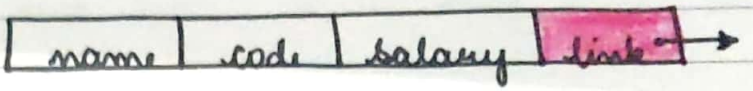
```
struct node
{
    type 1  member 1;
    ......
    ......
    struct node *link;  → pointer to next node.
};
```

Example:-

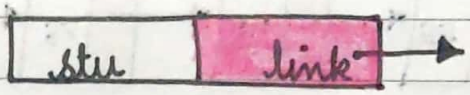
a)
```
struct node
{
    int value;
    struct node *link;
};
```



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



c)
```
 struct node
 {
  struct student stu;
  struct node * link;
 };
```



Following Operations we will discuss

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

Traversing a single linked list

Traversing → visiting each node.
(starting from first node till the last node)

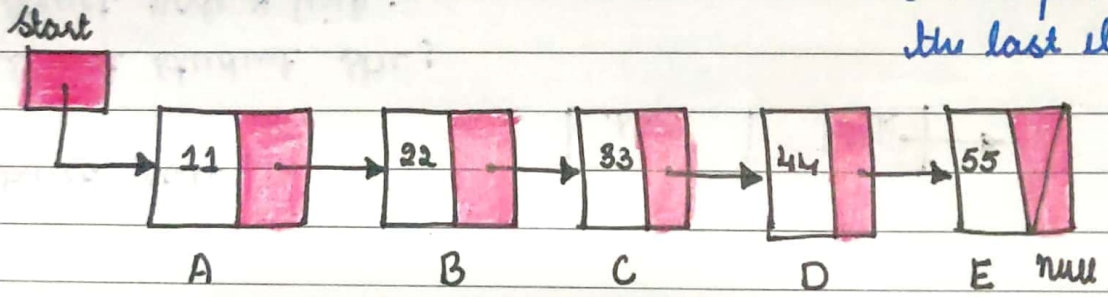
'P' → structure pointer
↳ point (node currently being visited)

first node visit so,

P = Start;
P → info
P = P → link (to next node movement)

} can be done only when P != NULL value
↓
(link part value of the last element)

Ex :-



(assuming P point A)

P → info = 11 P → link = B
P = P → link
P → info = 22 P → link = C

Note :- If we use start = start → link, instead of P = P → link then we will lose start and that is the only means of accessing the list.

to count the no of elements of the linked list

```
void count ( struct node * start)
{
  Struct node *p;
  int count = 0;
  p = start ;
  while ( p != null)
  {
    P = P → link;
    count ++;
  }
Pf (" No of elements are %d \n", count);
}
```

Searching in a single linked list

for searching an element, we traverse the linked list and while traversing we compare the info part of each element with the given element to be searched.

'item' – element we want to search.

```
void search ( struct node * start , int item )
  {
     Struct node *p = start;
     int pos = 1;
     while ( p != null)
     {
        if ( p -> info == item )   (to element to search)
        {
           Pf (" Item %d found at position %d \n",
                                    item , pos);
           return;
        }
           p = p -> link ;   } go to next element
           pos ++;
     }
     Pf (" Item %d not found in list \n", item);
  }
```


Insertion in a Single linked list

In case while inserting a node :-

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

a) to insert a node, initially we need dynamically allocate space using malloc().

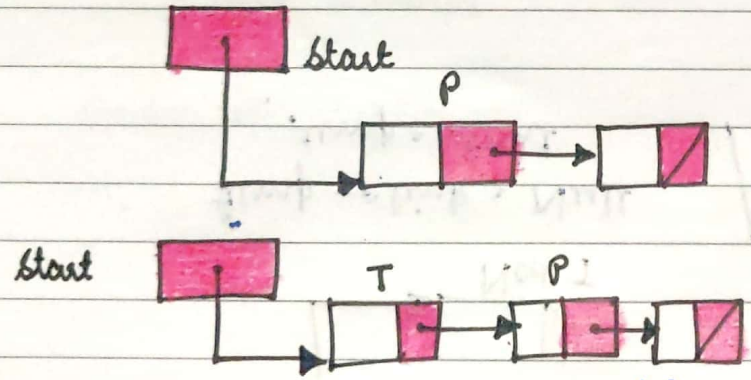
temp → is a pointer that point to this dynamically allocated node.

```
temp = (struct node *) malloc( sizeof (struct node));
temp → info = data;
```


Insertion at the beginning of the list

① insert node 'T' at the beginning of the list.

② Suppose the first node of the list is 'P', we need to add 'T' before P.



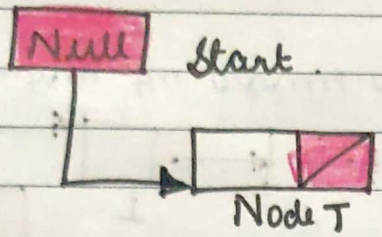
a) 'T' should contain the address of P. which is available in start.

Temp → link = start

(Order of this stmt to be maintained, because only start has address of 'P', if we point start = temp, we will loose the address of P.)

b) 'T' to be first node :- start = temp;

Insertion in an empty list

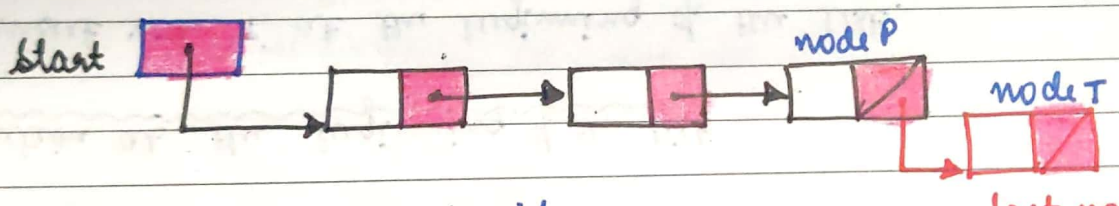


temp → link = Null
temp = Start

/

temp → link = Start.
(Since Start initially was Null)

Insertion at the end of the list



a) We should have a pointer 'P' pointing to node P.

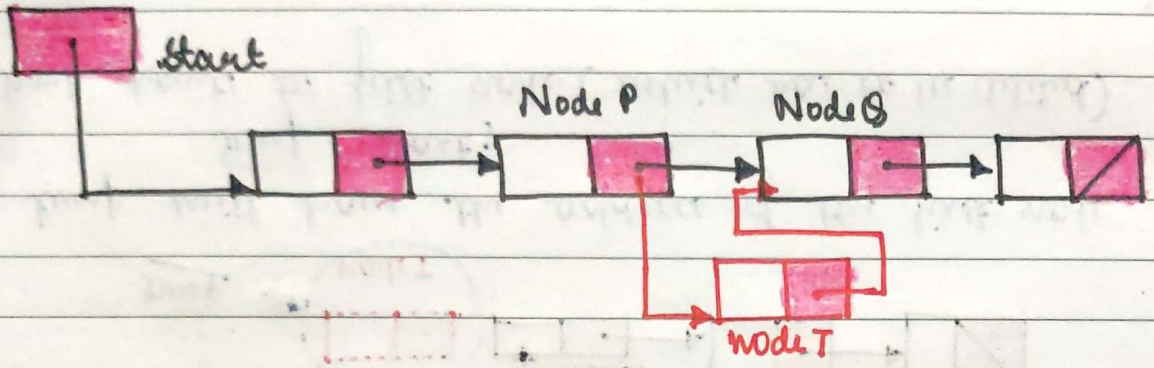
then P → link = temp;
temp → link = Null;

} P pointing to temp

b) information about the linked list is available at Start, We need to traverse the entire list to get to pointer P.

```
P = start;
while ( p -> link != NULL)
P = p -> link ;
```
} reach p.

Insertion in between the list nodes



we have two pointers P & Q pointing to P & Q node.

```
temp -> link = (Q) address is init P -> link;
P -> link = temp;
```

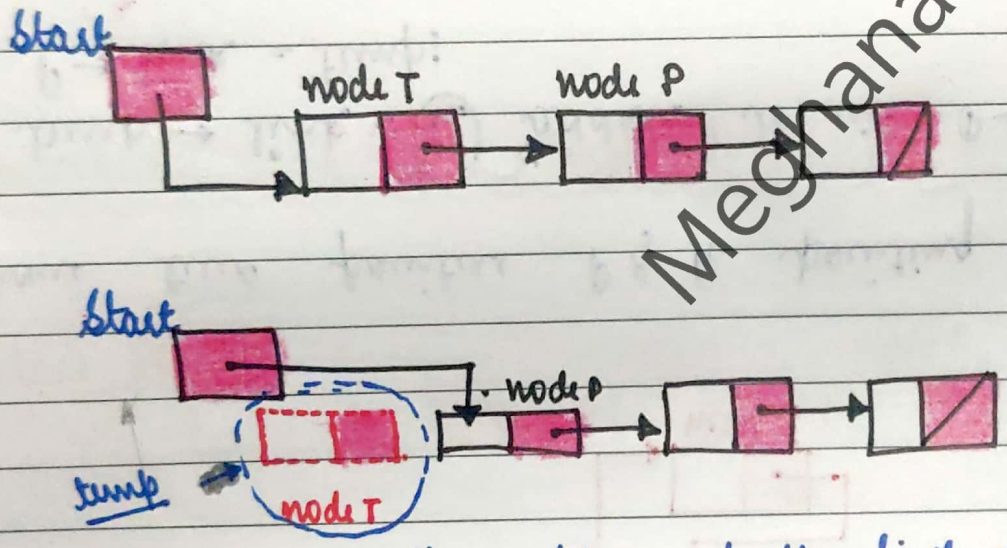
order is imp(※)

Deletion in a Single linked list

1. Deletion of first node
2. Deletion of the only node.
3. deletion in between the list
4. Deletion at the end

} all cases we call free(temp) to physically remove 'node T' from the memory.

Deletion of first node



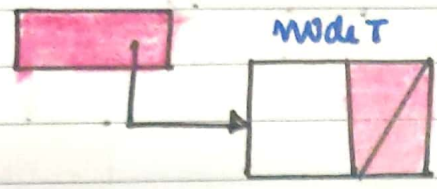
temp will have the address of the first node.

temp = start;

temp points to first node (which has to be deleted)

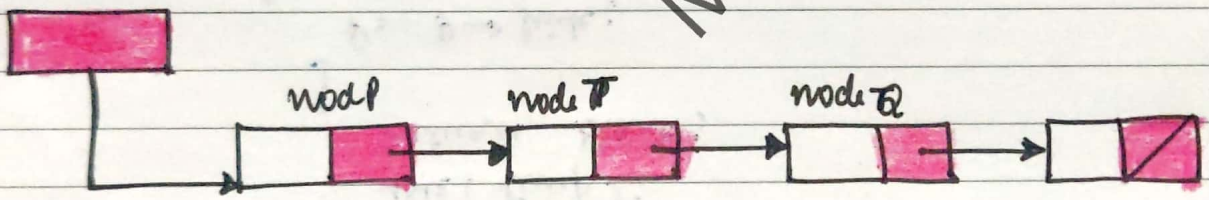
start = start → link (will have address of P)

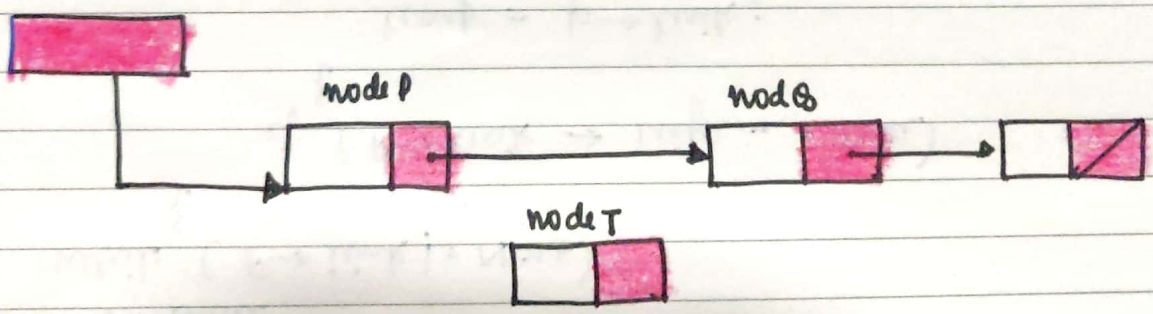
Deletion of the Only node



temp = Start;
Start = In Null;

Deletion in between the list nodes





Suppose temp pointer point to node T,

2 pointers

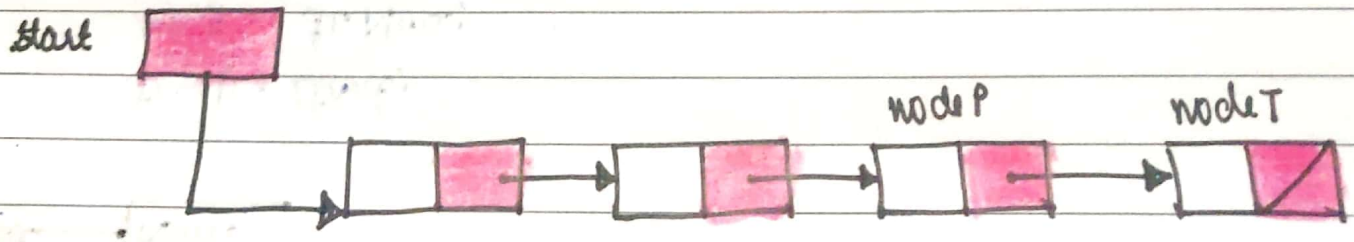
P Q (pointing to P & Q respectively)

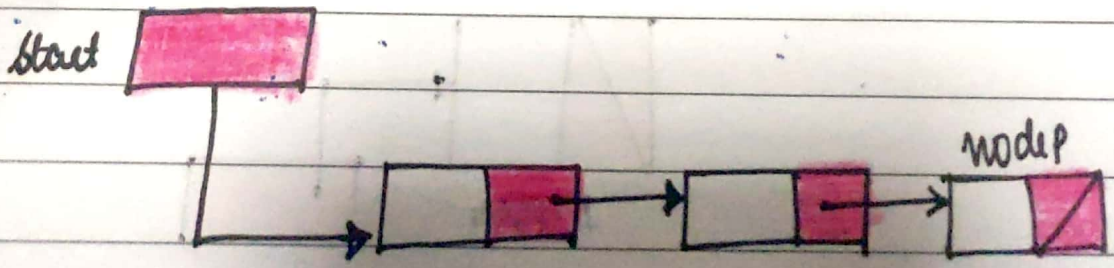
P → link = Q (address of Q
is in temp → link)

P → link = temp → link

```
P = start ;
while ( P → link != NULL)
    {
        if ( P → link → info == data)
            {
                temp = p → link ;
                P → link = temp → link ;
                free ( temp ) ;
                return start ;
            }
            P = P → link ;
    }
```


Deletion at the end of the list





if 'P' is a pointer to node P,
then node 'T' can be deleted by writing the following:-

```
P -> link = Null;
P -> link = temp -> link
    free(temp)
```

Doubly linked list

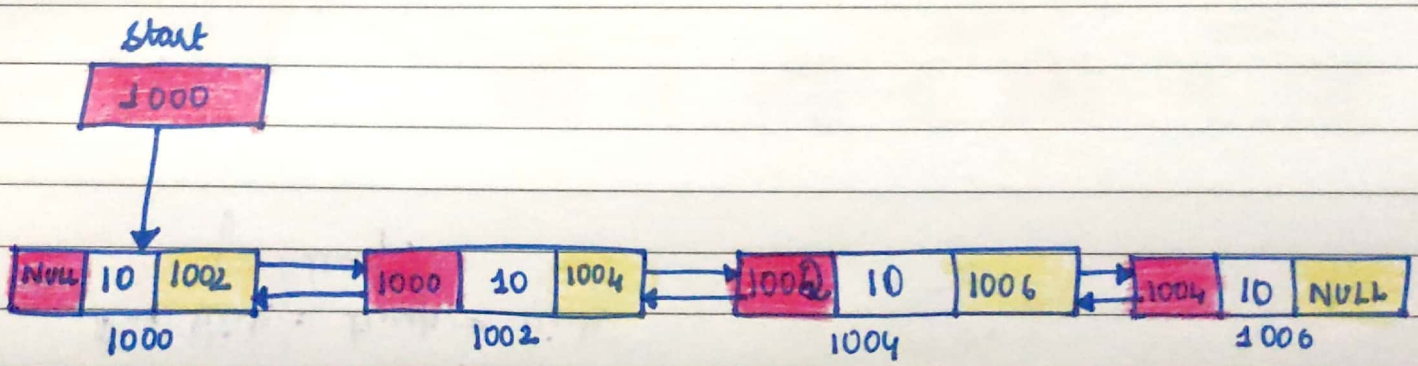
1. Doubly linked list.
2. Memory representation.
3. Operations.
4. Example program.

Definition :-

1. A double linked list is a two-way list in which all nodes will have two links.
2. Each node contains three fields :

   Left link
   data
   Right link

3. This helps in accessing both successor node and predecessor node from the given node position.



Operations

1. Create a new node
2. Create a list of nodes with data
3. Insertion :- a) beginning
   b) end
   c) between
4. Deletion :- a) beginning
   b) end
   c) between
5. Traverse node

Define a node

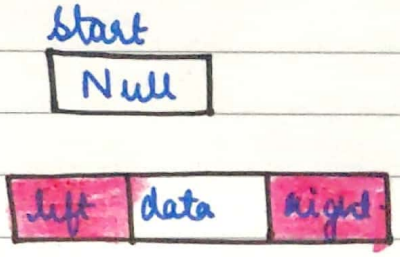
```
struct dlinklist
{
    struct dlinklist *left;
    int data;
    struct dlinklist *right;
};

typedef struct dlinklist node;

node * start = NULL;
```

Structure

node is the name of the structure (alias)



(Create a node)

return struct

returns address of new node.

```
node * getnode()
{
  node * newnode;
  newnode = (node*) malloc (sizeof (node));
  Pf("Enter data");
  Sf("%d &newnode->data);

  newnode -> left = NULL;
  newnode -> right = NULL;
  return newnode;
}
```

address of new node.

create a block of node memory.

Create a doubly LL of n nodes

n → no of elements, we will create.

```
void createlist (int n)
{
  int i;
  node * newnode, *temp;

(each iteration for (i=0; i<n; i++)
to get 1 new
node)
        if (start == Null)
          start = newnode
        else
          temp = start (will point to start)
```

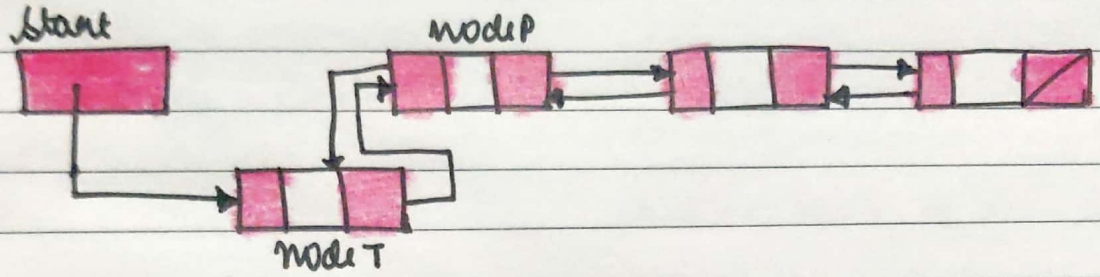
```
while (temp->right != NULL)
temp = temp -> right
temp -> right = newnode
```

new node.

Traversing a doubly linked list

```
void display (struct node * start)
{
    struct node *p;
    if (start == Null)
{
    Pf ("list is empty\n");
    return;
}
    P = start;
    Pf ("List is :\n");
    while (P != NULL)
    {
        Pf ("%d", P->info);
        P = P -> next;
    }
    Pf ("\n");
```

① Inserting at the beginning of the list



'T' is first node

T' prev → null

temp → prev = null;

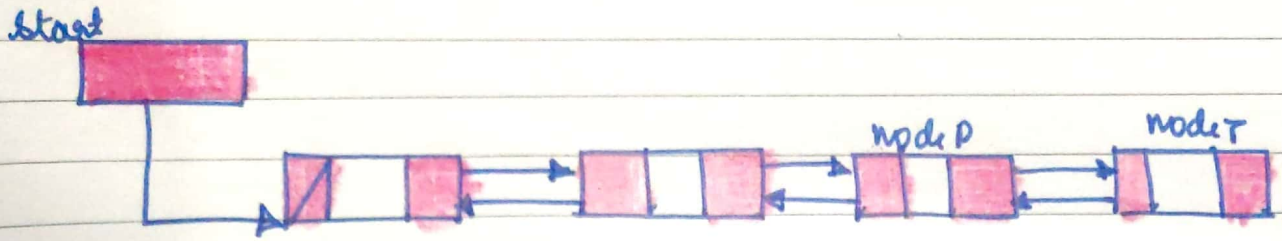
temp → next = start (to give the address of node P).

previous of node 'P' now should point to node T

start → prev = temp;

start = temp (since 'node T' is the first node).

② Insertion at the end of the list

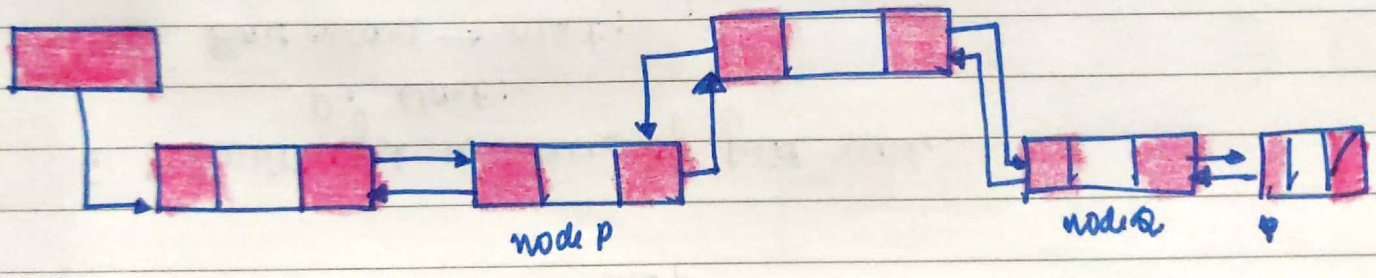


temp→next = Null (last node)

p→next = temp;

temp→prev = p;

③ Insertion in between the node



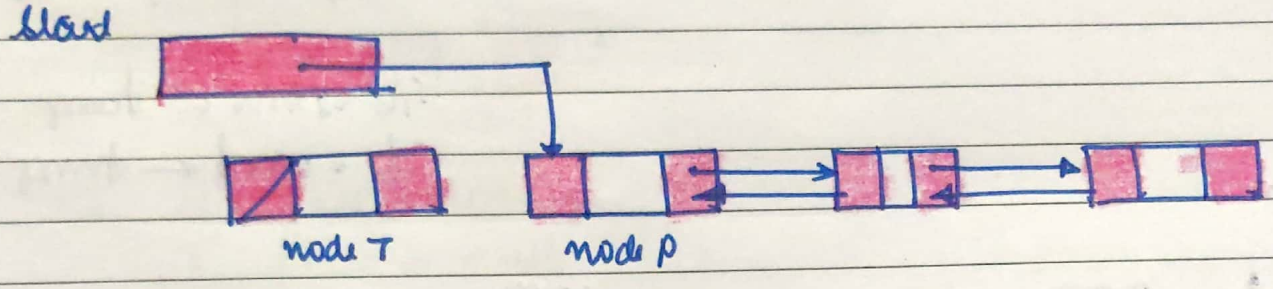
temp→prev = p;
temp→next = q;

q→prev = temp;

p→next = temp;

Deletion from doubly list

① Deletion of the first node.



① P will get address of first node.

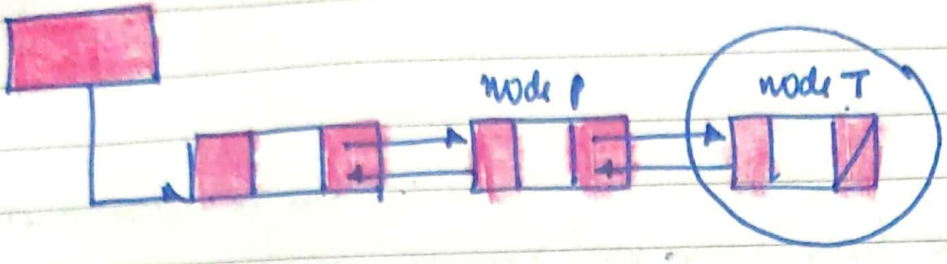
```
P = start;
start = start → next;
```

'P' previous should contain Null.

```
start → prev = null;
```

```
temp = start;
start = Null
```

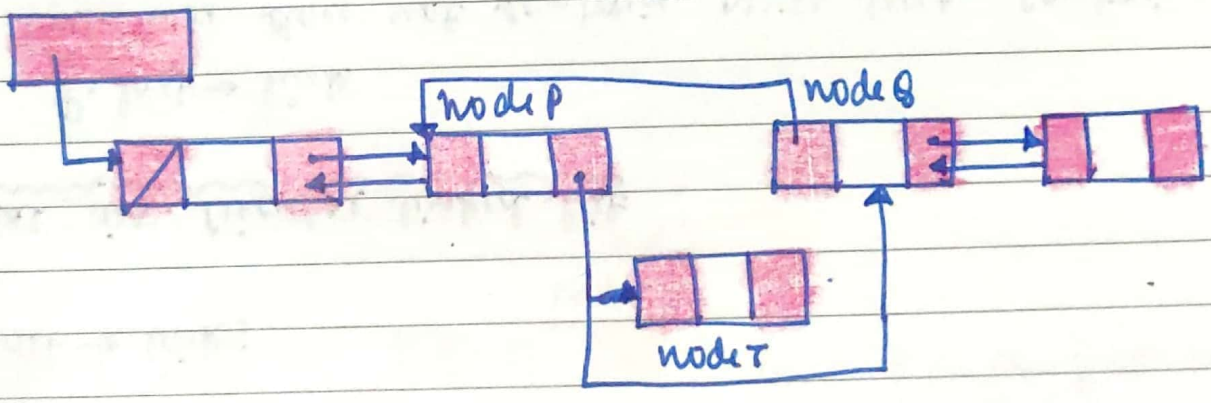
Deletion at the end of the list



P → next = Null

temp → prev → next = Null;

Deletion in between the nodes



P → next = Q;
Q → prev = P;

P → next = temp → next;
temp → next → prev = p;

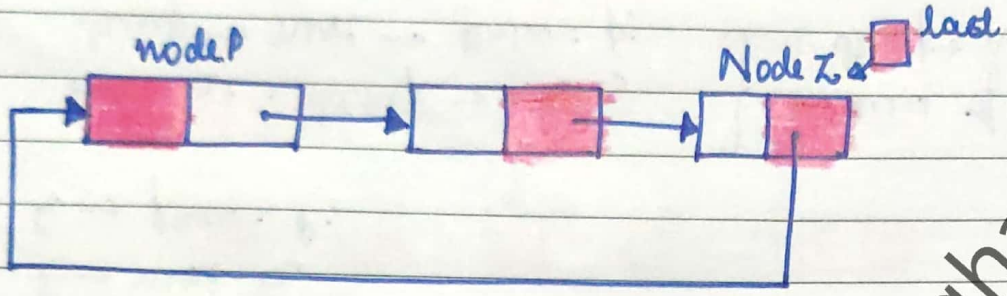
} address of q is in temp → next

address of P is stored in temp → prev
temp → prev → next = temp → next;
temp → next → prev = temp → prev.

} we need only pointers to node for deleting it.

Circular linked list

Traversal in circular linked list



P = last → link;

Traversal in circular linked list

P = last → link

link of last node does not contain NULL but contains the address of first node.

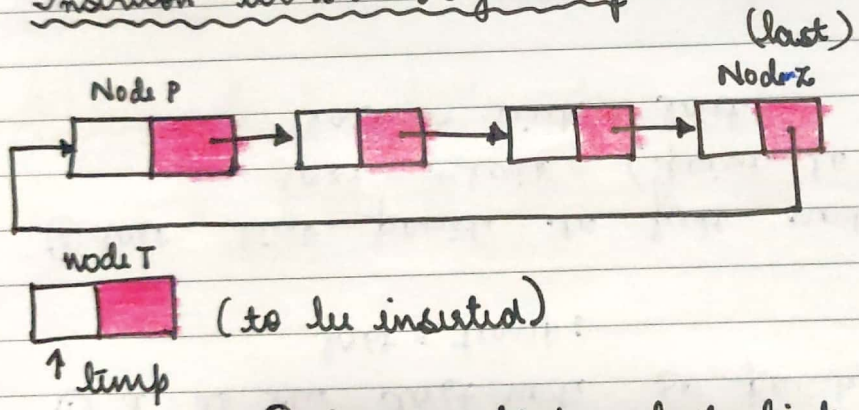
So, terminating condition of loop becomes:

(P != last → link)

※ We have to use do-while loop in the display function because if we take a while loop then the terminating condition will be satisfied in the first time only and the loop will not execute at all.

```
void display ( struct node * last)
{
    struct node *p;
    if (last == Null)
    {
      Pf ("list is empty \n");
      return;
    }
    P = last -> link;
    do
    {
      Pf ("%d", P -> info);
      P = P -> link;
    } while ( P != last -> link);
    Pf (" \n");
}
```


Insertion at the beginning



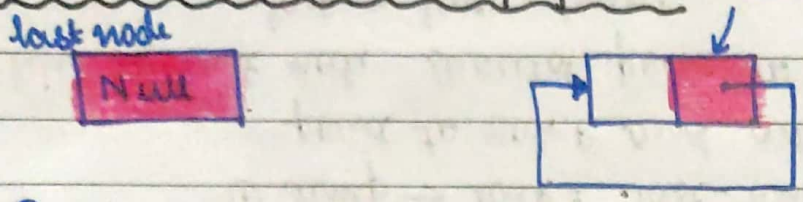
Since 'P' is the first node so last → link points to P.

① temp → link = last - link; (link of node T should point to node P and address of node P is in last → link

② link of last node should point to T.

last → link = temp;

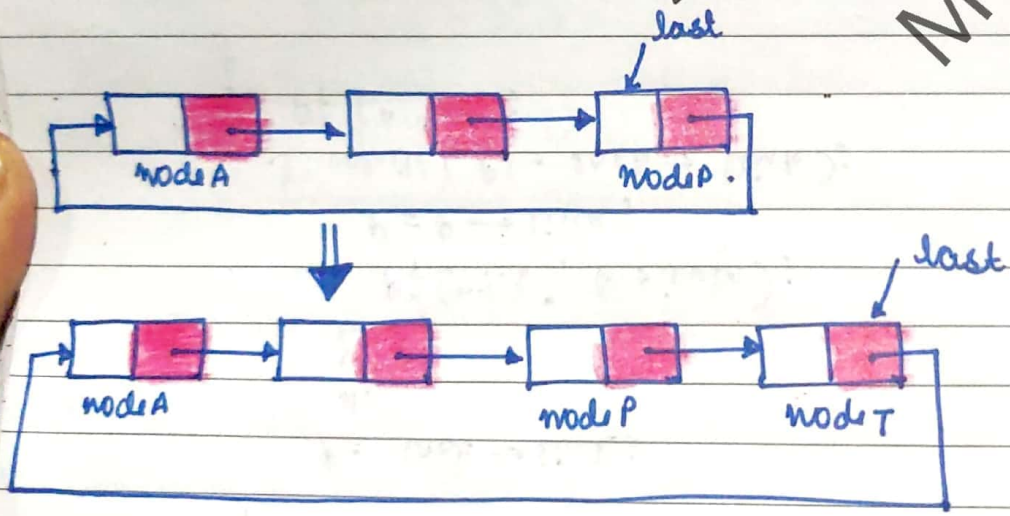
Insertion in an empty list



① 'T' is the last node so pointer point to last
last = temp;

② last link points to first node, here T is the 1st node.
last → link (point to node T (or last))
last → link = last.

Insertion at the end of the list



① Before insertion, last points to node P, last → link points t
node A.

② T should point to 'A' (to fetch the address)
temp → link = last → link;

③ node P should point to node T.

```
last → link = temp;
```

③ last should point to 'node T'

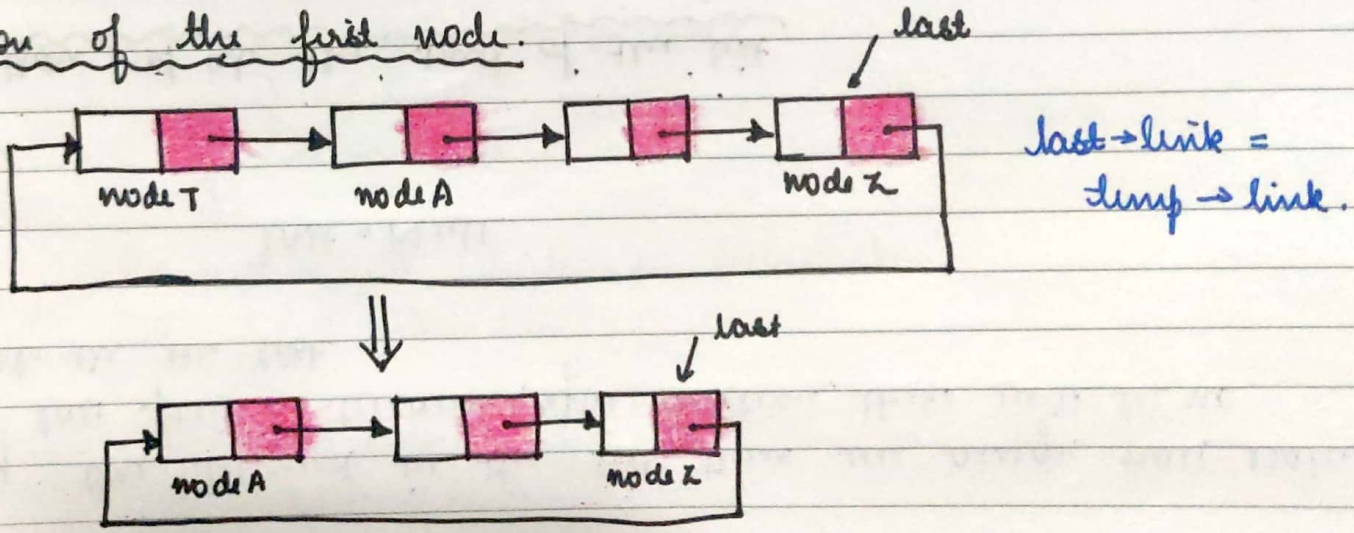
```
last = temp;
```

Insertion in between the nodes

```
P = last → link;
temp → link = p → link;
P → link = temp;
last = temp;
```

Deletion

deletion of the first node.



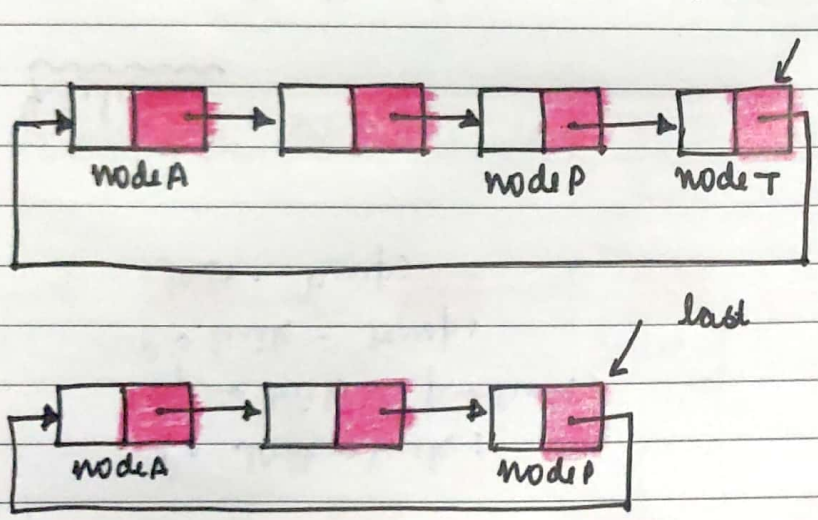
last → link = temp → link.

deletion of the only node

Only one element in the list then we assign null value to last pointer because after deletion there will be no node in the list.

last = Null.

deletion of at the end of the list



① now last points to nodeT last → link point to nodeA, 'P' points to node P.

after deletion link of node P should point to node A 'address of A' is in last → link;

① P → link = last → link

now P is the last node so last should point to P.

last = P;

Program :-

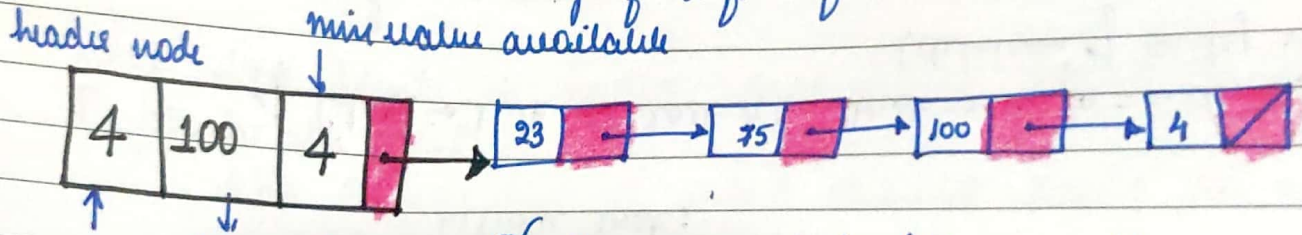
```
struct node * del ( struct node * last, int data)
{
    struct node * temp, * p;
    if (last == Null)
    {
        pf ("list is empty\n");
        return last;
    }
if (last -> link == last && last -> info == data)
                                (deletion of only node)
    {
        temp = last;
        last = null;
        free (temp);
        return last;
    }
if (last -> link -> info == data) (deletion of first node)
    {
        temp = last -> link;
        last -> link = temp -> link;
        free (temp);
        return last;
    }
P = last -> link (deletion in between)
while (P -> link != last)
    {
    if ( P -> link -> info == data)
    {   temp = p -> link;
        P -> link = temp -> link;
        free (temp);
        return last;
    }  P = P -> link;
```

```
if (last -> info == data)  (deletion of last node)
{
    temp = last;
    p -> link = last -> link;
    last = p;
    free (temp);
    return last;
}
Pf ("Element %d not found \n", data);
return last;
}
```


Header Linked List

① The structure of the header node is different compared to the structure of remaining node.

② header node → summary of info of other nodes.



③ (now no need to traverse the list)

④ insertion / deletion update of count and info should be made.

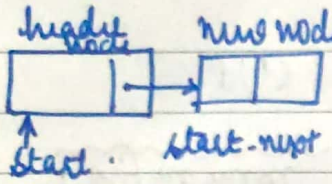
⑤ ptr = start

ptr = start → next (node is one next to start)

header linked list insertion at the first node.

header will always remain as the first node.

algo { algo hll_insert_fn (start, newinfo)



{

new_node = getnode()

new_node → info = newinfo

new_node → next = start → next

start → next = new node.

start → ncount = start ncount + 1 → (node count increased by 1)

if (start → maxinfo < new info)

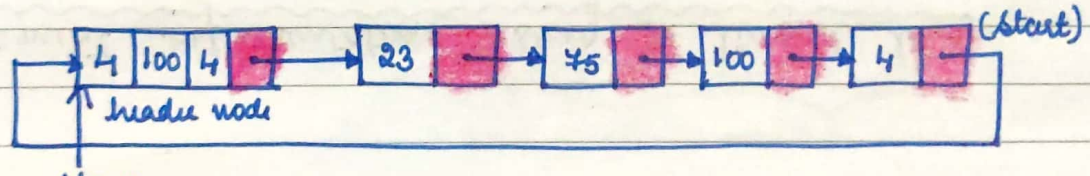
start → max info = newinfo

if (start → minfo > newinfo)

start → minfo = new info

}

Circular header linked list :-



ptr = (start) instead of start next → not for circular

while (ptr ≠ null) → (ptr ≠ start)

Print ptr → info

p = ptr → next;

}

Applications of Linked List (please go through)

1. Polynomial addition
2. Reverse linked list
3. Searching on sorted linked list
4. Searching on unsorted linked list
5. Finding maximum and minimum node

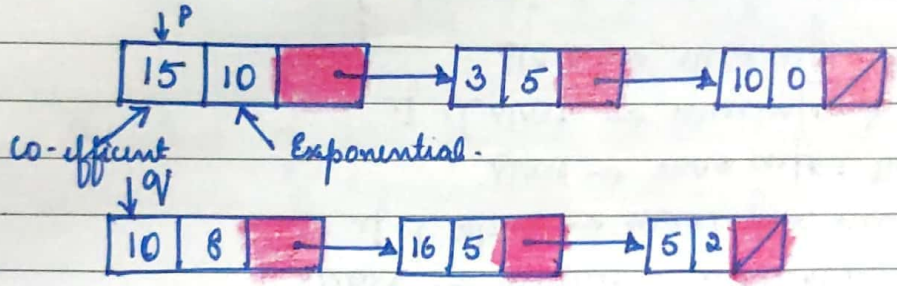
} not there in the syllabus.

6.

Polynomial representation using a linked list

$P(x) = 15x^{10} + 3x^5 + 10$

$Q(x) = 10x^8 + 16x^5 + 5x^2$



① P compared to q
P > q (copy)
P++

② Likewise cond and copy the remaining nodes.

Polynomial addition using linked list

$P(x) + Q(x) = 15x^{10} + 10x^8 + 19x^5 + 5x^2 + 10$

R: 

Algorithm :- algo poly_add (poly1, poly2, poly) — 2 expressions → resultant.

{

ptr1 = poly1, ptr2 = poly2, ptr = newnode( ), poly = ptr

while (ptr1 → next and ptr2 → next) (nodes to be processed)

{

if (ptr1 → exp > ptr2 → exp)

{

ptr → coef = ptr1 → coef
ptr → exp = ptr1 → exp
ptr1 = ptr1 → next.

}

else if (ptr1 → exp < ptr2 → exp)

{

ptr → coef = ptr2 → coef
ptr → exp = ptr2 → exp
ptr2 = ptr2 → next

}

else

{

if co-efficients are same (add)

ptr → coef = ptr1 → coef + ptr2 → coeff
ptr → exp = ptr1 → exp
ptr1 = ptr1 → next, ptr2 = ptr2 → next

go to next node.

}

ptr → next = newnode( ), ptr = ptr → next

}

not null — dump the value to resultant

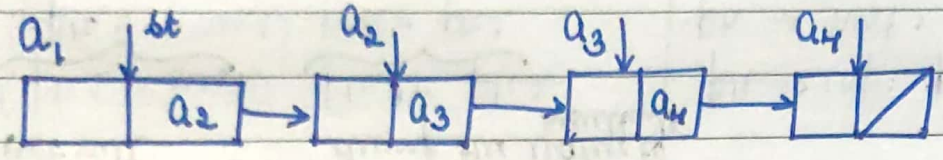
if (ptr1 → next) ptr = ptr1
if (ptr2 → next) ptr = ptr2

while (ptr → next)

{ ptr → coef = ptr → coef
ptr → exp = ptr → exp
ptr = newnode( )
ptr = ptr → next
ptr → next = null }

copy until last P1, P2 turns null

② Reverse of a linked list



algo ll_reverse (st)

// a chain pointed at by st = {a1, a2 ... am}
// is reversed so that after execution it will be
st = {am, ... a2, a1}.

※ (Only link will be reversed) ※

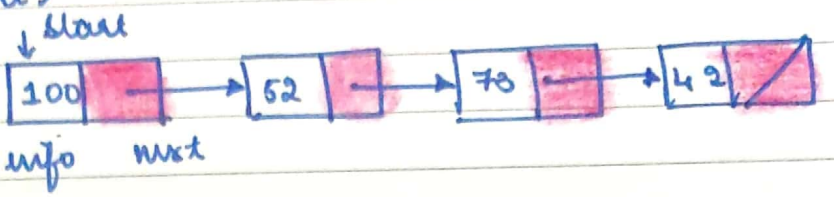
```
{
  ptr ← st
  q ← 0
  while (ptr ≠ null)
   {
     r ← q
     q ← ptr
     ptr ← ptr → next
     q → next = r
   }
   st = q
}
```

trace.

st	ptr	r	q
a1	a1	0	0
	a2	0	a1
	a3	a1	a2
	a4	a2	a3
	.		

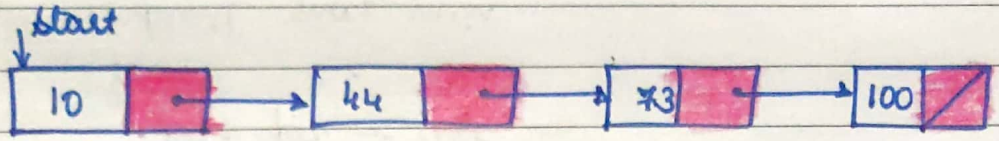
③ Calculate maximum and minimum node of a linked list

```
algo linkedlist - max min (start)
{
  ptr = start
  flag = 1
  while ( ptr ≠ null)
  {
    if ( flag == 1 )
    {
      max = ptr → info
      min = ptr → info
      flag = 0.
    }
    else
    {
      if (max < ptr → info)
      {
        max = ptr → info)
      }
      if (min > ptr → info)
      {
        min = ptr → info
      }
    }
    ptr = ptr → next
  }
  print max, min
}
```



first node value is both min & max values.

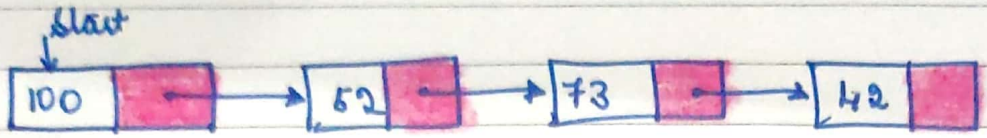
④ Linked list Searching On Sorted linked list



item = 105

```
algo ll sorted search ( start, item)
{
   ptr = start
  while ( ptr ≠ null and ptr → info < item )
    {
       ptr = ptr → next
    }
    if ( ptr → info == item)
    {
     Print "Succesful searching at", ptr)
  }
    else
    {
      print "Unsuccesful search"
    }
}
```

⑤ Linked list searching on Unsorted linked list




```
Algo U_unsorted_search ( Start, item )
{
    ptr = Start
  while ( ptr ≠ null and ptr → info ≠ item)
    {
      ptr = ptr → next
    }
     if (ptr == null)
    {
     print "Unsuccessful search"
    }
     else
      {
        print "Found at", ptr
      }
}
```

Assume the following.

Representing chains in C

For representing chains in C we require the following capabilities.

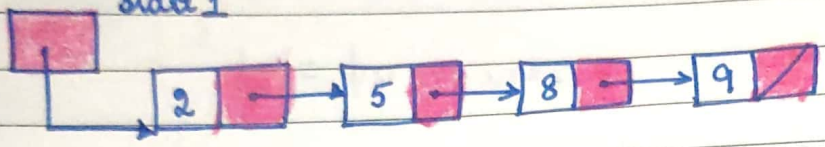
a) A mechanism for defining a node's structure
b) A way to create new nodes using malloc
c) A way to remove nodes that are no longer need using free()

Applications (if asked in exam)

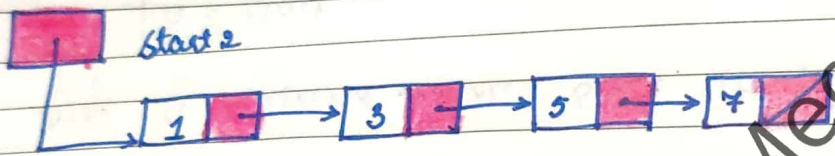
Merging.

two sorted lists.

① Start 1

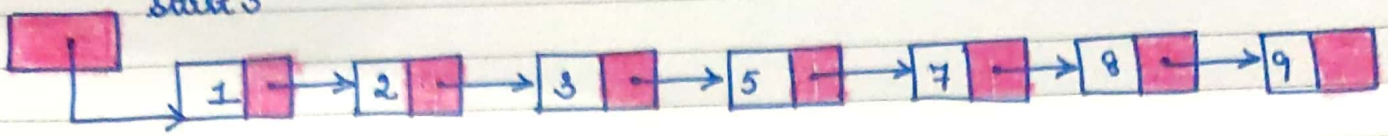


Start 2



merged list

start 3



① 2 pointers considered P_1, P_2.

3 cases while comparing $P_1 \rightarrow$ info and $P_2 \rightarrow$ info.

a) if $(P_1 \rightarrow \text{info}) < (P_2 \rightarrow \text{info})$

the new node that is added to the resultant list has info equal to $P_1 \rightarrow$ info. After this we will make P_1 point to the next node of first list.

b) if $(P_0 \rightarrow \text{info}) < (P_1 \rightarrow \text{info})$

the new node that is added to the resultant list has info equal to $P_2 \rightarrow$ info. After this we will make P_2 point to the next node of second list.

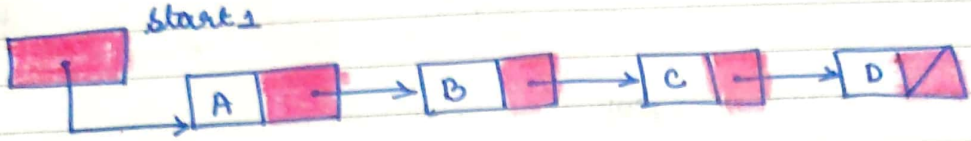
c) if $(P_1 \rightarrow \text{info}) = (P_0 \rightarrow \text{info})$

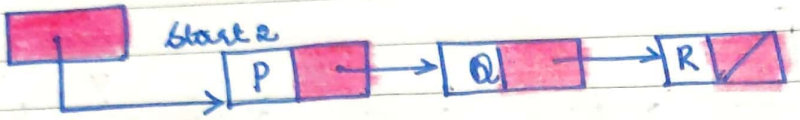
the new node that is added to the resultant list has info equal to $P_1 \rightarrow$ info or $P_2 \rightarrow$ info. After this we will make P_1 and P_2 point to the next nodes of first and second list.

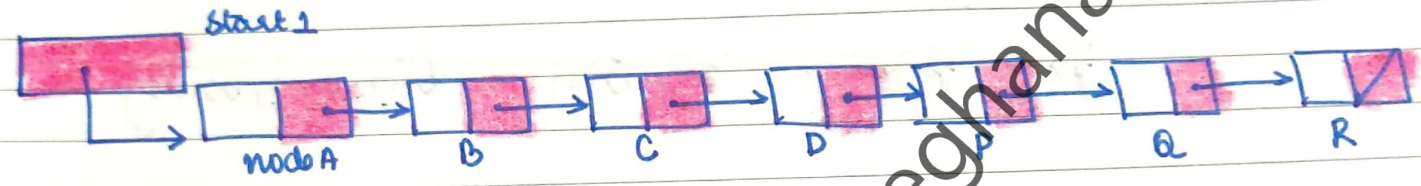
```
while (p1 != null && p2 != Null)
{
    if (P1 -> info < p2 -> info)
    {
        start 3 = insert (start 3, p1 -> info);
        P1 = P1 -> link;
    }
    else if (P2 -> info < P1 -> info)
    {
        start 3 = insert (start 3, p2 -> info);
        P2 = p2 -> link;
    }
    else if (P1 -> info == P2 -> info)
    {
        start 3 = insert (start 3, p1 -> info);
        p1 = p1 -> next
        p2 = p2 -> next
    }
}
```

Concatination :-

Suppose we have two single linked lists and we want to append one at the end of another. For this the link of last node of first list should point to the first node of the second list.







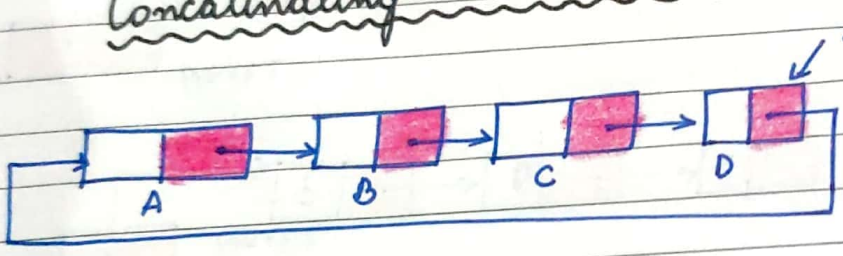
for concatination, link of node D should point to node P. To g
the address of node D, we have to traverse the first list till t
end. Suppose ptr points to node D, then ptr → link should
be made equal to Start 2.

```
ptr → link = Start 2;
```

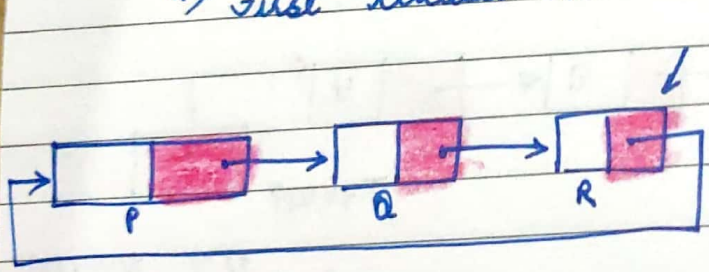
```
Concat
Struct node *.. (Struct node * Start 1, Struct node *Start
{
    Struct node *ptr;
    if (Start 1 == Null)
```

```
{
    start1 = start2;
    return start1;
}
    if (start2 == Null)
        return start1;
    ptr = start1;
while (ptr -> link != Null)
    ptr = ptr -> link;
ptr -> link = start2;
    return start1;
}
```

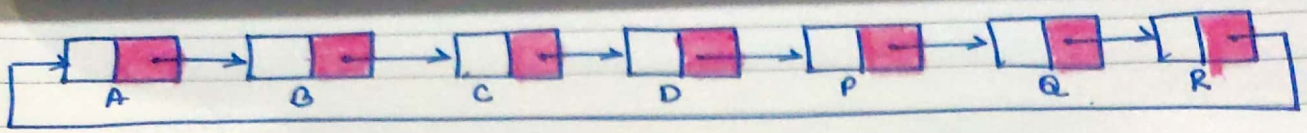
Concatinating Circular Linked List



a) First circular LL



b) Second circular LL



Concatenated list

Link of node D should point to node P.

last 1 → link = last 2 → link;

We will lose the address of node A, so before writing this statement we should save

last 1 → link.

ptr = last 1 → link.

Link of node R should point to node A

last 2 → link = ptr;

the pointer last 1 should point to node R

last 1 = last 2;

```
Struct node * concat (Struct node * last1, Struct node *
                                        last2)
{
  Struct node * ptr;
    if (last1 == Null)
  {
    last1 = last2;
    return last1;
  }
  if (last2 == Null)
    return last1;

  ptr = last1 -> link;
  last1 -> link = last2 -> link;
  last2 -> link = ptr;
  last1 = last2;
  return last1;
```


Disadvantage of LL

1. In arrays we can access the n^{th} element directly, but in linked list we have to pass through the first $n-1$ elements to reach the n^{th} element. So when it comes to random access, array lists are definitely better than linked lists.
2. In LL the pointer fields take extra space.
3. Writing programs for ll is more difficult than that of arrays.