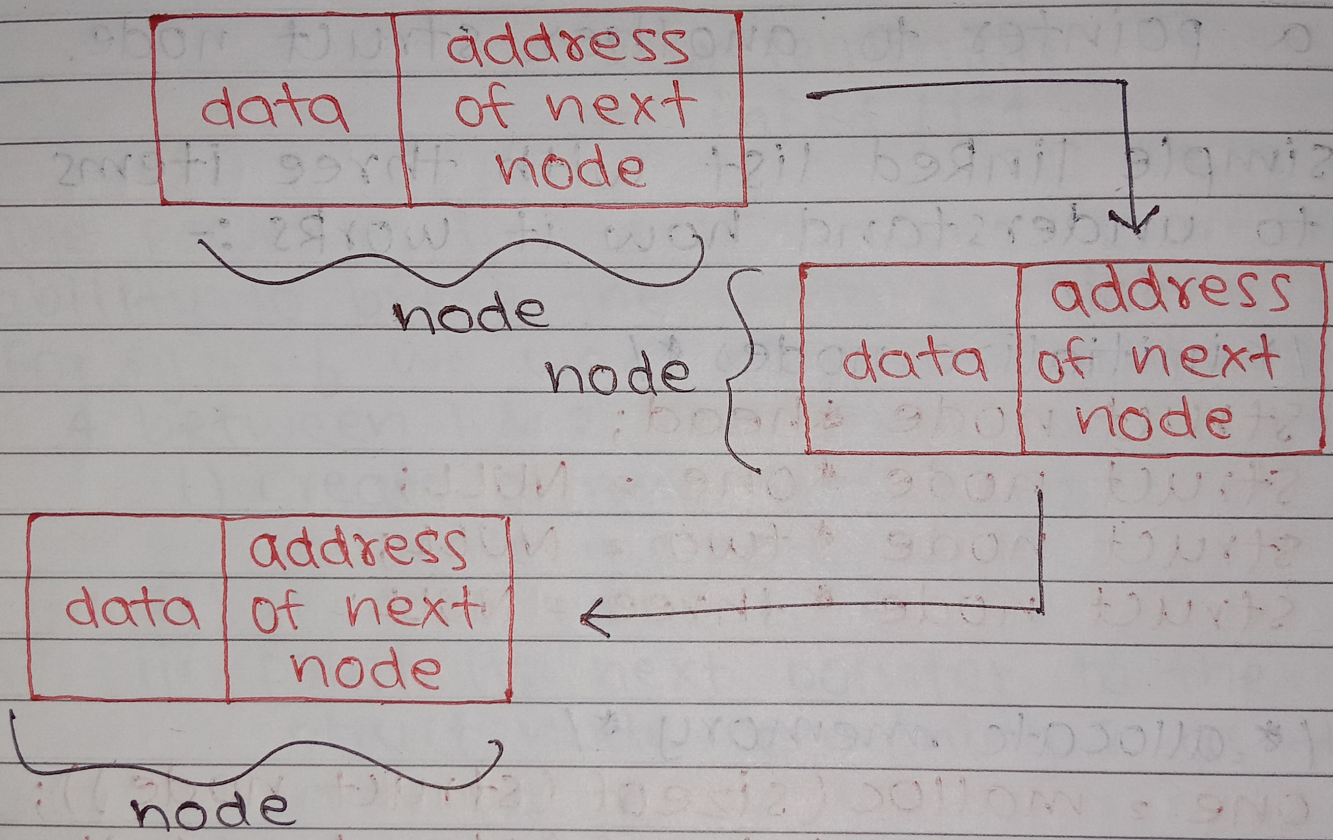
# Linked List

A Linked List is a Linear Data structure that includes a series of connected nodes. Here each node stores the data & the address of the node / next node. For eg :-




* Representation of a Linked List :-
There are a series of connected nodes in Linked List & each node consists :-
  i) a data item.
  ii) an address of another node.
we wrap both the data item & the next node reference in a structure as :-

(2)

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

Each struct node has a data item & a pointer to another struct node.

* simple linked list with three items to understand how it works :-

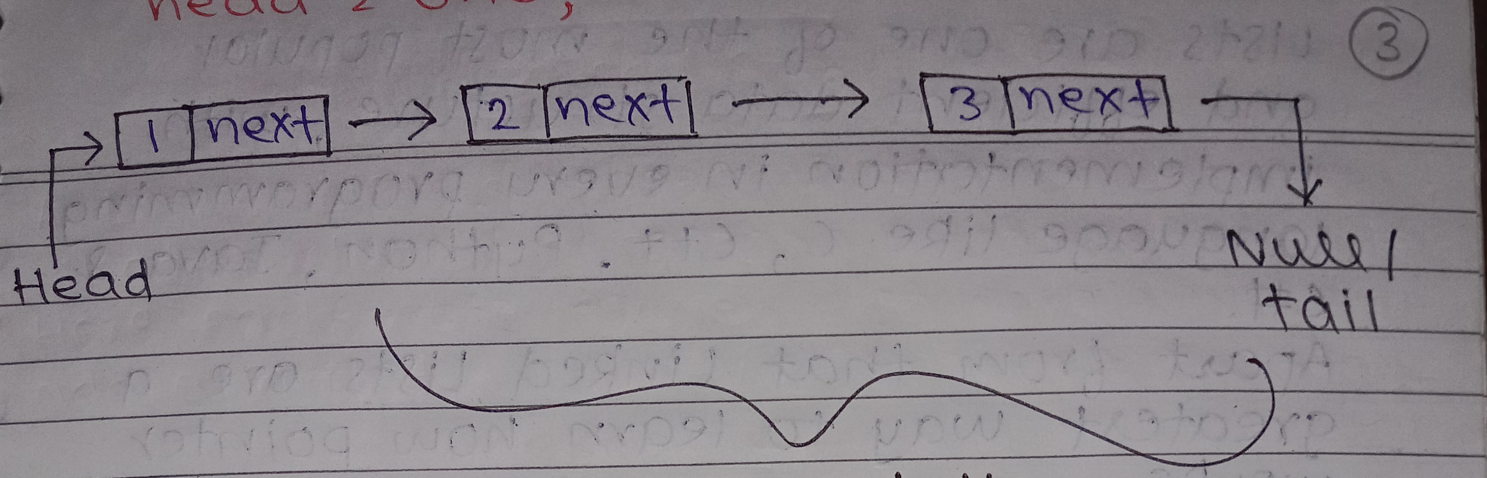
```
/* initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* assigning data values */
one -> data = 1;
two -> data = 2;
three -> data = 3;

/* connect nodes */
one -> next = two;
two -> next = three;
three -> next = NULL;
```

```
/* save address of first node in head */
head = one;
```




Representation of Linked List

The power of linked list comes from the ability to break the chain & rejoin it.
For eg :- if we want to put an element 4 between 1 & 2, the steps would be :-

i) create a new struct node & allocate memory to it.
ii) it data value is 4.
iii) point its next pointer to the struct node containing 2 as the data value.
iv) change the next pointer of 1 to the node we just created.

* doing something similar in an array would have required shifting the positions of all the elements.

## * Linked List utility :-

Lists are one of the most popular and efficient data structure, with implementation in every programming language like C, C++, Python, Java & C#.
Apart from that Linked Lists are a greatest way to learn how pointer works.

## * Linked List implementation in C :-

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

void printLinkedList(struct node *p){
    while(p != NULL){
        printf("%d ", p -> value);
        p = p -> next;
    }
}

int main(){
    struct node *head;
    struct node *one = NULL;
    struct node *two = NULL;
    struct node *three = NULL;
```

(5)

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

one -> value = 1;
two -> value = 2;
three -> value = 3;

one -> next = two;
two -> next = three;
three -> next = NULL;

head = one;

printLinkedList(head);
}
```

Output of above program :-

1 2 3

* **LinKed List Application :-**
  i) Dynamic memory allocation.
  ii) implemented in stack & Queue.
  iii) in undo functionality & softwares.
  iv) hash tables & graphs.

* **Advantages of LinKed List :-**
  i) we can keep adding & removing elements without any capacity constraints.

* Disadvantages of Linked List :-

(6)

i) extra memory space for pointer is required (for every node 1 pointer is needed).

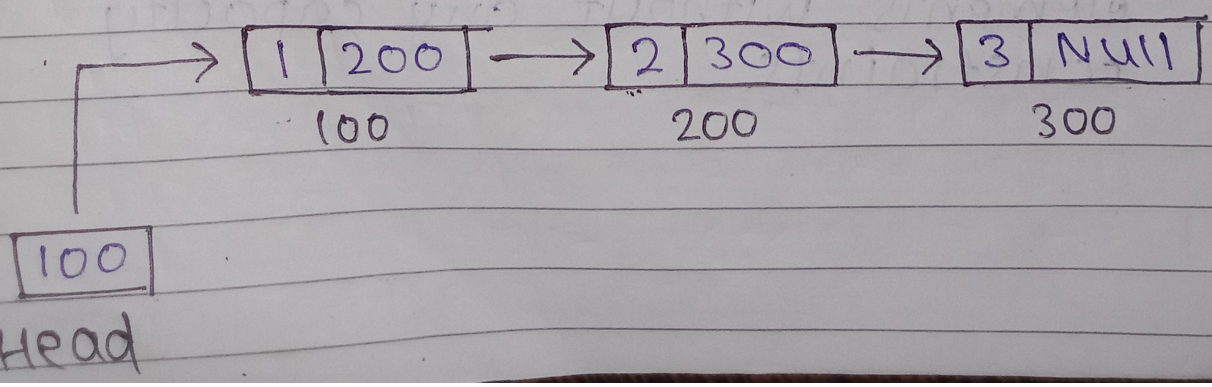
ii) random access not allowed as elements are not stored in contigous memory location).

## Types of Linked List

The followings are the types of linked list :-

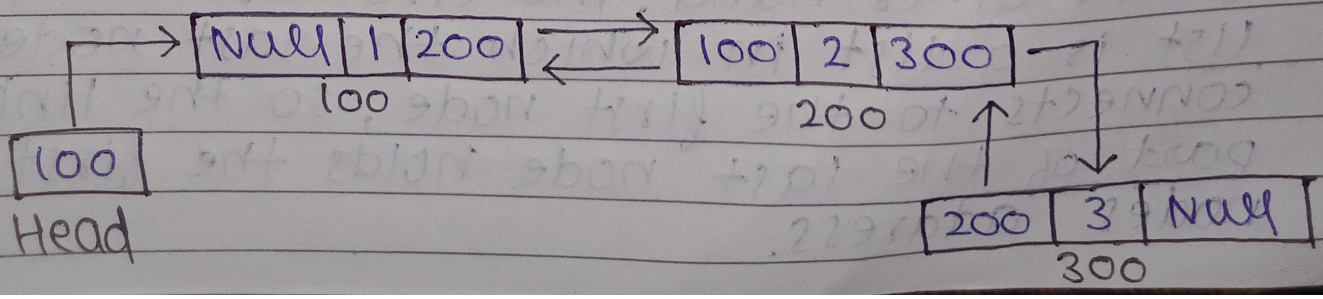
i) Singly Linked List
ii) Doubly Linked List
iii) Circular Linked List
iv) Doubly Circular Linked List.

* Singly Linked List = It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a Singly Linked List.
It contains two parts, data part & pointer part.
Suppose, we have 3 nodes & the addresses of these three nodes are 100, 200 & 300.
The representation of 3 nodes as a linked list is shown below :-



There are three different nodes having address 100, 200 & 300. The first node contains the address of the node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the null value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as Head Pointer.
The linked list, which is shown previous is known as singly linked list.
In singly linked list, only forward traversal is possible, we can not traver-se in the backward direction.

* Doubly Linked List = The doubly linked list contains 2 pointers. we can define the doubly linked list as a linear data structure with 3 parts, the data part & the other 2 address part.
Doubly linked list is a list that has 3 parts in a single node, includes one data part, a pointer to previous node & a pointer to the next node.
Suppose, we have 3 nodes, and the address of these nodes are 100, 200 and 300. The representation of these nodes in a doubly linked list are :-



The node in doubly linked list has two address part, one part stores the previous node's address while the other part of the node stores the next node's address.
The initial node in the doubly linked list has the null value in the address part which provides the address of the previous node.

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

Representation of Singly Linked List.

Representation of Doubly Linked List.

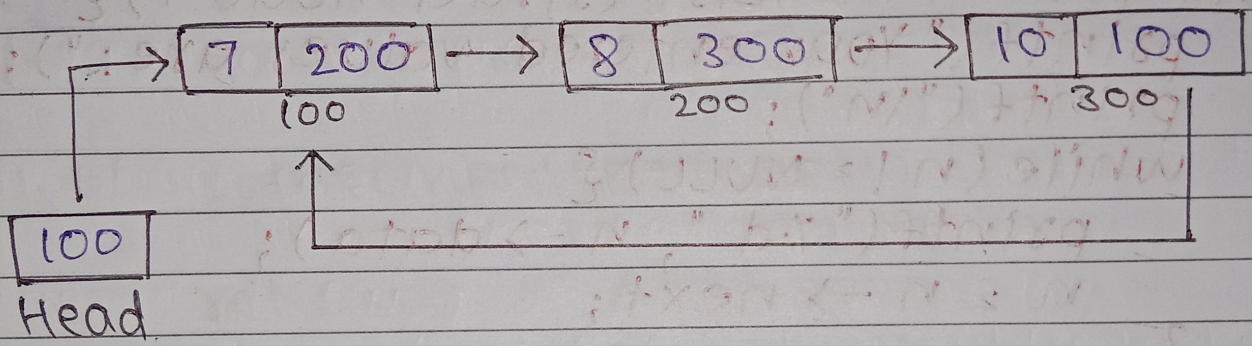
```
struct node {
   int data;
   struct node *pre;
   struct node *nex;
};
```

* Circular Linked List = A circular linked list is a variation of singly linked list. The only difference b/w the singly linked list and a circular linked list is that the last node does not point to any node in a singly linked list, so its link part contains a null value.
On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first nodes address.

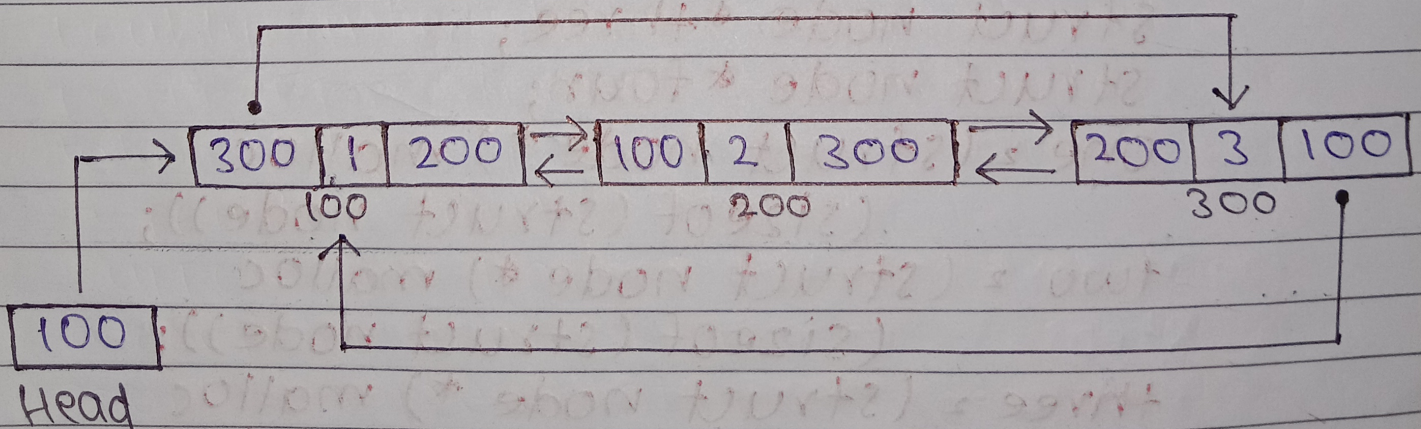
The circular linked list has no starting & ending node. We can traverse in any direction, i.e., either backward or forward.

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

Representation of circular linked list.




* Doubly circular linked list : The doubly linked list has the features of both the circular linked list as well as doubly linked list.



* Program to take the value of linked list from user & print.

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
void printLinRedList (struct node *) {
    printf("Your entered values are:");
    printf("\n");
    while(n != NULL) {
        printf("%d ", n -> data);
        n = n -> next;
    }
}
int main() {
    struct node *one;
    struct node *two;
    struct node *three;
    struct node *four;
    one = (struct node *) malloc
            (sizeof (struct node));
    two = (struct node *) malloc
            (sizeof (struct node));
    three = (struct node *) malloc
            (sizeof(struct node));
    four = (struct node *) malloc
            (sizeof (struct node));
```

```
printf("enter first node data value");
scanf("%d", &one->data);
one->next = two;
printf("enter second node data value");
scanf("%d", &two->data);
two->next = three;
printf("enter third node data value");
scanf("%d", &three->data);
three->next = four;
printf("enter four node data value");
scanf("%d", &four->data);
four->next = NULL;

printLinkedList(one);

return 0;
}
```

## * Insert elements to a Linked List.

We can add elements either in the beginning, in the end or in the specific position.

### ① Insert at the beginning

Steps to insert new node at the beginning :-

i) allocate memory for new node.
ii) store data.
iii) change next of new node to point to head.
iv) change head to point to recen -tly created node.

```
struct node *newNode;
newNode = (struct node *) malloc
          (sizeof(struct node));
newNode -> data = 4;
newNode -> next = head;
head = newNode;
```

### ② Insert at the end

Steps to insert new node at the end :-

i) Allocate memory for new node.
ii) store data.
iii) Traverse to last node.
iv) change next of last node to recent created node.

(13)

```
struct node *newNode;
newNode = (struct node *) malloc
          (sizeof(struct node));
newNode -> data = 4;
newNode -> next = NULL;
struct node *temp = head;
while(temp -> next != NULL){
    temp = temp -> next;
}
temp -> next = newNode;
```

(3) Insert at specific position

Steps to insert new node at the specific position :-

i) allocate memory & store data for new node.

ii) traverse to node just before the required position of new node.

iii) change next pointers to include new node in between.

```
struct node *newNode;
newNode = (struct node *) malloc
          (sizeof(struct node));
newNode -> data = 4;
struct node *temp = head;
for(int i = 2; i < position; i++){
    if(temp -> next != NULL){
        temp = temp -> next;
    }
}
newNode -> next = temp -> next;
temp -> next = newNode;
```

(14)

* Delete elements from a linked list.
we can delete element either from the beginning, from the end or from the specified position.

① Delete from the beginning
Steps to delete node from the beginning :-
i) simply point head to the second node.

```
head = head -> next;
```

② Delete from the end
Steps to delete node from the end :-
i) traverse to second last element.
ii) change its next pointer to null.

```
struct node *temp = head;
while (temp -> next -> next != NULL)
{
    temp = temp -> next;
}
temp -> next = NULL;
```

## ③ Delete from specific position.

Steps to delete node from position :-

i) traverse to element before the element to be deleted.

ii) change next pointer to exclude the node from the chain.

```
for(int i = 2; i < position; i++){
    if(temp -> next != NULL){
        temp = temp -> next;
    }
}
temp -> next = temp -> next -> next;
```

* Menu driven program for create, insert & delete linked list.

(16)

```
#include<stdio.h>
#include<stdlib.h>
struct node{
    int info;
    struct node *linR;
};
struct node *start = NULL;
void createlist(){
    if(start == NULL){
        int n;
        printf("enter the no. of nodes
            you want: ");
        scanf("%d", &n);
        if(n != 0){
            int data;
            struct node *newnode;
            struct node *temp;
            newnode = (struct node *)
             malloc(sizeof(struct node));
            start = newnode;
            temp = start;
            printf("enter number to be
                inserted = ");
            scanf("%d", &data);
            start -> info = data;
            for(int i = 2; i <= n; i++){
                newnode = (struct node *)
                 malloc(sizeof(struct
                 node));
                temp -> linR = newnode;
```

```
            printf("enter no. to be inserted");
            scanf("%d", &data);
            newnode -> info = data;
            temp = temp -> link;
        }
    }
    printf("\n the list is created");
}
else{
    printf("\n the list is already create
                -d");
}
}
void traverse(){
    struct node *temp;
    if(start == NULL){
        printf("\n List is empty");
    }
    else{
        temp = start;
        while(temp != NULL){
            printf("Data = %d -> ", temp ->
                                    info);
            temp = temp -> link;
        }
    }
}
void insertAtFront(){
    int data;;
    struct node *temp;
    temp = (struct node *) malloc(sizeof
                                (struct node));
    printf("enter number to be inserted");
    scanf("%d", &data);
```

```
    temp -> info = data;
    temp -> link = start;
    start = temp;
}
void insertAtEnd(){
    int data;
    struct node *temp, *head;
    temp = (struct node *) malloc (sizeof
                    (struct node));
    printf("enter number to be insert");
    scanf("%d", &data);
    temp -> info = data;
    temp -> link = null;
    head = start;
    while(head -> link != NULL) {
        head = head -> link;
    }
    head -> link = temp;
}
void insertAtPosition(){
    struct node *temp, *newnode;
    int pos, data, i = 1;
    newnode = (struct node *) malloc
            (sizeof (struct node));
    printf("enter position & data = ");
    scanf("%d %d", &pos, &data);
    temp = start;
    newnode -> info = data;
    newnode -> link = 0;
    while (i < pos - 1) {
        temp = temp -> link;
        i++;
    }
    newnode -> link = temp -> link;
```

```
        temp -> link = newnode;
    }
void deleteFirst(){
    struct node *temp;
    if(start == NULL){
        printf("list is empty");
    }
    else{
        temp = start;
        start = start -> link;
        free(temp);
    }
}
void deleteEnd(){
    struct node *temp, *pevnode;
    if(start == NULL){
        printf("list is empty");
    }
    else{
        temp = start;
        while(temp -> link != 0){
            pevnode = temp;
            temp = temp -> link;
        }
        free(temp);
        pevnode -> link = 0;
    }
}
void deleteposition(){
    struct node *temp, *position;
    int i=1, pos;
    if(start == NULL){
        printf("List is empty");
    }
```

```
        else{
            printf("enter index: ");
            scanf("%d",&pos);
            position = (struct node *)malloc
                    (sizeof(struct node));
            temp = start;
            while(i < pos-1){
                temp = temp -> link;
                i++;
            }
            position = temp -> link;
            temp -> link = position -> link;
            free(position);
        }
    }
int main(){
    int choice;
    while(1){
        printf("1.To see list ");
        printf("2.For insertion at first");
        printf("3.For insertion at end ");
        printf("4.For insertion at pos ");
        printf("5.For deletion at first ");
        printf("6.For deletion at end ");
        printf("7.For deletion at pos ");
        printf("8.To exit ");
        printf("\nenter choice:");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                traverse();
                break;
            case 2:
                insertAtFront();
                break;
```

```
            case 3:
                insertAtEnd();
                break;
            case 4:
                insertAtPosition();
                break;
            case 5:
                deleteFirst();
                break;
            case 6:
                deleteEnd();
                break;
            case 7:
                deletePosition();
                break;
            case 8:
                exit(1);
                break;
            default:
                printf("invalid choice");
        }
    }
    return 0;
}
```

(22) (Infix vs. Postfix vs. Prefix)

Quest. convert p-q-r/a to prefix & postfix.

((p-q)-(r/a))

((-pq)-(/ra))

--pq/ra → Prefix Expression

((p-q)-(r/a))

((pq-)-(ra/))

pq-ra/- → Postfix Expression