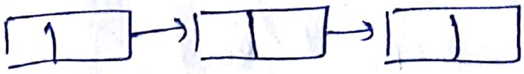
Module-3

Link List:

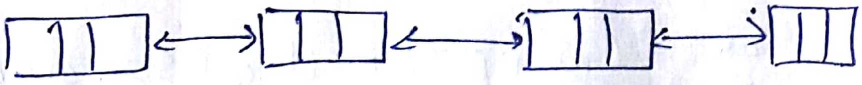
Link List: Lists are special data types in which data elements are linked to each other. The logical ordering is represented by having each element pointing to next element. Each element is called a node, which has two parts DATA or INFO: in which info. is saved and POINTER which points to next element.

Types:

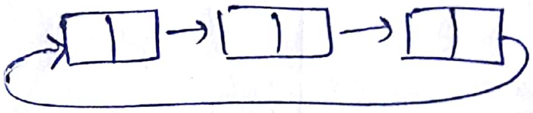
→ Single :



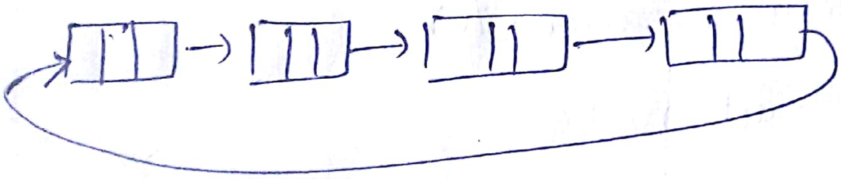
→ Double :



→ Circular



→ Doubly Circular.



List ADT:

Implementation of list using Array and Pointers:

→ Representation of Linear link List:

```
    struct node
    {
      int   a;
      struct node *next;        // ~~struct~~ pointer to node.
      };
      typedef struct node NODE;  // Type definition make
                                     it abstract.
      NODE *start;
                               // Pointer to the node of
start = (NODE *) malloc (size of(NODE)). linked list.
```

Program:

```
        #
        #
        # include <alloc.h>
          struct node
          {
            int data;
          struct node * next;
          }; *Start=NULL;
          void main()
        {
        struct node *nw, *start;
           int i,n;
        clrscr();
       start=0;
     pf("enter size of list");
      sf("%d",&n);
```

```
  for (i=0;i<n;i++)
  { pf("enter element");
 nw=(struct node *)malloc
  (sizeof(struct node));
   sf ("%d", &(nw->data));
     nw->next=head;
 } head=nw;
```

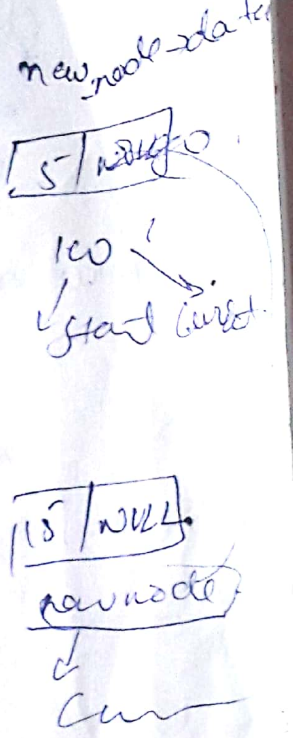
~~w=head;~~

```
#
#
#include <alloc.h>
struct node
 { int data;
 struct node * next;
 }*start=NULL;
 void create()
  { char ch;
 do
 {
struct node *newnode,*current;
new_node=(struct node*)
malloc( sizeof(structnode));
```

```
P.f ("enter the data");
s.f ("%d", &new_node->data);
 new_node->next = NULL;
 if (start= =NULL)
  { start=new_node;
  current = new_node;
 }
  .else
{
current ->next=new_node;
current= new_node;
 }
P.f ("do you want to continue")
 ch= getche();
 } while(ch!='n');
 }
```

```
void display()
{
struct node *new_node;
P.f (" elements of list =");
new_node = start;
while (new_node!=NULL)
{P.f (" %d --->", new_node->data;
new_node = new_node ->next;
 }
  P.f ("NULL);
 }
```

```
Void main()
 {
  create();
  display();
 }
```



→ Inserting a Node At Beginning.

```
1. Start.
2. Check for overflow → if (newnode =NULL), then
                          print overflow and exit.
                       else.
   new_node =(struct node*)malloc (sizeof(struct node));
          end if.
3.     set   new_node [data ]= item.
4.     set   new node [next]= null.
5.       if (start=NULL)
           {
               Start = new _node )
               current = new_node;
         else.
         new_ node ->next -> start;
           start = new _node
6. end.
```

Inserting A Node At End of List:

```
1. Start.
2. Declare two pointer nodes as - newnode
     and cu
```

rogram to insert At End!

```
Void Insert_at-end()
{
 struct node *new-node, *current;
new_node = (struct node*)malloc
  (sizeof(structnode));
   if (new-node == NULL)
   pf (" failed toallocate memory");
    else.
 pf (" enter data ");
   sf ("%d", &new Node->data);
    new node->next = NULL;
   if (start==NULL)
   {   start = new-node;
             current = new-node;
        }
  else
    {  temp=start;
   while(temp->next != NULL)
        {temp = temp->next;
        }
    temp->next=new_node;
       }
   }.
```

Insertion of Node At Desired location

```
Void insertLoc
(struct node * start, int item,
    int after)
{
 node * new_node ~~next~~, *loc;
 loc = search (start, after)
 if (loc==(start *) NULL)
     return;
new-node=(node *)malloc
  (sizeof (node));
 new_node->data = item,
new-node->next = loc->next
 loc->next = new_node;
}.
```

→ Deletion of Nodes:

1. Delete A Node from Beginning: (first

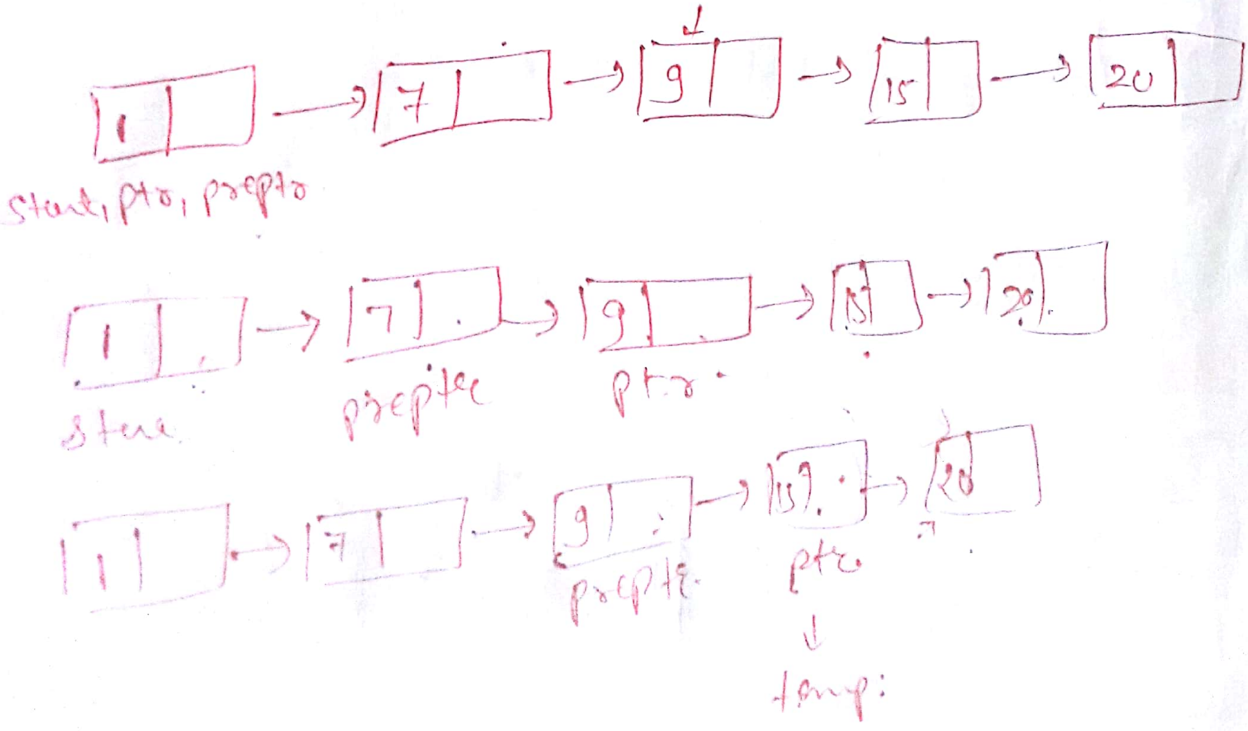
```
→ if start= NULL, then
   write underflow and return.
→ set ptr= start.
→ set start = start → Next.
→ free (ptr)
→ Exit
```

2. Delete Last Node.

```
1. → if start=NULL, then
      write UNDERFLOW. and return
2. → set Ptr = start.
3. → Repeat step 4 and 5 while PTR → Next,!= NULL
4. →    set preptr = ptr.
5. →    set ptr = ptr → next.
        end of loop.
6.   set preptr → next = NULL.
7.   free (ptr)
8.   Exit.
```

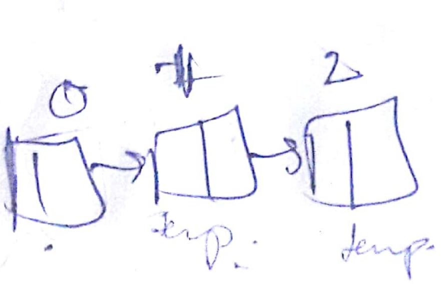
Deletion of Node After A Specified Location:

```
1. If start = NULL, then
       write underflow and exit.
2. set ptr = start.
3.     preptr = ptr.
4. repeat steps 5 and 6 while preptr->Data
                                      = NUM.
5.     set preptr = ptr.
6.     ptr = ptr -> next.
       (end of loop)
7. Set temp = ptr -> next
8. preptr -> next = Temp -> next
9. free(temp)
10. exit.
```



Insertion At Desired Location:

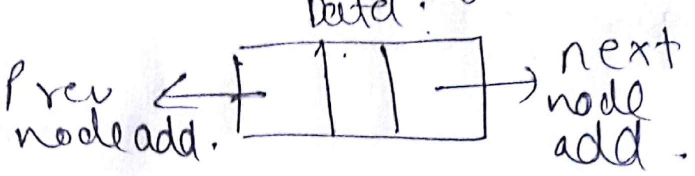
```
1.(Check for overflow?) if new_node = NULL, then
                        Print overflow and return.

   else
      new_node = assign memory using malloc function
      end of if.

2: Set  new_node -> data = item.
3:   if  start = NULL  then
              Set  start = new_node.
                   new_node -> next = NULL.
      end of if.
4. Initialise counter(I) and pointers. (Node *temp)
        set I = 0 , temp = Start
5.  Repeat step 6 & 7. until i < loc
    Set temp = temp -> next.                      i < 2
    . set i = i + 1
8  set new_node -> next = temp -> next.
9.  set temp -> next = new_node
10.  End.
```



ble link list:

5.

One of the most striking disadvantages of single and circular link list is the inability to traverse the list in backward direction. In most of the real world applications it is necessary to traverse the list in both directions. The most appropriate data structure for such an application is doubly link list.



Implementation:

```
Struct node
{ int num;
  struct node *prev;
  struct node *next;
}
```

* THE PREVIOUS field of first node and NEXT field of Last Node contains NULL.

Insertion

At Beginning

```
1. Create a new_node using
   malloc function.
2. Set new_node[data]= item
3.     new_node[prev]= NULL
4.  if (start == NULL)
      a) newnode→next=NULL
      (b) ~~new~~ start= newnode.
5. else.
      (a) newnode[next]=Start
(5) Start= newnode.
```

Insertion At End:

```
1. Create a newnode using malloc
   function and assign value
   to it.
2. Set newnodo [data] = item
3. Set newnode [next] = null.
4. if (start == NULL).
   { a) set newnode.[prev] = null
     b) Start = newnode.
5. Else.
   temp = start.
5. while (temp->next != NULL)
     temp = temp->next
   temp [next] = newnode.
6. temp [next] = newnode.
   newnode[prev] = temp.
7. newnode[prev] = temp.
8. End.
```

Location'.

..plementation of Stack and Queues using Link List!

Stack: Push

```
→ allocate memory for new node
→ ~~Ser~~ Set new_node→data = val
→ if TOP = NUL.
      Set new_node→next = NULL
          TOP = new_node.
  Else
      set new_node→next = Top
          TOP = new node.
  end of if.
→ end.
```

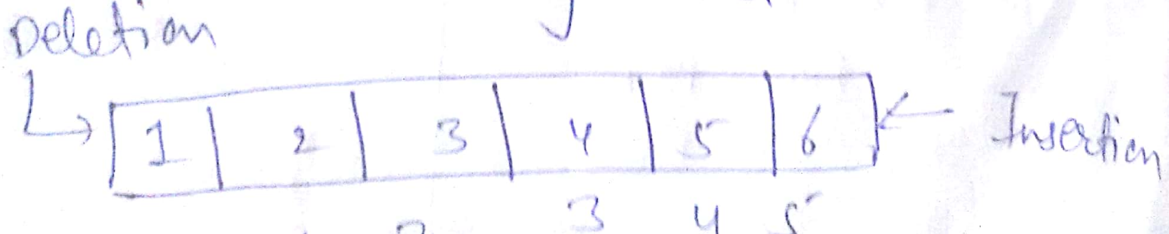
Pop

```
→ If TOP = NULL, then
      print overflow
  end of if.
- set PTR = TOP
- TOP = TOP→Next.
- free(ptr)
- End.
```

Queue: → It is a linear data structure which stores its elements in an ordered manner. First In First Out. (FIFO)

Representation → Using Array.

Representation → using Link List



Queue.

Using Array:

Insertion

```
1. If Rear= Max-1, then
      print overflow
        (end)
2.  if front==-1, and rear=-1,
    then set front=rear=0
        Else
3      set Rear= Rear+1
      (end of if)
3.     set EQ[Rear] = Num
4.        Exit
```

Deletion

```
1. If front = -1, then
      write underflow
   Else
      set front = front+1
      val = Q[front].
   end of if
2. Exit.
```

Using Link List:

Insertion

```
1. Allocate memory for newnode
   and name it as ptr.
2. set ptr->Data = Val.
3.   If front = NULL.
   Set front = Rear = PTR;
   Set front->next = Rear->next = NULL.
      Else.
         set . Rear->next = PTR
                  Rear = PTR
                  Rear->next = NULL
      end of if
4. End
```

Deletion

```
-> if front = NULL, then
   write underflow. Go to step3
-> set PTR = FRONT
-> Front = Front->next
-> free ptr.
-> End.
```