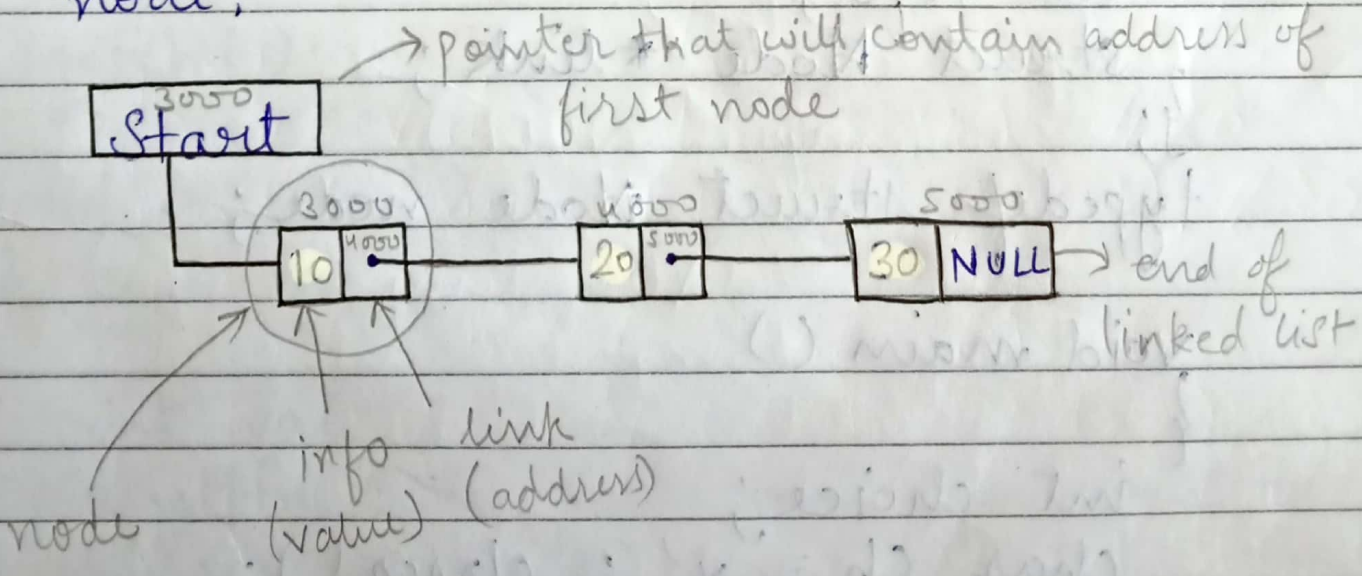
★ Linked List :

- Problems in Array :-
  (i) Static Memory Allocation
  (ii) difficulty in insertion and deletion

- Advantages in Linked List :-
  (i) Dynamic Memory Allocation
  (ii) Direct insertion & deletion

- Linked List is a collection nodes where each node will contain two parts, first part will contain value and second part will contain address of another node.



Note : It is difficult to maintain because of more & more nodes and it goes only in one

direction and there is no direct access in linked list.

- Construction of Node :-

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

datatype

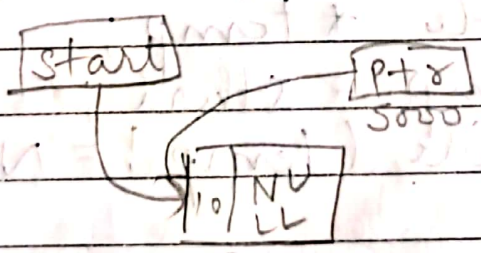
Ques WAP to create a linked list :

→

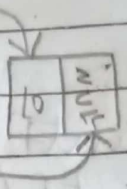
```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
void insert();
void display();
struct Node
{
   int info;
   struct Node *link;
};
typedef struct Node node;

void main()
{
   int choice;
   char ch = 'y'; clrscr();
   node *start = NULL, *ptr, *new;
   while (ch == 'y' || ch == 'Y')
```

```
{
printf("Enter 1 for create linked list\n");
printf("Enter 2 for display linked list\n");
printf("Enter 3 for exit\n");
printf("Enter your choice\n");
scanf("%d", &choice);
switch(choice)
{
case 1: insert();
        break;
case 2: display();
        break;
case 3: exit(1);
default: printf("Invalid choice");
}
flushall();
printf("Do you want to continue: Press
        y/Y for continue");
scanf("%c", &ch);
}
getch();
}
void insert()
{
int val;
printf("Enter value which you want to
  insert in node");
scanf("%d", &val);
ptr = (node *) malloc(sizeof(node));
```

datatype



```
ptr -> info = val;
ptr -> link = NULL;
if (start == NULL)
  {
    start = ptr;
  }
else
{
  nw = start;
  while (nw -> link != NULL)
    {
      nw = nw -> link;
    }
  nw -> link = ptr;
}
}
void display()
{
  node *temp;
  temp = start;
  while (temp != NULL)
  {
    printf("%d\t", temp -> info)
    temp = temp -> link;
  }
}
```



→ if ptr = NULL; then it disconnects the ptr address

malloc = garbage value;
calloc = initialize with 0

malloc syntax:
(datatype *) malloc (size of (datatype));

- Operations on Linked List :

(i) Insertion at the beginning.
(ii) Insertion at the end.
(iii) To Delete first node.
(iv) To Delete last node
(v) Searching (linked list)
(vi) To delete a particular node (by value)
(vii) To insert a new node after a particular value.
(viii) To reverse linked list.
(ix) To sort linked list.

1.)
```
void insert_beg (start)
{
    a = (node *) malloc (size of (node));
    printf (" Enter the value ");
    scanf (" %d ", &val);
    a → info = val;
    a → link = NULL;
    ptr → link = start;
    start = a;
    free (a);
}
```

2.)
```
void delete_first_node (start)
{
    node * temp;
    temp = start;
    start = start → link;
```

```
        temp -> link = NULL;
    printf("Deleted value is %d", temp->info);
        free(temp);
}
```

3.)
```
void delete()
{
    node *ptr, *nw;
    int val, del = 1;
    printf("Enter the number of element
            which you want to delete");
    scanf("%d", &val);
     nw = start;
    while (del != val+1)
     {
       nw = nw -> link;
        a = nw -> link;
       del++;
     }
    del = 1;
    nw = start;
   while (del != val)
   {
     nw = nw -> link;
     if (del == val-1)
       {
         nw -> link = a;
       }
     del++;
   }
}
```

val = 20

4.)
```
search_linked_list(start)
{
    node *ptr;
    int val, flag = 0; count = 0;
    printf("Enter the value which you
            want to search");
    scanf("%d", &val);
        ptr = start;
    while(ptr != NULL)
    {
        count++;
        if(ptr -> info == val)
        {
            flag = 1;
            break;
        }
        else
        {
            ptr = ptr -> link;
        }
    }
    if(flag == 0)
    {
        printf("Value not in linked list");
    }
    else
    {
        printf("Address of node = %d", ptr);
                (Position " " " , count);
    }
}
```

5.) void insertion_after_value (start)

```
{
  int val1, val2, flag = 0; count = 0;
  printf("Enter value which you want to insert");
  scanf("%d", &val1);
    nw = (node *) malloc(size of (node));
    nw -> info = val;
    nw -> link = NULL;
printf("Enter value after which you want
   to insert new node");
scanf("%d", &val2);
    ptr = start;
while (ptr != NULL)
  {   count ++;
    if (ptr -> info == val2)
      {
        flag = 1;
        break;
      }
  } ptr = ptr -> link;
if (flag == 0)
  {
    printf("Cannot insert");
    exit();
  }
else
  { nw -> link = ptr -> link;
    ptr -> link = nw;
  }
}
```

6.)
```
void delete_particular_node()
ptr1 = NULL;
ptr2 = start;
while (ptr2 != NULL)
{
  if (ptr2 -> info == val)
  {
    flag = 1;
    break;
  }
ptr1 = ptr2;
ptr2 = ptr2 -> link;
ptr1 -> link = ptr2 -> link;
  free (ptr1);
  free (ptr2);
}
```

7.)
```
reverse_linked_list (start)
{ node *prev, *next, *current;
  prev = NULL;
  next = NULL;
  current = start;
 while (current != NULL)
 { next = current -> link;
   current -> link = prev;
    prev = current;
    current = next;
 }
    start = prev;
}
```

12/10/19

* Stack:

Stack is a linear data structure in which insertion and deletions can take place on a single end, this end will be known as 'TOP STACK' works on LIFO principle. (Last in first out)

- Applications on Stack :-

i) Stack is used to implement recurrsive functions.

ii) Stack is used to solve arithmetic expression

Eg: A + B * C / d

iii) PUSH ( ) → It is a function which is used to insert a new elemekt in stack.

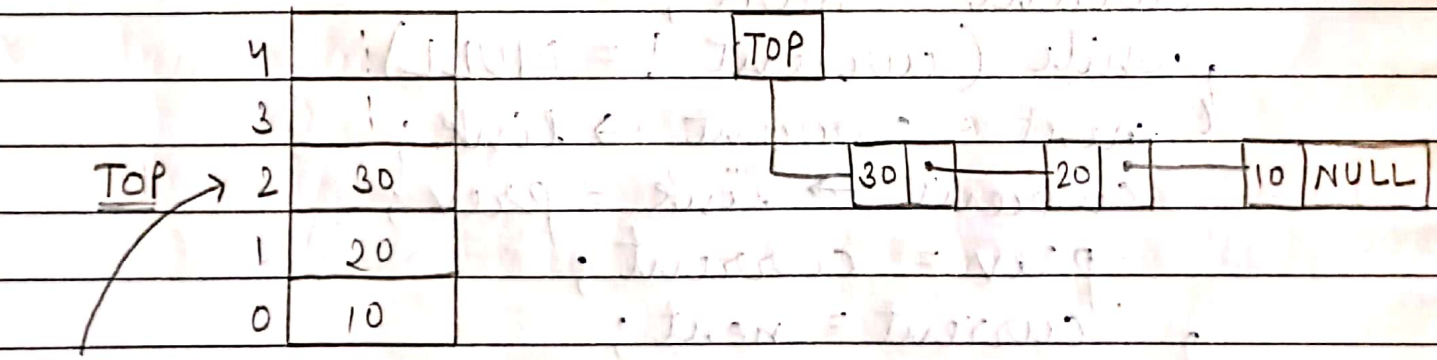
iv) POP ( ) → It is a function which is used to delete an element from stack.

v) Stack can be created by using :

a) Array b) Linked List

Eg:

- By using Array
- By using Linked List



TOP = 2

↳ It will contain the position of last inserted element in stack.

- Stack by using Array :-
  i) Underflow → deletion (empty)
  ii) Overflow → insertion (full)

Algorithm :

PUSH (Stack, N, Val)

// here n = maximum number of element in stack

Step 1 : if TOP == N-1
then
write "Overflow and Exit" ;

Step 2 : else
Set TOP = TOP + 1 ;
STACK [TOP] = VAL ;

Step 3 : exit ( ) ;

Algorithm :

POP (Stack , N , Val)

Step 1 : if TOP == -1
then
write "underflow and exit" ;

Step 2 : else
Set VAL = STACK [TOP] ;
TOP = TOP - 1 ;

Step 3 : write "deleted value is VAL" ;

Step 4 : exit ( ) ;

- Stack by using Linked List :-

Algorithm :

PUSH (TOP, VAL)

Step 1: ptr = new node;
Step 2: set ptr → info = val;
ptr → link = NULL;
Step 3: set ptr → link = TOP;
TOP = ptr;
Step 4: ptr = NULL;
Step 5: free (ptr);
Step 6: exit ();

Algorithm:

POP (TOP, VAL)

Step 1: if TOP == NULL
then
write "Underflow and exit";
Step 2: else
ptr == NULL;
TOP = TOP → link;
Step 3: write "deleted value is ptr → info";
Step 4: exit ();

* Arithmatic Notations:
i) Infix expression (between the operands Eg: A+B)
ii) Postfix expression (after the operands Eg: AB+)
iii) Prefix expression (before the operands Eg: +AB)

- Operator Priorities :- (Imp.)
i) ( ) → Bracket
ii) ^ → caret means power

iii) *, /, % ... → priority is same (left to right)
iv) +, − → priority is same (left to right)

- Infix to Postfix Conversion Examples:-

i) A + [B * C] / D (Infix)

↓ (Postfix)

= A + [BC*] / D
= [A] + [BC*D/]
= ABC*D/+ (Postfix)

ii) (A + B * [(C/D)^E]) → Infix

↓ Conversion

= (A + B * [CD/] ^ E)
= (A + B * [CD/E^])
= ([A] + [BCD/E^*])
= ABCD/E^*+ → Postfix

- Infix to Prefix Conversion Examples:-

i) A + [B / C] * D / E (Infix)

↓ Conversion

= A + [/BC] * D / E
= A + [*/BC] D/E
= A + / * / BCDE
= +A / * / BCDE (prefix)

ii) $(A + B * (C / D) \hat{} E)$ → Infix

↓ Conversion

$= A + B * / CD \hat{} E$

$= A + B * \hat{} / CDE$

$= A + * B \hat{} / CDE$

$= + A * B \hat{} / CDE$ → prefix

- Infix to Postfix Conversion by using Stack :- (Imp)

$A + B * (C / D) \hat{} E$ → Infix

Symbol scanned	STACK	Postfix Expression
(	[(]	
A	"	A
+	[(|+]	A
B	"	A B
*	[(|+|*]	A B
(	[(|+|*|(]	A B
C	"	A B C
/	[(|+|*|(|/]	A B C
D	"	A B C D
)	[(|+|*]	A B C D /
^	[(|+|*|^]	A B C D /
E	"	A B C D / E
)	—	A B C D / E ^ * +

↓ postfix

- Rules or Algorithm :-

Step 1: Put left paranthesis into the left side of infix exp" & right paranthesis to the right side of infix exp"

Step 2: Scan infix expression from left to right and repeat step 3 to 6 for each element of infix expression.

Step 3: If an operand encounter then, add it into postfix expression.

Step 4: If left paranthesis is encounter then, push it into stack.

Step 5: If an operator is encounter then,
a) repeatly POP from stack and add it into the postfix expression, if the operator which is on the top of stack have same priority or higher priority.
b) add this operator into stack

Step 6: If a right paranthesis is encounter then repeatly POP from the stack and add it into postfix expression until a left paranthesis is encounter, also remove the left paranthesis from the stack.

Questions : (Imp.)

a) (A - B) * (D / E)

b) (A + B ^ D) / (E - F) + G

c) A + (B + C - (D / E ^ F) * G) * H

⇓ conversion

Sol c): = A + (B + C - DEF^/ * G) * H

= A + (B + C - DEF^/G*) * H

= A + (BC + - DEF^/G*) * H

= A + BC + DEF^/G*- * H

= A + BC + DEF^/G*-H*

= ABC + DEF^/G*-H* +

Sol c.) By using Stack

Symbol scanned	Stack	Postfix Expression
(	(	
A	(	A
+	(+	A
(	(+ (	A B
B	(+ (	A B
+	(+ (+	A B C
C	(+ (+	A B C +
-	(+ (-	A B C +
(	(+ (- (	A B C + D
D	(+ (- (	A B C + D
/	(+ (- (/	A B C + D E
E	(+ (- (/	A B C + D E
^	(+ (- (/ ^	A B C + D E
F	(+ (- (/ ^	A B C + D E F
)	(+ (-	A B C + D E F ^ /
*	(+ (- *	A B C + D E F ^ /
G	(+ (- *	A B C + D E F ^ / G
)	(+	A B C + D E F ^ / G * -
*	(+ *	A B C + D E F ^ / G * -
H	(+ *	A B C + D E F ^ / G * - H
)	-	A B C + D E F ^ / G * - H * +

Algorithm to evaluate postfix expression

A * B + C - D → Infix

AB * + C - D
AB * C + - D
AB * C + D - → postfix

here,
A = 2, B = 3, C = 4, D = 5
put in the above equation
2 × 3 + 4 - 5
= 5

Ex: [TOP - 1] operator [TOP]
AB * C + D -) → Equation

Symbol	STACK
A	2
B	2 | 3
*	6
C	6 | 4
+	10
D	10 | 5
-	5

- Algorithm :-

Step 1: Add right paranthesis to the right side of the postfix expression.
Step 2: Scan postfix exp^n from left to right.
Step 3: If an operand is encounter then push this operand into stack.
Step 4: If any operator is encounter then pop top 2 elements from stack and perform following

operation:
result = [TOP-1] operator [TOP]

Step 5: Push this result into stack.

Step 6: Repeat above 2 steps until a right paranthesis is encounter.

Step 7: If a right paranthesis is encounter then pop top elements from stack and treat as a result.

- Operations on linked list:
  - ↳ To sort a singly linked list

8.)
```
void sort_linked list ( )
{
    node *i, *j;
    int temp;
    for (i = start ; i -> link != NULL ; i = i -> link)
    {
        for (j = i -> link ; j != NULL ; j = j -> link)
        {
            if ( i -> info > j -> info)
            {
                temp = i -> info;
                i -> info = j -> info;
                j -> info = temp;
            }
        }
    }
```

15/10/19

* Queue :
It is linear data structure in which insertion and deletion are performed on different ends. These ends are known as FRONT and REAR.

- FRONT is used to store position of first element (oldest element) in queue.
- REAR is used to store position of last element in queue.
- Queue works on FIFO principle (First In First Out).
- Queue can be implemented by using Array and linked list.

* Application of Queue :-
Queue is used to implement time sharing operating system.

* Types of Queue :-
i) Simple Queue
ii) Circular Queue
iii) DE-Queue (double ended)
iv) Priority Queue

"SIMPLE QUEUE"

* Queue using Array :-

i) Algorithm

Insertion_queue (front, rear, val, size, queue[ ])

Step 1: if rear == size - 1
then
print "overflow" and exit

Step 2: if front == rear = -1
then
set front = rear = 0

Step 3: else
Set rear = rear + 1

Step 4: set queue [rear] = val;

Step 5: exit

ii) Algorithm

Deletion_Queue (queue, front, rear, val, size)

Step 1: if front == -1
then
print "underflow" and exit

Step 2: else
set val = queue [front];

Step 3: if front == rear
then
set front = rear = -1

Step 4: else
front = front + 1;

Step 5: print "deleted value is val";

Step 6: exit

i) Algorithm

Insertion_Circular_queue (queue, front, rear, val, size)

Step 1: if (front == 0 && rear == size - 1)
||
(front == rear + 1)
then
print "Overflow" and exit

Step 2: if (rear == size - 1)
then
set rear = 0

Step 3: if (front = rear = -1)
then
set front = rear = 0

Step 4: else
rear = rear + 1

Step 5: set queue [rear] = val;

Step 6: exit

ii) Algorithm

Deletion - circular - queue (front, rear, val, size, queue)

Step 1: if (front == -1)
then
print "underflow" and exit

Step 2: else
val = queue [front].

Step 3: if (front == rear)
then
set front = rear = -1

Step 4: if (front == size - 1)
then
set from = 0

Step 5: else
front = front + 1

Step 6: print "deleted value is val";

Step 7: exit

"CIRCULAR QUEUE"

* Queue using Linked List :-

i) Algorithm

Insertion-queue (front, rear, val)

Step 1: Set ptr = (node *) malloc (sizeof (node))

Step 2: if (ptr == NULL)
then
write "overflow" and exit

Step 3: else
set ptr → info = val
ptr → link = NULL

Step 4: if (rear == NULL)
then
set front = rear = ptr

Step 5: else
rear → link = ptr
rear = ptr

Step 6: free (ptr)

Step 7: exit

★ "DE-Queue" (Double Ended Queue):
In it, insertion and deletion can be performed on both ends (front and rear).
There is two types of DE-Queue:
i) Input Restricted DE-Queue → at front (insertion), at rear
ii) Output Restricted DE-Queue → at front, at rear (deletion)

- Input Restricted DE-Queue → In it, insertion will take place on a single end but deletion can be performed on both ends.
- Output Restricted DE-Queue → In it, deletion will take place on a single end but insertion can be performed on both ends.

Note: insertion at front:

```
front = front - 1
Queue (front) = val
```

deletion at rear:

```
val = Queue (rear)
rear = rear - 1
```

★ "Priority Queue"
It is a special queue in which each value/node also contain one additional information i.e. known as priority of node/value. Eg: Operating System (shut-down)

- The node/value which have higher priority will used first.
- If two node/value have same priority then, we will use FIFO principle.