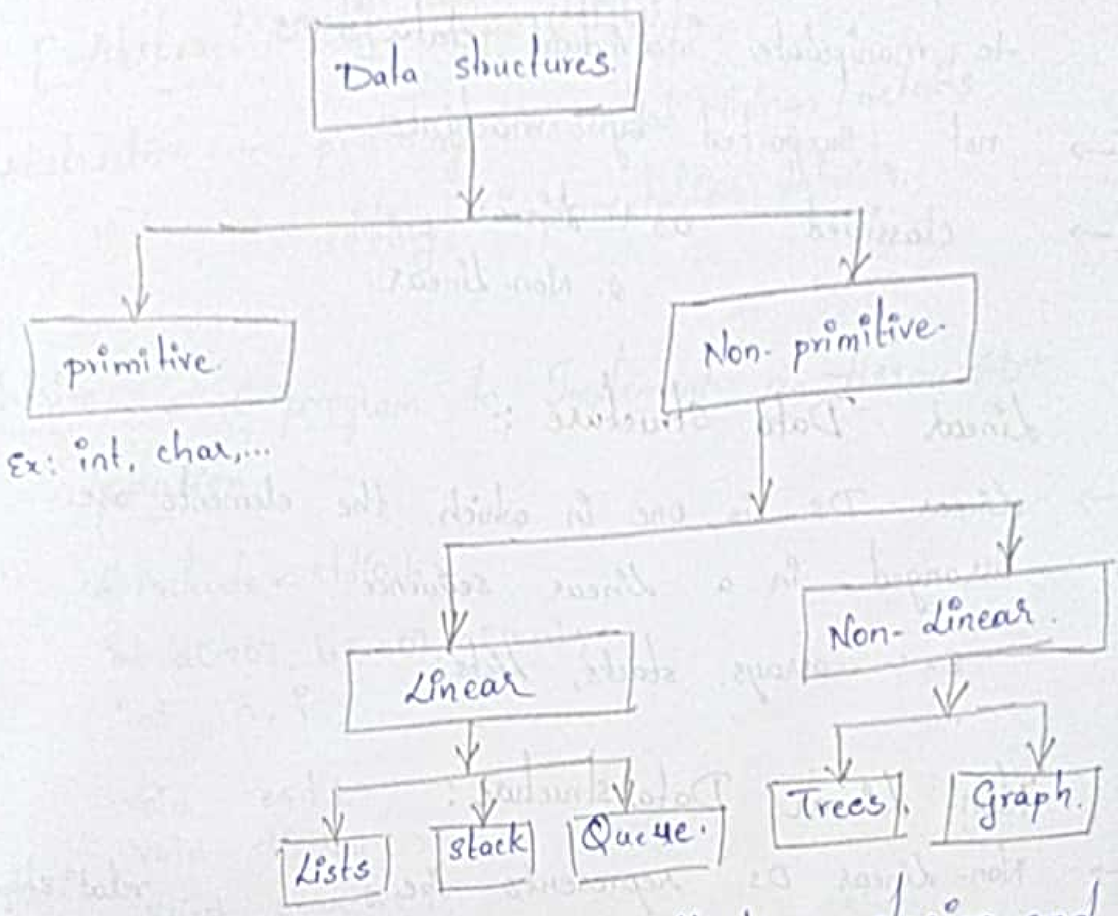
1/1/19.

Data Structures



* Data structure - It is a method of storing and Organizing the data in the memory Efficiently.

Data structure = Data elements + Operations + Algorithms.

* It is the study of different methods Operations and designing algorithms for the given Operations.

* primitive :-

→ are those which can be directly supported by the machine (without any code or program)

→ All primary data types come under this category.

Ex:- int, float etc.

Non-Primitive Data structures :-

→ Non-primitive Data structures has no specific instruction to manipulate individual data items

→ not supported by machines

→ classified as 1. Linear
2. Non-Linear.

Linear Data structure :-

→ Linear Ds is one in which the elements are arranged in a linear sequence.

Ex:- arrays, stacks, Lists.

Non-Linear Data structure :

→ Non-Linear Ds represents the non-Linear relat^n ship (non-sequencial) between the elements.

Ex:- trees & Graphs.

* Common Operations performed are :

1. Insertion
2. Deletion
3. Searching
4. Sorting
5. traversing.

18/7

Data Abstraction

[Abstract Data Type (ADT)]

- Abstraction - providing essential things / details
  hiding implementation details.

* Write a C program to Implement an Array ADT Operation.

```
#include <stdio.h>
int a[100], b[100], c[100];
int n, i;

void read();
void display();
void search();
void insert();
void reverse();
void merge();
void main()
{ int ch;
  while (1)
  {
    printf("\n Array ADT Operations are: ");
    printf("\n 1. read \n 2. display \n 3. search \n 4.
          insert \n 5. reverse \n 6. merge \n 7. exit");
    printf(" Enter your choice : \n");
    scanf("%d", &ch);
        switch(ch)
        {  case 1: read(); break;
           case 2: display(); break;
```

```
case 3:   search(); break;
case 4:   insert(); break;
case 5:   reverse(); break;
case 6:   merge(); break;
case 7:   exit(0); break;
default:  printf("Invalid choice\n");
        }
      }
    }

void read()
{
   printf("Enter n:\n");
   scanf("%d", &n);
   printf("Enter array: \n");
   scanf(  for(i=0; i<n; i++)
             scanf("%d", &a[i]);
}

void display()
{
   read();
   printf("The array Elements are: \n");
   for(i=0; i<n; i++)
     printf("%d", a[i]);
}
```

```
void search ( )
 int element, i, flag=0;
{ read ( );

 printf(" Enter Element to be searched: \n");

 scanf(" %d", & element");

 for(i=0 ; i<n; i++)

 {       if (a[i]==element)
            { pf("\n element found at %d", i);
                flag = 1;
            }
     }
     if (flag==0)
        pf("\n element not found");

void insert ( )

{  int element, index ;
   pf("\n Enter index");
   scanf("%d", & index);
  pf("\n enter value:");
  scanf ( " %d", &element);
for(i=n; i>index ; i--)
   a[i] = a[i-1]
a[index] = element;                n++;
 { for(i=0 ; i<=n ; i++)} (or)  display ( );
 { printf (" %d", a[i]); }

}
```

```
void reverse ( )
{   int t;
  for(i=0; i<n/2 ; i++)
  {     t= a[i] ;
      a[i] = a[n-i-1];
       a[n-i-1] = t ;
   }
   display( );
 }.

void merge ( )
{
  int i, j, m ;
  pf(" Enter second array: \n");
  scanf( "%d",&m);
  pf("\n Enter array: ");
  for(i=0 ; i<m; i++)
   scanf ("%d", & b[i]);
   for(i=0 ; i<n ; i++)
       c[i] = a[i] ;
     for (j=0 ; j<m ; j++ )
     { c[i] = b[j];
          i++;
     }
```

pf("After merging : \n");
for(i=0 ; i<n+m; i++)
pf("%d", c[i]);
}.

ADT :

Data ~~abstraction~~ ob abstraction means Representing only essential features by hiding all implementation Details.

Abstract Data type :-

* It is a specification of set of data and set of operations that can be performed on the given data.
* Every data structure has to ADT.

* The concept of abstraction means
1. we know what a data type does. (or a function does)
2. How it is done is hidden.

* Abstract data Type consists
- Declaration of data. (objects / instances)
- Declaration of Operations (functions).

Example :-

Array ADT.

• objects :-
 int a[ ], b[ ], c[ ];
 • functions :-
 void read ();
 void display();
 void merge();
 ⋮

Stack using Arrays.

A stack is a linear list in which all additions and deletions are restricted to one end, called "top."

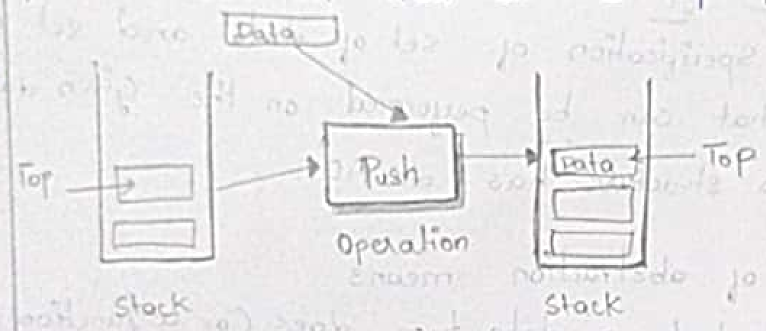
Principle of stack : Last-in-first-out
(or)
first-in-Last-out.

Basic Stack Operations :-

- Basic stack operations are push(), pop(), stack top()

- 1. Push()

push() adds an item at the top of the stack.



→

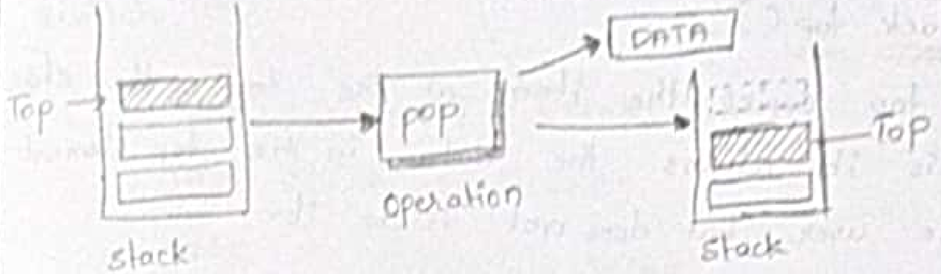
Stack ADT

object(-)
int stack[];
function :-
void push();
void pop();
void display();

2. Pop()

- when we use pop() on a stack, we remove the item at the top of the stack and return it to user.

- As we have removed the top item, the preceding item becomes the top of the stack.



Top :-

It always shows the current position of the stack.

Overflow:-

whenever we want to insert an element in stack, which is already full, then the overflow condition occurs.

```
if ( top > max-1)
pf(" stack is full\n");
```

underflow :-

It occurs when we try to delete an element from the stack which is empty.

```
if (top == -1)
pf("stack is empty");
```

Applications of the stack:-

1. Converting infix expression to postfix expression.
   ex :- (a+b) ⟶ (ab+)
2. postfix expression evaluation.
3. Implementing Recursion.
4. Multi-tasking
5. Scheduling -Algorithms.

Program :-

```
#include<stdio.h>
#include<stdlib.h>
#define max 50 ;
int top=-1;
void push();
void pop();
void display();
int main()
{
    int ch ;
    while (1)
    {
        pf ("\n stack Operation:");
        pf("\n 1: push \n 2. pop \n 3. display
            \n 4. exit");
        pf("\n enter choice :");
        sf("%d", &ch);
```

Continue →

```
switch (ch)
{
      case 1: push (); break;
      case 2: pop (); break;
      case 3: display (); break;
      case 4:   exit(1);
   }
}
}
void push ()
{   int element;
    pf("\n enter element");
    sf ("%d", & element);
    if ( top > max-1)
       pf("stack is full");
    else
     {   top= top+1;
         stack[top] = element;
     }
 }
 void pop ()
{  if (top==-1)
   pf("\n stack is empty");
  else
  { pf("\n deleted element: %d", stack[top]);
     top= top-1;
   }
}
```

```
Void display ( )
{  int i;
  if (top == -1)
  P.f ("\n stack is empty");
  else
 {   for (i= top ; i>=0 ; i--)
     Pf ("%d", stack [i]);
  }
}.
```

QUEUE

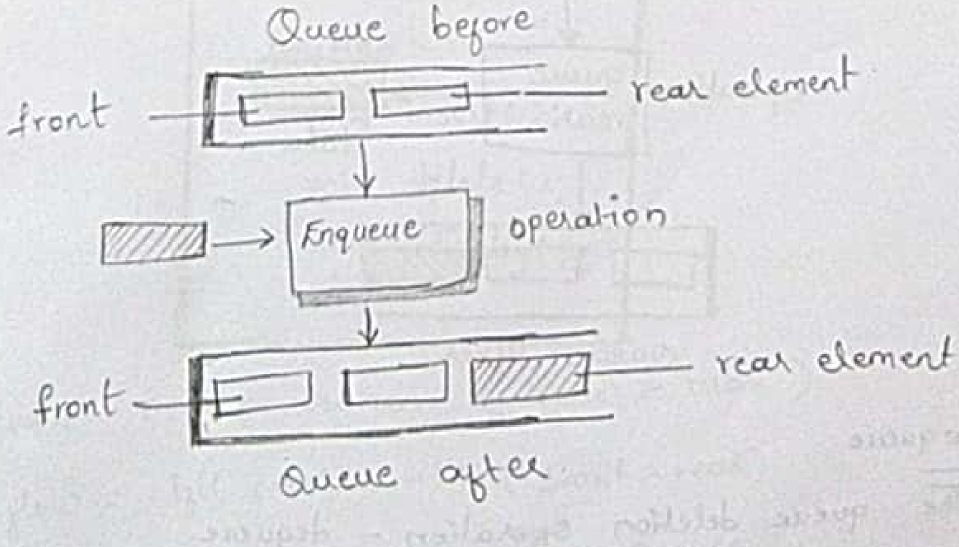
- A queue is a linear List in which data can be inserted at one end, called "rear".
- deleted from the other end, called the "front".
- These restrictions ensure that the data is processed through the queue, in order in which they are recieved.
- principle of Queue – first in - first out.

basic Operations :-

- four basic Operations are performed.
  1. Data, inserted at the rear
  2. deleted at the front
  3. data can be retrieved from rear end.
  4. data can also be retrieved from front end.

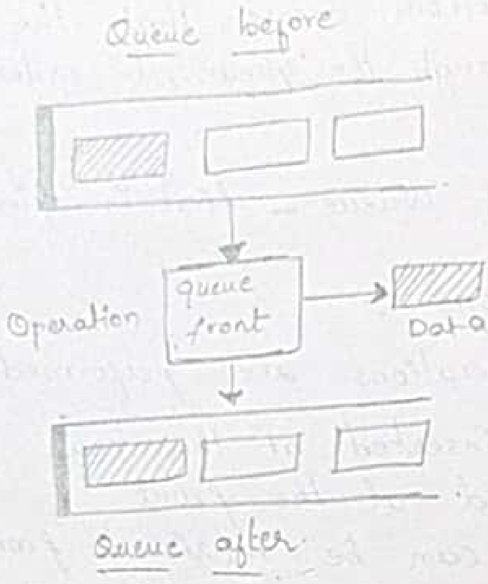
1. Enqueue.

- The queue insert operation is known as enqueue
- After the data have been inserted into the queue, the new element becomes the rear.
-



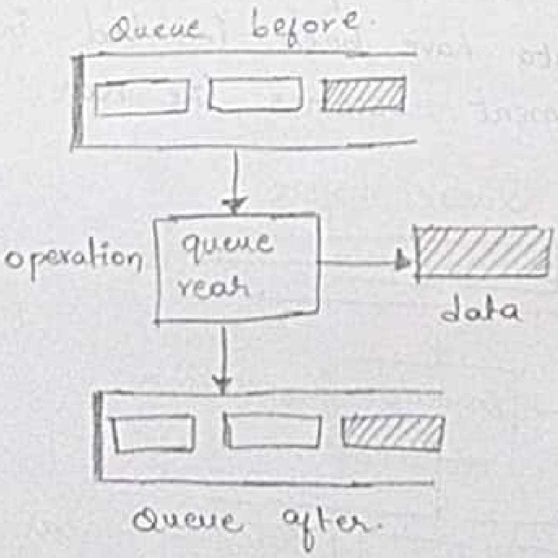
2. Dequeue front.

* Data at the front of the queue can be retrieved with queue front.
* It returns the data at the front of the queue without changing contents of the queue.



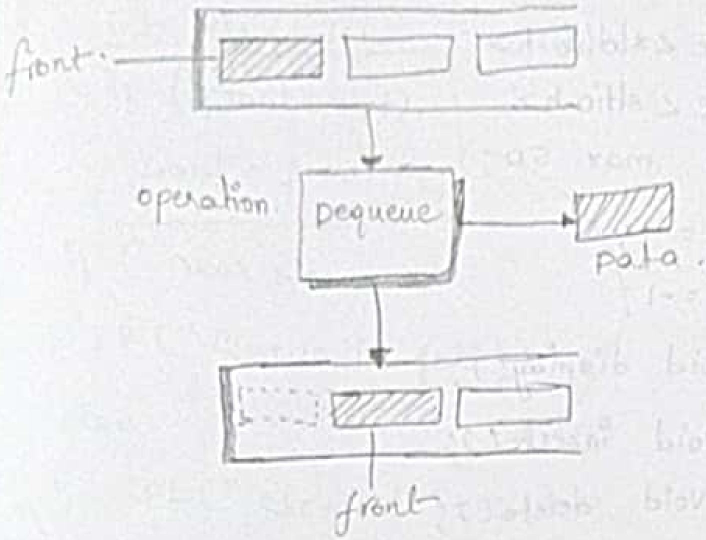
3. Queue rear.

- A parallel operation to queue front, but retrieves the data at the rear end of the Queue.



4. Dequeue

- The queue deletion operation – dequeue.
- The data at the front of the queue are returned to the user.
- If there is no data returned – queue is in underflow state



Overflow :-
when we try to insert an Element into the Queue, which is already full, then it is called overflow condition.

underflow :-
when we try to ~~inse~~ delete an element from a stack which is empty, it is called underflow condition

Queue ADT.

Object :
  int queue[];

function :-
  void insert();
  void delete();
  void display();
  :

overflow :- occurs when (rear ≥ max)
underflow :- if((front == -1) || (front > rear)
  underflow condition occurs.

Program :-

```
#include <stdlib.h>
#include <stdio.h>
# define max 50;
int rear=-1;
int front=-1;
        void display();
        void insert();
        void delete();
void main()
{
  int a[max]; int ch;
  while (1)
  {
      pf("\n Queue Operations");
      pf("\n 1. insert\n2. delete \n 3. display \n4. exit");
      pf("\n Enter your choice: ");
      sf("%d", &ch);
  Switch (ch)
  {
    case 1 : insert(); break;
    case 2 : delete(); break;
    case 3 : display(); break;

    case4 : exit(1)
  }
 }
}
```

```
void insert ( )
{ int element;
  if (front==-1)
    front= front+1;
  if ( rear>max-1)
   pf("Queue is full")
  else
  {  pf("element is : ");
     sf("%d", &element);
     rear= rear+1;
     a[rear] = element;
  }
}.
void delete ( )
{
   if ((front==-1)||(front>rear))
   pf(" queue is empty");
   else
  {  pf ("deleted element:%d", a[front]);
     front= front+1;
  }
}
void display ( )
   int i;
{  if((front==-1)||(front>rear))
   pf(" queue is empty");
   else
 { pf("Queue elements are: ");
   for(i=front; i<=rear; i++)
   pf(" %d\t", a[i]);
 }
}
```

Linked List.

Self-referential structure.

self-referential structures are those structures that have one or more pointers, which point to same type of structure as there member i.e., structures pointing to the same type of structers are self-referential structures.

ex:-

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

ex:-

```
struct student
{ int roll no;
  Char name[20];
  float avg;
  :
} struct no student *next;
```

program :-

```
// Demonstrating self-referential.

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

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

main ()
{  struct node *a, *b, *c, *p;

   a = (struct node *) malloc (sizeof (struct node));
   b = (struct node *) malloc (sizeof (struct node));
   c = (struct node *) malloc (sizeof (struct node));

   pf("\n enter data items:");
   sf ("%d %d %d", &a->data, &b->data, &c->data);

      a->next = b;
      b->next = c;
      c->next = NULL;

   pf("\n data elements are :");

   p = &a;
 while( p!=NULL)
 {  pf(" %d\t", p->data);
      p = p->next;
  }
}
```

Creating Linked List

function :-

```
void create ( )
{   int c;
    struct node *ptr, *cptr;
    ptr = (struct node *) malloc (size of (struct node));
    pf ("\n enter first node: ");
    sf ("%d", & ptr→data);
    head = ptr;
    pf ("\n do yo want more node (0/1): ");
    sf (" %d", &c);
    while (c==1)
    { cptr = (struct node *) malloc (size of (struct node));
      pf ("\n enter next node data: ");
      sf ("%d", &cptr→data);
      ptr→next = data cptr;
      ptr = cptr;
      pf ("\n do you want next node (0/1): ");
      sf ("%d", &c);
    }
    ptr→next = NULL;
}.
```

Scanned by CamScanner

Before main()
(Declarations, etc).

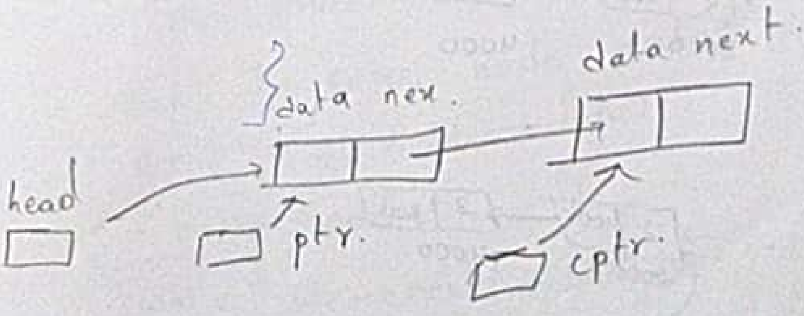
```
#include<stdio.h>
#include<stdlib.h>.

struct node
{
   ______ data;
struct node *next;
};
struct node *head = NULL;

main ()
{
      ch;
while(1)
{
```

Display menu.

1. Create ( );
2. display( );
3. insert begin();
4. insert.end();
5. insert.after();
6. delete.begin();
7. delete.end();
8. delete.after();
9. search ( );
10. exit(1);

```
Switch (ch)
{
      case 1 :- ______
      case 2 :- ______
      case 3 :- ______
```

menu itemes.



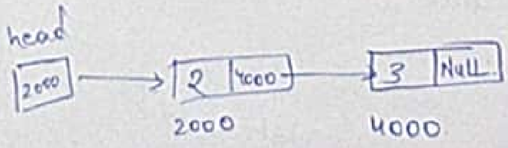
Display list :-

funct^n :-

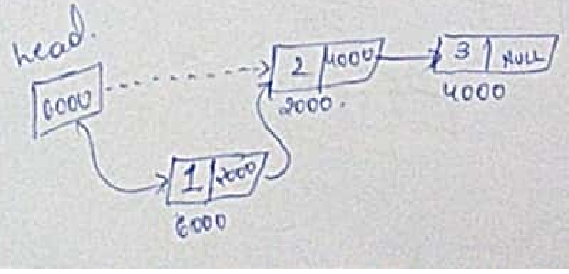
```
void display()
{
    struct node *p;
    if (head == NULL)
    pf("SLL is empty");

    else
    { p = head;
      pf("\n SLL nodes are: ");
      while (p != NULL)
      {
          pf("%d\n", p->data);
          p = p->next;
      }
    }
}
```

Inserting at begining node :-

Before :-



After.



void insert.begin()

{

struct node *ptr;

ptr = (struct node *)malloc (~~str~~ sizeof (struct node));

pf("\n enter node data:");

sf("%d", &ptr→data);

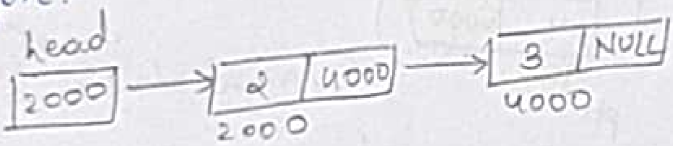
ptr → next = head;

head = ptr;

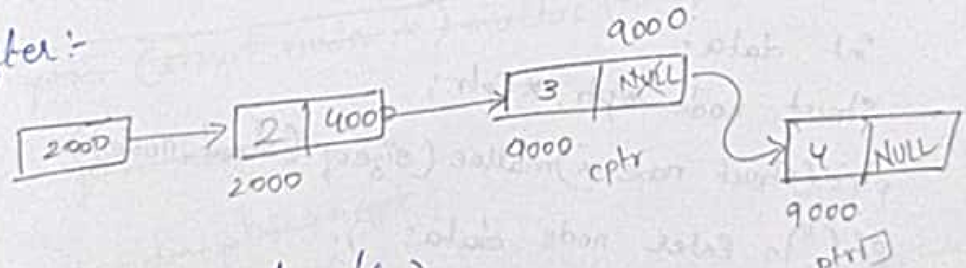
}.

Inserting at the End :-

before:-



After:-



void insert.end()

{

struct node *ptr, *cptr;

ptr= (struct node *) malloc (sizeof (struct node));

pf ("\n enter node data:");

sf ("%d", & ptr→ data);

cptr= head;

while (cptr→next! = NULL)

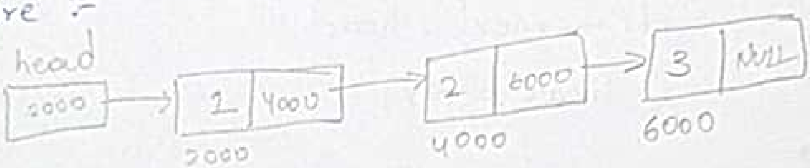
{ cptr= cptr→ next;

}

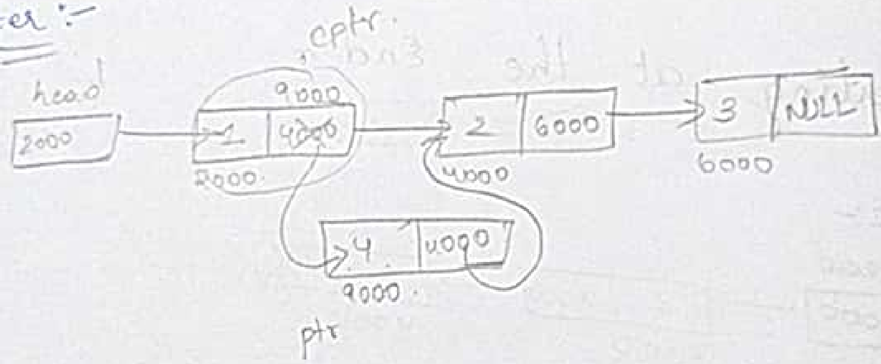
```
cptr -> next = ptr;
ptr -> next = NULL;
}.
```

Insertion After Given Node:

Before :-



After :-

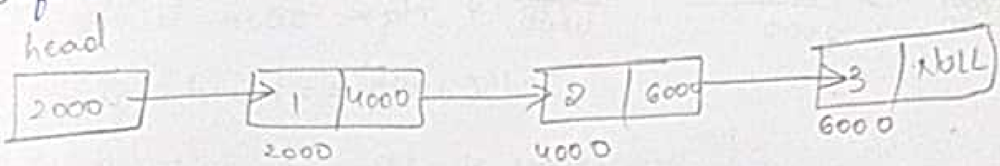



```
void insertAfter()
{
   int data;
   struct node *ptr, *cptr;
   ptr = (struct node *)malloc (sizeof (struct node));
   pf("\n Enter node data: ");
   sf("%d", &ptr->data)
      cptr = head;
   pf("\n enter node after which u want to insert:");
      sf("%d", &data);
   while (cptr->data != data)
   {
      cptr = cptr->next;
   }
```

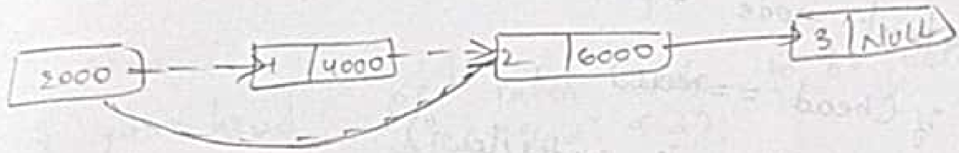
```
ptr->next= cptr->next
cptr->next= ptr;

}.
```

Delete at begin ( ).

Before.



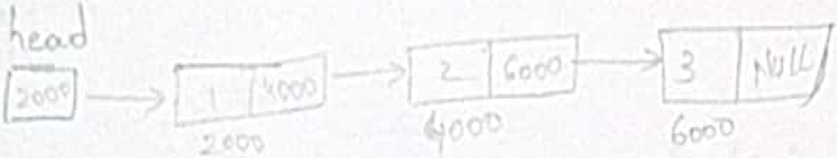
After:-



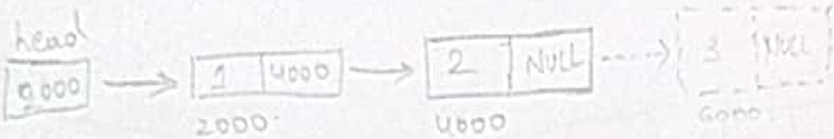

```
Void delete.begin( )
{   struct node *ptr;
    ~~ptr= (struct node *) malloc (sizeof (structnode));~~
    ~~ptr= head;~~
    ~~head= ptr->next;~~
    ~~free(ptr);~~
}.  if (head == NULL)
    pf("\n SLL is underflow");

    else
  { ptr= head;
    head= ptr-> next;
    free (ptr);
  }
}.
```

Delete at the End.

before :-



After :-

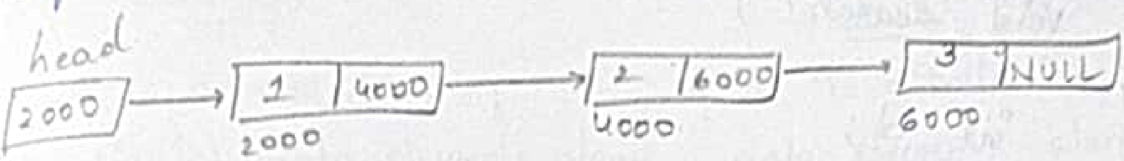



```
void delete.end()
{
  struct node *ptr, *cptr;
    if (head ==NULL)
    { pf("\n su is underflow");
    }
    else
    { ptr= head;
    while (ptr->next != NULL)
      { cptr=ptr;
        ptr= ptr->next;
      }
    Cptr->next= NULL;
    pf("deleted data is %d", ptr->data);
     free(cptr);
  }
}
```

// At the end of Loop last node name is ptr & before node name is cptr //

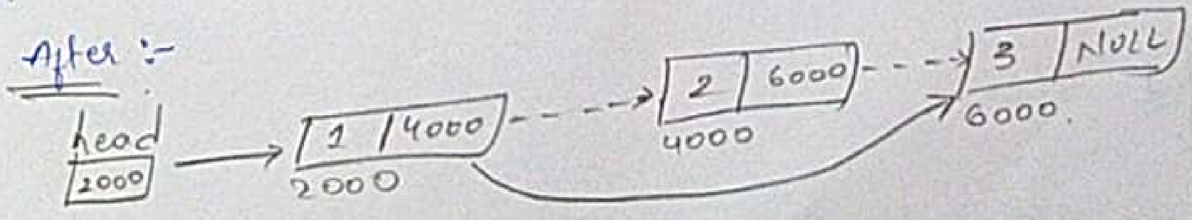
Deletion After Given Node :-

before :-




```
void delete.after( )
      int d;
{   struct node *ptr, *cptr;
     if (head == NULL)
     { pf ("\n SLL is underflow");
     }.
    else                            pf(" enter node to be deleted : ");
  { ptr= head ;               sf ("%d", &d);
     while (ptr→ data != d)
     { cptr= ptr ;
         ptr = ptr→next.
      }.
     cptr→next= ptr→next;
     pf(" deleted node is : %d", ptr→data);
     free(cptr);
  }.
}.
```

After :-



Searching a Node :-

```
void search( )
{   int d;
  struct node *ptr;
   int flag=0;
  if (head == NULL)
  pf("\n SLL is underflow");
  else
{   ptr= head;        pf("\n enter node to be searched:");
                      sf ("%d", &d);
   while ( ptr->next != NULL) )
   {
       if (ptr->data ==d)
          flag = 1;
       ptr=ptr->next;
    }
 }
 if ( flag == 1)
 pf("\n node found");
 else
 pf("\n node not found");
}
```

Important points :-

Arrays

1. It is a collection of similar data elements, stored in continuous memory locat^n.
2. Random access with the Index value.
3. Accessing is fast.
4. Insertion & deletion operat^n takes more time.
5. Fixed size

Linked Lists

1. It is a collect^n of similar data elements stored in different memory locat^n.
2. sequencial Access from first node.
3. Accessing is slow.
4. Insert^n & delet^n operat^n takes less time.
5. Dynamic size.

Linked List :-

- A Linked List is Linear collect^n of Data elements these data elements are called nodes. Every node contains 1 or more data fields & 1 or more pointers to the next node or previous node.
- Linked Lists are used for developing other data structures such as stacks, Queus & trees.
- Underflow - if (head == NULL), we get the underflow condition.

Scanned by CamScanner

Stack Using SLL

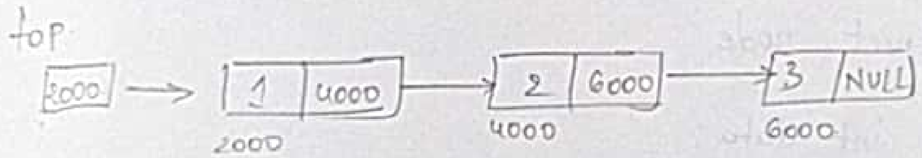
```
struct node
{  int data;
   struct node  *next;
}
int  main()
{   struct  node  *top==NULL;
    Display  menu.

  }
}
```

```
//push() (insertion begin())
 void push()
{
    struct  node  *ptr;
    ptr= (struct node *)malloc(sizeof(struct node));
    pf(" Enter data:");
    sf("%d", &ptr->data);

     ptr->next = top;
     top= ptr;
}
```

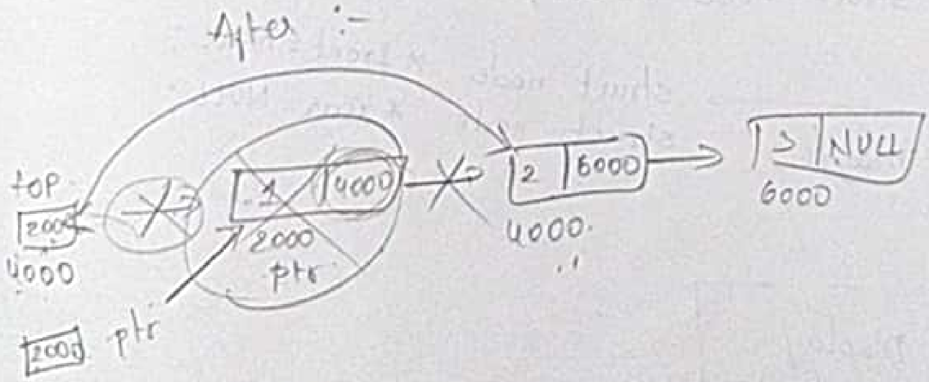
// pop() (deletion begin()) //

```
void pop()
{  struct node *ptr;
   if ( top == NULL)
   pf (" underflow condition \n");
   else
   {  ptr= top;
      top = ptr→next;
      free(ptr);
   }
}.
```

Before :-



After :-



// Display () //

```
void display ()
{
  struct node *ptr;
  if (top == NULL)
  pf (" under flow Condition \n");
```

```
else
{  ptr= top;
   pf("%d", ptr->data);
   pt while ( ptr != NULL )
      {
        pf ("%d \t", ptr->data);
         ptr= ptr->next;
      }
  }
```

Queue Using SLL.

```
struct node
 {  int data;
    struct node *next;
 };  ------------>  struct node *front = NULL;
int main()          struct node *rear = NULL;
{

   Display
    menu.

 }
}
```

01/2/19

```
void insert()
{  struct node *ptr;
   ptr= (struct node *)malloc(sizeof(struct node));
   pf(" enter node data: ");   sf("%d",&ptr->data);
   if(front == NULL)
   {   front = ptr;
       rear = ptr.
       rear->next = NULL;
   }
   else
   {  rear->next=ptr;
      rear= ptr ;
      rear->next= NULL;
   }
}.

void delete()
{ struct node *ptr;
  if((front == NULL)
  pf(" underflow condition \n");
                                  else if (front == rear)
                                  { ptr = front;
                                    pf("\n deleted: %d',
                                          ptr->data);
                                    free(ptr);
                                    front= NULL ; rear =NULL;
                                  }.
  else
  {  ptr= front;
     front= ptr->next;
     free (ptr) ;
  }.
}
```

Scanned by CamScanner

// display () //

```
void display ()
{  struct node *ptr;
   if(front==NULL)
   pf(" underflow condition");

   else
   { ptr= front;
     pf(" nodes are: ");
     while(ptr!=NULL)
     { pf(" %d \t", ptr->data);
       ptr= ptr->next;
     }
   }
}.
```

Circular Linked List

// Creating circular Linked List//

```
void create()
{  int c;
   struct node *ptr,*cptr;
   ptr= (struct node *)malloc(sizeof(struct node));
   pf("\n enter ___________
   ______________________ Same as SLL
   ______________________.

   ptr->next=head;

}.
```

// Display circular Linked List//

```
Void display()
{ struct node *p;
   if(head==NULL)
   pf("CLL is empty \n");
   else
   {   p= head;

      do
      { printf("%d\t", p->data);
        p=p->next;
      } while(p!=head)
   }
}
```

// void insertbegin() //

```
void insertbegin()
{
  struct node *ptr, *cptr;
  ptr= (struct node *) malloc (sizeof(struct node));
  pf("\n enter node data:");
  sf("%d", &ptr->data);
  cptr= head;
  while (cptr->next!=head)
  { cptr= cptr->next;
  }
  ptr->next= head;
  head = ptr;
  cptr->next= head;
}
```

// Insertend () //

```
void insertend ()
{
  struct node *ptr, *cptr;
  ptr= (struct node *)malloc(sizeof (struct node));
  pf("\n enter node data: ");
  sf("%d", &ptr->data);
  cptr= head;
  while(cptr->next !=head)
  { cptr= cptr->next;
  }
  cptr->next= ptr;
  ptr->next = head;
}
```

// delete begin () //

```
void delete begin
{   struct node *ptr, *cptr;
    if(head==NULL)
    pf("\n CLL is underflow");
    else
    {  cptr = head;
       while (cptr->next != head)
       { cptr = cptr->next;
       }
    ptr = head;
    pf("\n deleted node : %d", ptr->data
    head = head->next;
    cptr->next = head;
    free(ptr);
  }
}
```

// delete_end() //

```
void delete_end ( )
{ Struct node *ptr, *cptr;
  if (head==NULL)
  pf("\n CLL is underflow");
  else
  { cptr = head;
     while ( cptr-> next != head)
       {   ptr = cptr;
           cptr = cptr->next;
       }
 pf ("\n deleted element: %d", cptr->data);
    ptr->next = head;
    free(cptr);
} }
```

// Searching //

```
void search()
{
    ────
    ────   } Same as SLL.
    ────
}
```

// Insert_after particular node //

```
void insert_after()
{
    ────
    ────   } same as SLL.
    ────
}
```

// deleting particular node //

```
void delete_after()
{
    ────
    ────   } same as SLL
    ────
}
```

2/08/19.

Double Linked List

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

struct node
{
    struct node *prev;
    struct ~~n~~ int data;
    struct node *next;
}
struct node *head=NULL.

void main()
{
    while(1)
    { switch(ch)
      {
      }
    }
}.

void create()
{   int c;
  struct node *ptr, *cptr;
  ptr= (struct node *) malloc (sizeof (struct node));
  pf ("\n enter first node data : ");
  sf (" %d", &ptr->data);
      ptr->prev=NULL;
      head= ptr;
pf("\n enter 0/1 for more nodes: ");
```

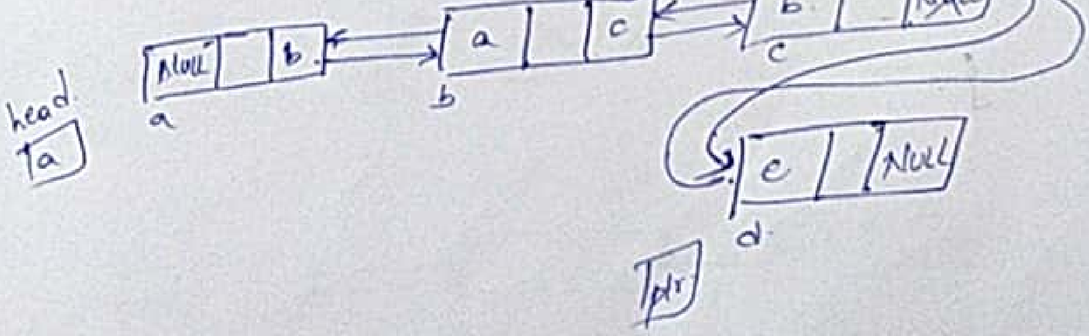
```
scanf("%d", &c).
while (c==1)
{
    cptr = (struct node *) malloc (sizeof (struct node));
    pf("\n enter node data:");
    sf("%d", &cptr->data);
      ptr->next = cptr,   cptr->prev = ptr;
      ptr = cptr;            pf("\n enter 0/1 for more:").
                             sf ("%d", &c);
   ptr->next = NULL;
 }
}

// void  insert() //  (begining).

void  insert_begin ()
{
   struct  node  *ptr;
  ptr = (struct node *) malloc (sizeof (struct node)).
  pf("\n enter node data:");
  sf("%d", &ptr->data);
     ptr->next = head;
     head ->prev = ptr;
     head = ptr;
     ptr->prev = NULL;
}.
```

8/8/2019.

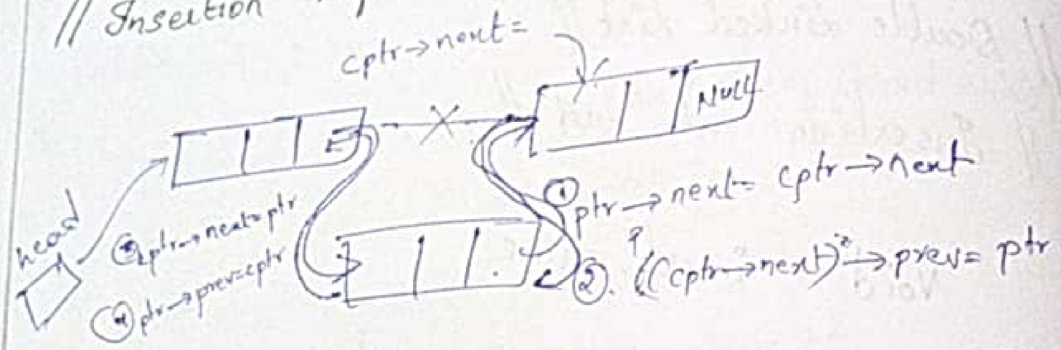
// Double Linked List //

// Insertion at end //

```
void insert_end()
{
    struct node *ptr, *cptr;
    ptr = (struct node *) malloc(sizeof(struct node));
    pf("enter node data: \n");
    sf("%d", &ptr->data);
    cptr = head;
    while (cptr->next != NULL)
    {
        cptr = cptr->next;
    }
    cptr->next = ptr;
    ptr->prev = cptr;
    ptr->next = NULL;
}.
```

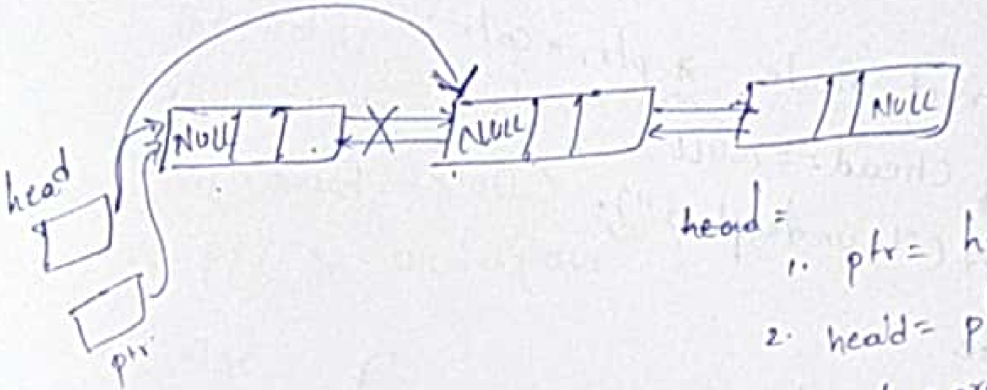


// Insertion After Given node //




```
void insert_after( )
{   int d;
    struct node *ptr, *cptr;
ptr= (struct node *) malloc (sizeof (struct node));
    pf("enter node data:");
    sf("%d", &ptr→data);
  pf("enter after which you want to enter:");
  sf("%d", &d);
  cptr= head;
  while (cptr→data != d)
{
   cptr= cptr→next;
}
ptr→next= cptr→next
(cptr→next)→prev= ptr;
cptr→next= ptr;
ptr→prev = cptr;
}.
```

// Deletion at begin //

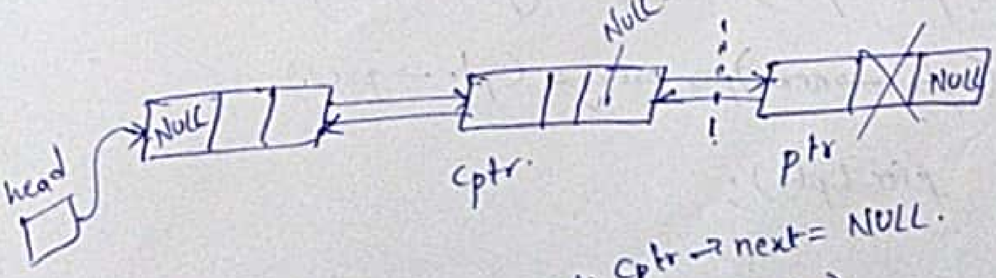


head =

1. ptr = head.
2. head = ptr → next
3. head → prev = NULL

```
void delete ( )
{
    struct node *ptr ;
    if (head == NULL)
    pf(" underflow \n");
    else
    { ptr = head ;
    head = ptr → next ;
    head → prev = NULL ;      pf("deleted: %d",
                                   ptr → data);
    free(ptr);
    }
}.
```

// Deletion at end //



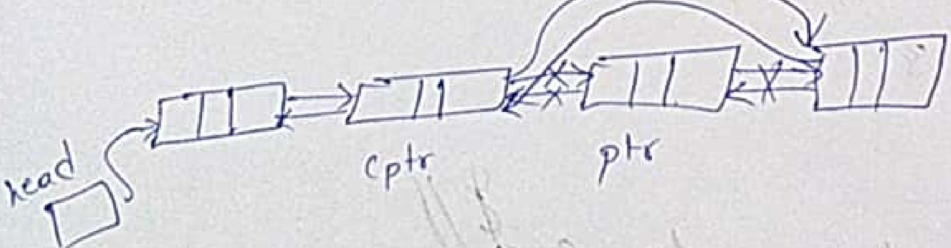
1. cptr → next = NULL.
2. free(ptr).

```
void delete end ()
{ struct node *ptr, *cptr;
  if (head==NULL)
  pf("\n underflow");

  else
  { ptr=head;
    while (ptr->next!=NULL)
    { cptr=ptr;
      ptr=ptr->next;
    }
    cptr->next=NULL;
  pf("deleted: %d", ptr->data);
    free (cptr);
  }.
```

(or)

```
(ptr->prev)->next=
          NULL;
free (ptr);
```

// Delete given node //



Alternate:-

```
(ptr->prev)->next = (ptr->next)
(ptr->next)->prev = (ptr->prev).
free (cptr);
```

Scanned by CamScanner

```
void    delete_after( )
{   int  d;
  struct node  *ptr, *cptr;
   if (head == NULL)
   pf("\n  underflow");

  else
   {      ptr= head;
       pf("\n enter  node to  be deleted ");
       sf("%d", &d);
        while ( ptr->data != d)
          {
               cptr = ptr;
               ptr = ptr->next;
          }
       cptr->next = ptr->next;
      (cptr->next)->prev = cptr;

       }
  }
}.
```