Date
02/08/19

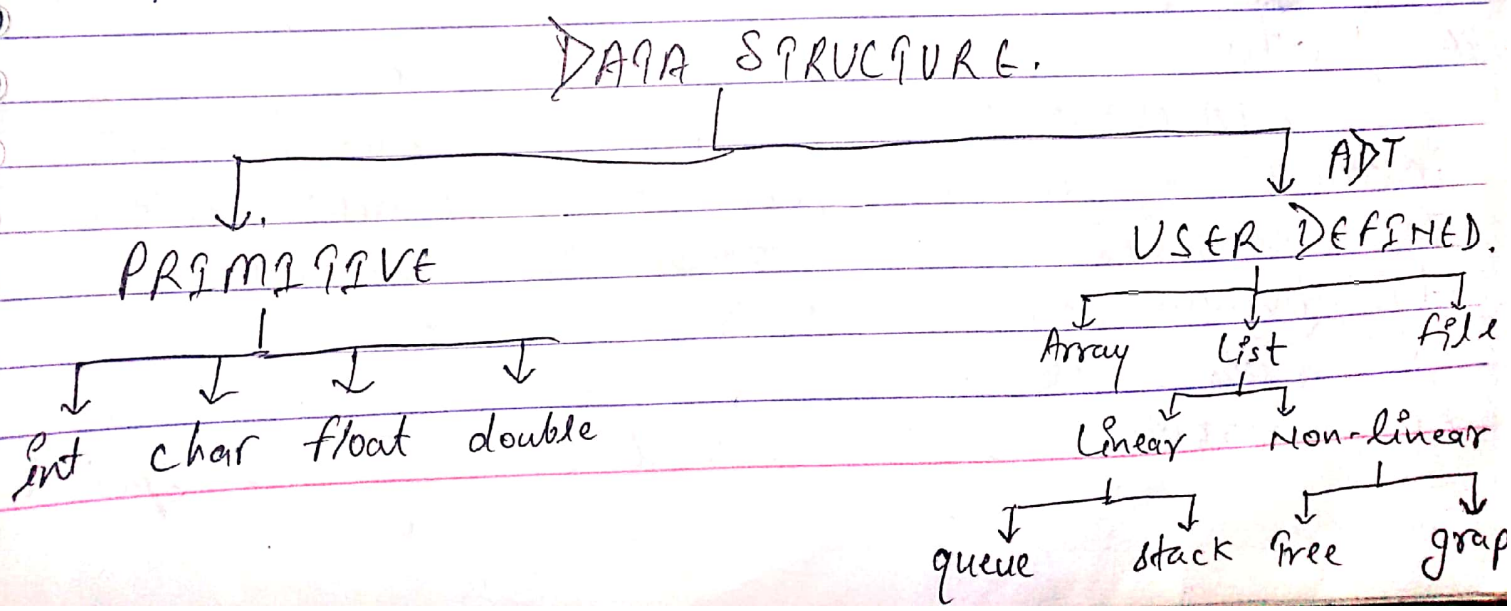
DATA STRUCTURE

* Data Structure is a way of collecting & organizing data in such a way that we can perform operations on these data effectively.

* Anything that can store data can be called as datastructure, hence int, float, char etc. are the datastructure.

* They are known as primitive (builtin) datastructure.
* We have some complex datastructure, which are used to store large & connected data. These are known as abstract datastructure (user defined data structure). for ex- LinkList, tree, graph, stack, Queue.

* All these datastructure allows us to perform different operations on data. We select these datastructure based on which type of operation is required. for example- Queue.



NEED OF DATA STRUCTURE.

* It gives different level of organization data. It tel
* It tells how data store & access in it's elementary level.
* Provide operation on group of data, such as adding an item, deleting an item, updating an item, searching an item etc.
* Provide a means to manage huge amount of data efficiently.
* Provide fast searching & sorting of data.

OPERATIONS ON DATA STRUCTURE.

* Traversing
* Searching
* Inserting
* Deleting
* Sorting
* Merging

ELEMENTARY DATA ORGANIZATION

* Data - Data can be defined as Representation of facts, concepts or instructions in a formulazid manner suitable for communication, interpretation or processing by human or electric machine.
* Data is represented with the help of characters like alphabets, digit & speial

character.

* Data items - A set of character which are used together to represent specific data element is called data items or fields. for example - name of student in a class represented by dataitem, say Name

* Record - Record is a collection of related dataitem.

* File - file is a collection of logically related records. for example - payroll file might consist of payrecords for a company.

* Entity - An entity is a person, place or thing, event or concepts about which information recorded.

* Attributes - Attributes gives the characteristics or properties of entity.

* Data value - A data value is a actual data or information contained in each attribute

Primary & Secondary Key -

* Primary Key - A field or a collection of fields in a record which identifies a record uniquely is called primary key. for example - ID NO.

* Secondary Key - A field in a record which identifies the record but not uniquely is called Secondary Key.

ALGORITHM -

* An algorithm is a finite set of instruction kept in a particular order to perform a particular task.

* Algorithm is not a complete code or program. It is just a core logic (solution), of a problem which can be expressed as an high level in formal description. It is also known as pseudo code.

* An algorithm can be expressed in three ways
  (i) In any natural language.
  (ii) In a programming language or in.
  (iii) Or in form of flowchart.

OR PERFORMANCE

* EFFICIENCY of ALGORITHM -

An algorithm is said to be efficient & fast if it takes less time to execute & consumed less memory space. The performance of an algorithm measured on the basis of two properties.

(i) Space Complexity
(ii) Time Complexity.

Programme under execution is known as process.

* SPACE COMPLEXITY -

It is amount of memory space required by an algorithm, during the execution of ~~any~~ algorithm.

Space complexity must be taken seriously for multiuser system & insituation where memory is limited.

An algorithm generally required space for following component.

(a) Instruction Space

It is a space required to store the executable version of programme. This space is fixed but varies depending upon no. of lines of code in a program.

(b) Data Space.

It is the space required to store all the constants & variables (including temperory variable) values.

(c) Environment Space

It is the space required to store the environment information needed to resume the suspended process.

* TIME COMPLEXITY

The time Complexity is a way to represent the amount of time required by the programe to run till its complition.

It is generally a good pratice to try to keep the time required minimum, so that our algorithm complete its algorithm in minimum time possible.

ASYMPTOTIC NOTATION -

*. Asymptotic Notation are languages that allow us to analyze an algorithm running time by identifying it's behaviour as the input size for an algorithm increases.

* This is also Known as algorithms growth rate.

* Asymptotic Notation means a way to write down & talk about the fastest & the slowest possible running time of an algorithm.

* BIG O NOTATION

• Big O is an asymptotic Notation for worst case running time.

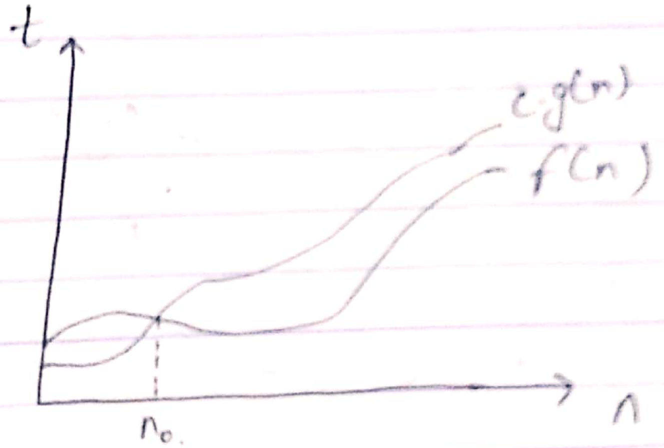
closest upper bound → tight upper bound

- This notation denotes the asymptotic tight upper bound of function f(n).

Let f be a non negative function then we define the three most uncommon asymptotic bound as follow-

f(n) is O(g(n)). if there exist a +ve constant c & n_0 such that $f(n) \leq c.g(n)$ where $c \geq 0$ & $n_0 \leq n$

If $0 \leq f(n) \leq c.g(n)$, we say that g(n) is tight upper bound of f(n).



for example- let $f(n) = 5n+2$. & $g(n) = n$

for $c = 6$

$5n+2 \leq 6.n$

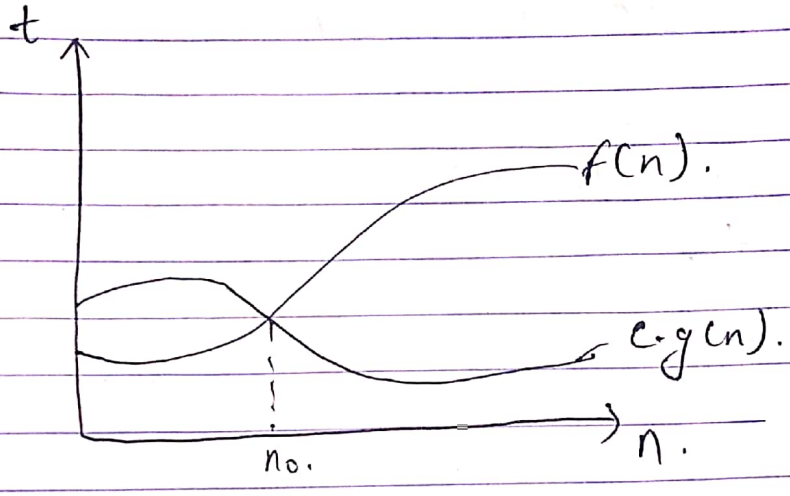
$n = 2$

* BIG Ω NOTATION -

- It is an asymptotic notation for the case running time.

- It provide asymptotically tight lower bound for given function $f(n)$.

- $f(n)$ is said to be $\Omega \cdot g(n)$ & written as $f(n) = \Omega(g(n))$. if & only if (iff) there are +ve constant c & n_0 such that $0 \leq c \cdot g(n) \leq f(n)$. for all $n_0 \leq n$.
  if $f(n) = \Omega g(n)$ we say that $g(n)$ is lower bound of $f(n)$.

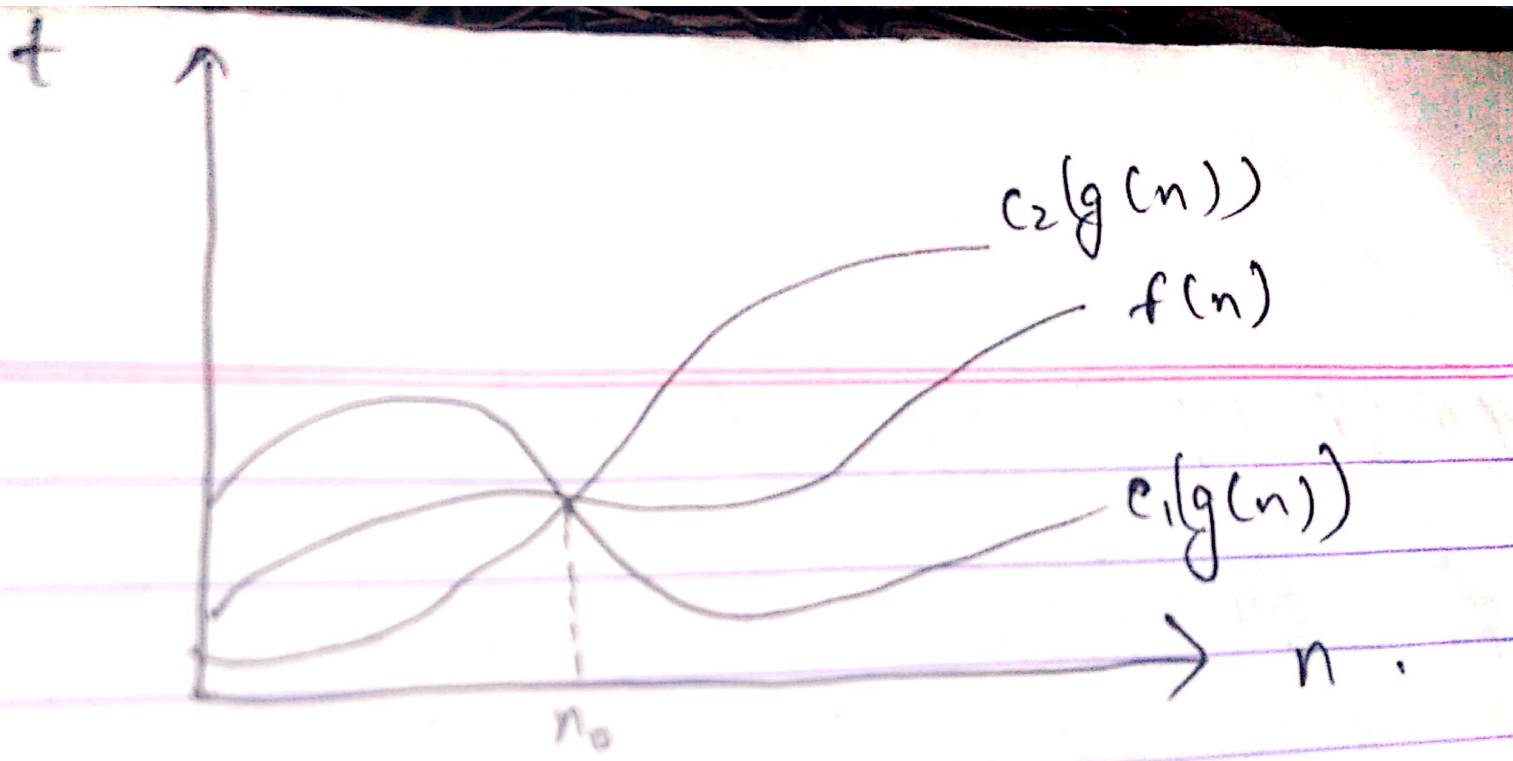


* BIG θ NOTATION -

- It is an asymptotic notation for average case running time.

- It is an asymptotic notation to denote asymptotically tight upper bound & tight lower bound.

- We say $f(n) = \theta(g(n))$ written as $f(n) = \theta(g(n))$ iff there are +ve constant c_1, c_2 & n_0 such that $0 \leq c_1(g(n)) \leq f(n) \leq c_2(g(n))$ $\forall c_1$ & $c_2 > 0$ & $n_0 \leq n$



Q ⇒
```
for(i=1; i<=n; i++).
{
    for(j=1; j<=i; j++).
    {
        for(K=1; K<=100; K++)
        {
            printf("ABC");
        }
    }
}.
```

$i = 1$	2	3	.....	K
$j = 1$	2	3		K
$K = 100$	100	100		100
$P = 1 \times 100$	2×100	3×100		$K \times 100$

$1 \times 100 + 2 \times 100 + 3 \times 100 + \ldots K \times 100$.

$100 (1 + 2 + 3 + 4 + 5 \ldots + n)$

$100 \left(\frac{n(n+1)}{2} \right) = O(n^2)$.

Q ⇒
```
for(i=1; i<=n; i++)
{
    for(j=1; j<=i^2; j++).
    {
        for(K=1; K<=n/2; K++)
        {
            printf("ABC");
        }
    }
}
```

$i = 1$	$i = 2$	$i = 3$	.....	n
$j = 1$	$j = 4$	$j = 9$		n^2
$K = n/2$	$K = n/2 \times 4$	$K = 9 \times n/2$		$n/2$.
$1 \times n/2$	$4 \times \frac{n}{2}$.	$9 \times 9 \times \frac{n}{2}$		

Pf $1 \times n/2 \quad 4 \times n/2 \quad 9 \times n/2 \quad \ldots \quad i^2 \times \frac{n}{2}$.

$\frac{n}{2} + 4 \times n/2 + 9 \times \frac{n}{2} + \ldots\ldots\ldots n^2 \times \frac{n}{2}$.

$\frac{n}{2}\left(1 + 2^2 + 3^2 + \ldots\ldots n^2\right)$.

$\frac{n}{2} \times \frac{n(n+1)(2n+1)}{6} = O(n^4)$.

Q
```
for (i=1; i<=n) i = i x 2).
    printf ("ABC");
```

$i = 1 \quad 1 <= n, \; 2 <= n, \; 4 <= n, \; 8 <= n, \; 16 <= n.$

$2^K = n \Rightarrow K = \log_2 n.$

Q1.
```
for ( i= n/2 ; i<=n ; i++)
    for ( j=1 ; j<=n/2; j++ ).
        for(K=1; K<=n ; K= K* 2) .        O(n^4)
            printf ("ABC");
```

Q2.
```
for ( i=n/2 ; i<=n ; i++).
    for (j=1; j<=n; j= j x 2).
        for ( K=1; K<=n; K= K* 2).        O(n^4)
            printf("Shiv");
```

Q3
```
A ()
{
    while (n > 1)                  → n/2 times   O(n)
        n = n/2.
        printf("ABC");
}.
```

ABSTRACT DATA TYPE - (ADT)

* Abstract Data Type is a process of providing only essentials & hiding the details is known as abstraction.

* Abstract Data Type is a type whose behaviour is defined by set of values & set of operations.

* The definition of ADT only mentions what operation are to be performed but not how these operations will be implemented.

* It involves what can be done in the data, not how has to be done.

* The user of data type only with the knowledge of values that can take & operation that can be performed on them. without any idea of how these types are implemented.

ARRAY

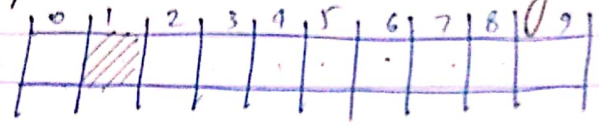
Array is a collection of homogenous data type.

Declaration -

data type a [size];

ex- int a [10];

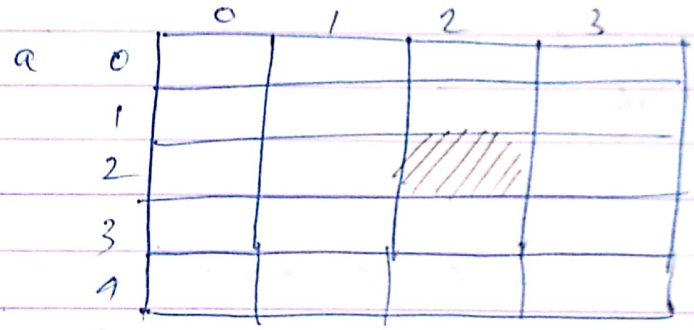
Actual (overview) Representation of memory -



2D - array declaration -

data type a [size1] [size2];

ex- int a [5] [4];



Overview Representation of Memory.

a[2][2].

* ROW MAJOR ORDER.

$$A[i,j] = B.S. + W[M[i] + j].$$

A[i,j] represents the address of ith row & jth column.

B.S. stand for Base Address

W stand for word size (size of data

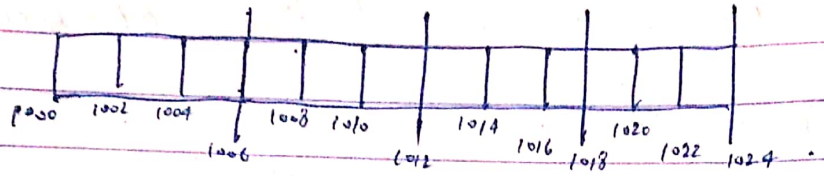
M is a no. of column in an array.

$$B.A. + M(R.(j) + i).$$

$$2000 + 9(6 \times 4 + 3)$$

Q a[4][3] whose B.A is 1000. Find B.A of a[3][2].

⇒



$a[3][2] = 1000 + 2(3 \times 3 + 2)$.

$= 1022$.

Q If an array of size 6×5 has an B.A. of 2000. What will be the B.A of a[3][4] R.M.O.

$a[3][4] = 2000 + 4(5 \times 3 + 4)$.

$a[3][4] \Rightarrow 2000 + 76$

$\Rightarrow 2076$

C.M.O.

$a[3][4] = 2000 + 4(6 \times 4 + 3)$

$= 2000 + 108$.

$= 2108$.

* COLUMN MAJOR ORDER.

When the indexing starts from zero.

$A[i,j] = B.A + M((j-1) + (i-1))$.

$A[i,j] = B.A + W(R(j) + i)$.

R B.A + W(i-(R_i)C + j-(C_j))

C B.A + W(j-(C_j)R + i-(R_i))

C = (C_n - C_1 + 1) ... (2+3) — 2 to 4

R = (B.A + W(i+2) ... C + j-2 (4)

2 3 9

... 4-2+1 ... (A) -(C)

4-(-2)+1

SPARSE MATRIX -

* Matrix with maximum zero's element is called sparse matrix.
* A Matrix containing more no zero's value then non-zero's value is called sparse matrix.
* A matrix which is not sparse is called dense matrix.

* REPRESENTATION

A Sparse matrix can be represented by two method (i) Triplet method it is also known as array method - (ii) Link List method.

* Representation of Sparse matrix using array -

- In this representation we consider only non-zero's value with their row & column index values.
- In this representation zeroth row stores total no. of rows, total no. of columns & total no. of non-zero's value in the sparse matrix.

for example- consider a following sparse matrix

	0	1	2	3	4
0	0	0	0	1	0
1	0	2	0	0	0
2	0	0	2	3	0
3	0	0	0	0	1

4 x 5

row	0	1	2	2	3
col.	3	1	2	3	4
value	1	2	2	3	1

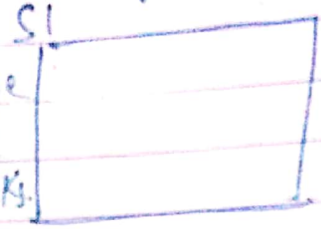
3 x n

where n is non-ze values.

*/ LINK LIST -

* Syntax for accessing a field by a pointer variable of structure type.

struct student s1, s2, *p.
p = &s1; (storing address of structure variable to pointer var p)



p→name	s1.name	name
p→roll	s1.roll	roll
p→mark	s1.mark.	marks

both are same.

* DYNAMIC MEMORY ALLOCATION -

Allocation of memory during execution of program is known as dynamic memory allocation. This is done by three special function in C which are as follow.

(i) malloc ()
(ii) calloc ()
(iii) realloc ()
(iv) free ()

(i) MALLOC

Malloc is used to create single unit of memory of given size during run tim.

(17)

S = struct student * malloc (sizeof(struct student));

Syntax for malloc is -

pointer_variable = (cast type*) malloc (size in bytes);

ex- p = int * malloc (sizeof (int));

(ii) CALLOC -

By using calloc function we create n element of corresponding datatype in contiguous memory allocation dynamically.

Syntax for Calloc() is →

pointer_variable = (cast type*) calloc (no of element, size of each element));

ex- p = (int*) calloc (10, size of (int));

Q Create a student array using calloc fn →.

struct student *s;

s = (struct student *) calloc (10, size of (struct student));

s = (struct student *) calloc (n, size of (struct stude

s → name	s[0]. name.
(s+1) → name.	s[1]. name .

Q WAP to create n student record having field name, rollno & marks. Print the detail of those student who had scored above 70% marks.

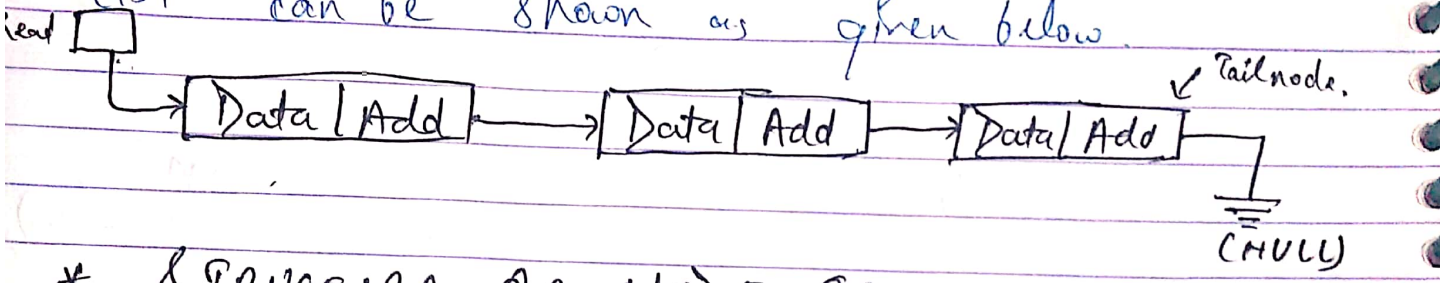
(3) pointer_variable = (cast type *) realloc (ptr, newsize in byte);
free (s)

```
#include <stdio.h>
struct marks
 {
  int m[5];
 };
struct student
 {
  char name[30];
  int rollno;
  int total;
  int per;
 struct marks sub;
 };
 void main()
 {
  int i, j, n;
  struct student s[n], *p;
 printf("Enter no. of Students");
 scanf("%d", &n);
 p = (struct student *) calloc (n, sizeof(struct student));
 printf("Enter Student Details \n");
 for(i=0; i<n; i++)
 {
  printf("Enter Student %d", i+1);
  printf("Enter student name");
  scanf("%s", (p+i)->name);
  printf("Enter student rollno");
  scanf("%d", (p+i)->rollno);
 for(j=0; j<5; j++)
  {
  printf("Enter Subjects %d", j+1);
```

```
        scanf("%d", (p+i)->sub.m[j]);
    }
}
for(i=0; i<n; i++)
{
(p+i)->total = 0;
for(j=0; j<5; j++)
{
(p+i)->total = (p+i)->total + (p+i)->sub.m[j];
}
(p+i)->per = (float)(p+i)->total/5;
}
for(i=0; i<n; i++)
{
if((p+i)->per > 70)
{
printf("Student Name & Roll No.\n");
printf("%s %d", (p+i)->name, (p+i)->rollno);
for(j=0; j<5; j++)
{
printf("Subject mark of %d", j+1);
printf("%d", (p+i)->sub.marks[i]);
}
printf("Total Percentage is %f", (p+i)->per);
}
}
```

LINK LIST -

Link list is a collection of **nodes** where each node contains two part a data part & a address part where data part contains information of different field of record whereas address part is a self referential field which contains the address of next node. Link list has one special pointer variable known as **head** which contains the address of first node. The last node of link list which is pointing to nothing (null) is known as **tail** of link list. Link list can be shown as given below.



* STRUCTURE OF NODE IS

```
struct node
{
    datatype  var1 ;
    datatype  var2 ;
      .
      .
      .
    datatype  varn ;
    struct node * link ;
};
```

Data

Address

* SYNTAX FOR CREATING A NODE -

```
struct node
{
    int
    ~~char~~ a;
    struct node * link;
};
void main()
{
struct node *p;
p = (struct node *) malloc (sizeof(struct node));
}
```

for accessing a we use p -> a.
for accessing next node p -> ~~next~~ link.

* VARIOUS OPERATION PERFORME ON LINKLIST

(i) Searching
(ii) Insert
(iii) delete
(iv) Traverse

* TRAVERSE -

```
void Traverse (struct node *p)
{
    while( p != Null)
    {
      printf("\n value is %d", p->a);
      p = p -> link;
    }
}
```

Null ka Next means segmentation fault.

* INSERTION

(i) While inserting a node we create a new node dynamically (ii) To fill the value of each field given by user in data part.
(iii) To search the position of node in existing list.
(iv) To link the node at desired position.

Insertion is done in three ways -

(i) Insertion at head. -

A new node is always inserted at head position.

(ii) Insertion at tail -

The new node is always inserted at after the last node of existing list.

(iii) Insertion in between (At desired location).

The new node is inserted between two existing node in existing list

(i) At head.

```
void Insert_at_beginning (struct node *p)
{
    struct node *t;
    t = (struct node *) malloc( sizeof(struct node));
    printf ("Enter any character");
    scanf ("%d", t->a);
    t->link = p;
    p = t;
}
```

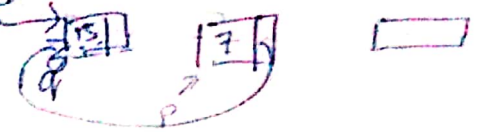
* Insertion on Tail -

```
void Insertionontail (struct node *head)
{
    struct node *n, *p;
    n = (struct node *) malloc (sizeof (struct node));
    printf (" Enter the character ");
    scanf (" %c", &n->a)
    n->link = NULL;
    if(head == NULL)
        head = n;
    else
    {
      p = head;
    while( p->link != NULL)
    {
    } p = p->link;

      p-> link = n
}
```

* SEARCHING -

```
void search ( struct node *head, int n).
{
  int f = 0, c = 0 ;
struct node *p ;
  p = head
  if (p == NULL)
  printf(" \n Element Not found List is Empty ");
   while (p
  else
   {
   while (p != NULL) {
    if (p -> a == n)
     {
        printf(" \n Element found in position
%od ",  f+1);
        c++;
     }
     f++ ;
     p = p -> link ;
  }
  if (c == 0)
  printf(" Element Not found ");
   }.
}.
```



DELETION

* Deletion of Head Node -

```
void Delete (struct node *head)
  {
    struct node *p;
    p = head;
    if ( p == NULL)
       printf("\n Link list is empty ");
    else
       head = head -> link;
       free(p);
  }
```

* Deletion of Tail Node -

```
void Delete ( Struct node *head).
  {
    struct node *p, *q;
    p = head;
    if (p == NULL)
       printf ("\n Link List is empty ");
    else
       q = p.
       while ( p -> next != NULL)
          q = p;
          p = p -> next
       if ( p == q)
       {
          head = NULL
```

```
        free(p);
        }
        else
        {
          q->next = NULL;
          free(p);
        }
      }
    }
```

* Deletion in between.

```
void Delete (struct node * head).
{
  struct node *p,*q;
  int t;
  p = head
  if (p == NULL)
  {
  } printf("\n Link list is empty");
  else.
  {
    printf("Enter the value");
    scanf("%d", &t);
    q = p;
    while(p != NULL)
    {
      if (t == p->a)
        break;
      q = p
```

```
    p = p -> link;
}
if (p == NULL)
    printf("\n Value not found in list No
            Node deleted ");
else
{
    if (p == q)
    {
    head = NULL
    free (p);
    }
    else
    {
      q -> next = p -> next
       free(p);
    }
}
}
}
```

CIRCULAR LINK LIST

DOUBLY LINK LIST -

(i) Insertion On Head -

```
void Insert_head ( )
{
    struct node *p, *pre, * next ;
    p = (struct node*) malloc( sizeof(struct node));
    printf(" Enter the value \n");
    scanf (" %d ", & p → a);
    if (head == NULL )
      {
        head = p ;
        p → pre = NULL;
        p → next = NULL;
      }
    else.
      {
        p → pre = NULL;
        p → next = head;
        head → pre = p
        head = p;
      }
}
```

(ii) Insertion On Tail

```
void Insert_tail ( )
{
    struct node *p, *q, * pre, * next ;
    p = ( struct node *) malloc (size of (struct node))
    printf (" Enter the value \n");
    scanf (" %d ", & p → a);
```

```
if (head == NULL)
{
    head = p;
    p -> prev = NULL;
    p -> NEXT = NULL;
}
else
{
    q = head;
    while ( q -> next != NULL)
    {
        q = q -> next;
    } q = q -> next;
    q -> next = p;
    p -> next = NULL;
    p -> pre = q;
}
}
```

DOUBLY

(iii) Insertion in between on the basis of data.

```
void Insert_in-between ( )
{
    struct node *p, *q;
    p = (struct node *) malloc (s.tze of (
    printf ("Enter the value \n");
    scanf ("%d", &p -> a);
    if (head == NULL)
    {
        head = p;
        p->link = NULL;
```

```
} p->pre = NULL;
else {
    q = head;
    if(q->link == NULL)
    {
        if(q->a < p->a)
        {
            q->link = p;
            p->pre = q;
            p->link = NULL;
        }
    else
        {
            p->pre = NULL;
            p->link = q;
            q->pre = p;
            head = p;
        }
    } // end of if.
else {
    while(q->next != NULL && q->a < new->a)
        q = q->next;
    if(q != head)
    {
        new->pre = NULL;
        new->next = q;
        q->pre = new;
        head = new;
    }
    else if(q->next == NULL && q->a < new->a)
```

P.T.O.

Insertion in b/w in decen

```
void insert (void)
{
    p = head
    node * pre , * ne , * new
    new = (node *) malloc (size of(node))
    printf("Enter the value");
    scanf("%d", & new → a);
    if (p == NULL)
    {
        head = new
        new → next = NULL;
    }
    else.
    {
        pre = head;
        ne = pre → next;
        if (ne == NULL)
        {
            if (pre → a) new → a )
            {
                new → next = head;
                head = new;
            }
        }
    }
    else
    {
while (ne != NULL && pre → a < new → a &&
        ne → a < new → a).
    {
        pre = ne;
        ne = ne → next;
    }
```

```
new -> next = pre -> next
pre -> next = new;
```

In Doubley Link list continoue.

```
{
    new -> next = NULL;
    new -> pre = q;
    q -> next = new;
}
else
{
    new -> pre = q -> pre
    new -> next = q;
    q -> pre -> next = new;
    q -> pre = new;
}
}
}
```