Asymptotic Notations:-

These are the expressions used to calculate the complexable of an algorithm.

* There are 3 types of Analysis that we perform for any algorithm.
  1. Best Case
  2. Worst Case
  3. Average case

Best Case:- It is a case in which further given input the algorithm takes less time and space.

Worst Case:- It is a case in which further given input the algorithm takes more time and space.

Average Case:- This lies b/w Best Case & Worst Case.

* There are 3 types of Asymptotic Notations:-
  (1) Big-oh (O) Notation
  (2) Theta (Θ) Notation
  (3) Omega (Ω) Notation

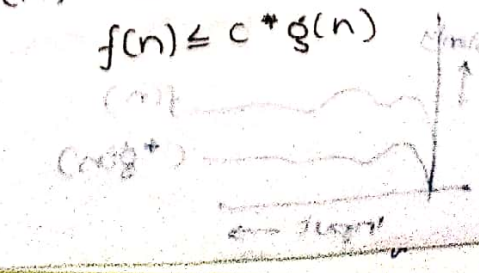
Big-O Notation:- This defines the upper bound of an algorithm.

* This discribes the worst case senario as it takes the maximum values than the input data.

Ex:- Consider $f(n) = 2n^2 + 3n + 5$ the find n & c value

(1) Under this big-oh condition this will be taken as $f(n)$ will be always $\leq c * g(n)$

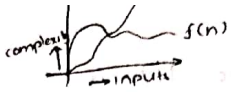
$$f(n) \leq c * g(n)$$



Sol:- Let $x=1$

$f(1) = 2(1)+3(1)+5$

$= 10$



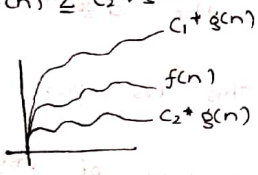
Here 10 is considered as the existing value such that $C=10$ & $n \leq 1$

Theta Notation :-

The theta Notation describes the bounded value in b/w the lower bound & upper bound.

* The condition with which the θ-Notation is applicable can be given as

$C_1 * g(n) \leq f(n) \leq C_2 * g(n)$



* As the value of θ - notation lies in b/w upper bound and the lower bound it resembles the average case.

Ex: $C_1 \leq 1/2 - 3/n \leq C_2$, n will be ≤ 7

$n \leq 7$

$C_1 = 1/2 - 3/7$

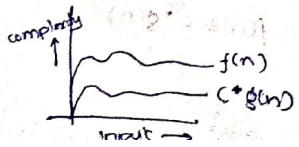
$C_1 = \frac{1}{14}$

$C_2 = 1/2 - 3/\infty = 1/2 - 0$

$= 1/2$

Omega Notation / Big-Omega Notation (Ω) :-

Omega Notation is used to defined the lower bound of $f(n)$



* This omega notation includes the condition

$f(n) \geq C * g(n)$

* It will be under best case analysed from many of the algorithm.

Ex:- Consider equations $f(n) = (n^3+n^2+n+1)$ &

$g(n) = 3n^2+2n+1$

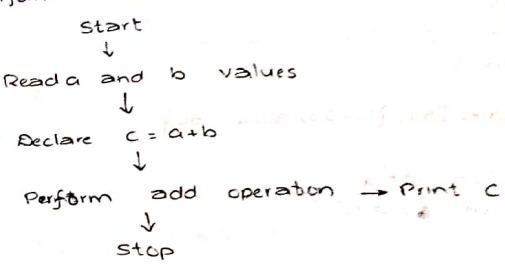
Let $n = 3$

$40 \geq 34$

$\therefore n \geq 3$,

Algorithm:- It is a step-by-step procedure used to describe the program execution.

Ex:- To perform add operation,



Arrays :- An array is a collection of the similar data items held at a (or) more adjacent cells

Syntax:- <datatype> <name of array> [size];

* Arrays are static elements in nature.
* Arrays have the static memory allocation.

Ex:- int a[5];
int marks[10];

* The array memory allocation will always starts with zero [0].

Types of Arrays :-

1. 1-dimensional Array
2. 2-dimensional Array
3. Multi-dimensional Array

One- dimensional Array :-

* It is also known as single dimensional linear array.

Declaration :-

Syntax :- <datatype> <array name> [size];

Ex :- int marks [10];

Initialization :-

Ex :- int marks [5] = {10, 20, 30, 40, 50};

char name[5] = {'a', 'b', 'c', 'd', 'e'};

Write a program to illustrate the declaration, initialization and accessing elements of an array.

```
<stdio.h>
<conio.h>
void main()
{
int marks [5]={10, 20, 30, 40, 50};
int i;
for(i=0; i<5 ; i++)
printf("%d", &marks[i]);
getch()
}
```

Write a program for declaring, reading and printing the array elements.

```
<stdio.h>
<conio.h>
void main()
{
int marks[5];
int i;
printf("enter array element");
for(i=0; i<5 ; i++)
scanf("%d", &marks[i]);
for(i=0;i<5;i++)
printf("%d",marks[i]);
getch();
}
```

Two- dimensional Arrays :-

A two- dimensional Arrays includes a number of indices which represent the row & the column data.

The basic application of the two-dimensional arrays is matrix representation

Syntax for declaration of 2- dimensional arrays :-

Syntax :- <datatype> <arrayname> [row] [column];

Ex :- int a [3] [3];

Syntax for initialization of 2- dimensional array :-

Syntax :- <datatype> <arrayname> [row][column]= {values};

int a[3][3]={1, 2, 3, 4, 5, 4, 7, 9, 8};

Write a program to illustrate the declaration, initialization and accessing of elements from 2- dimensional array.

```
<stdio.h>
<conio.h>
void main()
{
int a[3][3]={1,2,3,4,5,6,7,8,9};
int i,j;
for(i=0; i<3;i++).
for(j=0; j<3; j++)
printf("%d", a[i][j]);
getch();
}
```

Write a C-program to illustrate, declaring, reading & printing the array elements of 2- dimensional array

```
<stdio.h>
<conio.h>
void main()
{
int a[3][3];
int i,j;
for(i=0; i<3; i++).
for(j=0;j<3;j++)
scanf("%d", &a[i][j]);
printf("%d", a[i][j]);
getch();
}
```

Write the program to find product of 2 matrices.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[2][2], b[2][2], c[2][2], i,j,k;
clrscr();
printf("enter 2*2 matrix elements :");
for(i=1;i<=2;i++)
{
for(j=1; j<=2; j++)
scanf("%d", &a[i][j]);
}
printf("\n enter 2*2 matrix elements:");
for(i=1; i<=2; i++)
{
for(j=1; j<=2; j++)
scanf("%d", &b[i][j]);
}
printf("the 2*2 matrix elements of a:");
for(i=1; i<=2; i++)
{
printf("\n");
for(j=1; j<=2; j++)
printf("%d\t", a[i][j]);
}
printf("\n the 2*2 matrix elements of b:");
for(i=1; i<=2; i++)
{
printf("\n");
for(j=1; j<=2; j++)
printf("%d\t", b[i][j]);
}
for(i=1; i<=2; i++)
{
printf("\n");
for(j=1; j<=2; j++)
{
c[i][j]=0;
for(k=1; k<=2; k++)
```

```
{
c[i][j]= c[i][j] +a[i][k] * b[k][j];
}
}
printf("\n the product of two matrices: ");
for(i=1; i<=2; i++)
{
printf("\n");
for(j=1; j<=2; j++)
printf("%3d", c[i][j]);
}
getch(),
}
```

(OR)

```
void main()
{
int a[30][10], b[10][10], c[10][10], r, c, i, j, k;
printf("enter no of rows");
scanf("%d", &r);
printf("enter no of columns");
scanf("%d", &c);
printf("enter the values of 1st matrix:");
for(i=0; i<r; i++)
{
for(j=0; j<c; j++)
{
scanf("%d", &a[i][j]);
}
}
printf("enter the values of 2nd matrix:");
for(i=0; i<r; i++)
{
for(j=0; j<c; j++)
{
scanf("%d", &b[i][j]);
}
}
```

```
printf("multiplication of matrix a & b");
for(i=0; i<r; i++)
{
for(j=0; j<c; j++)
{c[i][j]=0; // here the initial value of c[i][j] is considered
   as zero(0).
for(k=0; k<c; k++)
{c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
}
}
```

Multi-dimensional Arrays:-

The arrays with 2 (or) more than dimensions can be called as multi dimensional arrays.

Syntax of Multi dimensional Arrays

Syntax :- <datatype> <arrayname> [size1][size2] .. [sizen];

Ex :- int a[2][2];
(or) int a[2][3][2];

In 3-dimensional matrix the first index represents the dimension where as 2 & 3 indices represents the rows & columns.

Ex :- For the statement int a[2][3][2] the output can be obtained as

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \quad \begin{bmatrix} 2 & 3 \\ 2 & 4 \\ 2 & 6 \end{bmatrix}$$

Here dimensions means no of matrices.

Pointer Arrays :-

These are the collection of addresses. The address of one array pointer varies from the representation of other elements address i.e the addresses are always will be unique in nature.

The pointers can be represented by the symbol "*"

Ex:- int *a[3];

The syntax of pointer Arrays is

Syntax:- <datatype> *<arrayname>[size];

Ex:- int *a[3];

```
* int *a[3];
  int i=10, j=20, k=30, y;
    a[0] = &i;
    a[1] = &j;
    a[2] = &k;
  for(y=0; y<3; y++)
  printf("%d", *a[y]);
```

Dynamic Memory Allocation:-

This covers all the disadvantages of arrays concept by allowing the memory to be allocated Dynamically.

There are 4 Dynamic Memory Allocation functions:

* Malloc()
* calloc()
* realloc()
* free()

These are used to allocate and dellocate the memory of a variable.

malloc() is used to dynamically allocate single large block of memory with a fixed size

Syntax of malloc() is

syntax:- ptr = (cast-type *) malloc(byte-size);

Ex:- ptr = (int *)malloc(100*(size-of(int));

This allocates 200 (or) 400 bytes of memory all together when taken as int (or) long-int.

calloc() - [contigous Memory Allocation]:-

It is used to allocate specified no.of block memory for the exact type

syntax:- ptr = (cast type *)calloc(n, element size);

Ex:- ptr = (float*) calloc(25, size of (float));

realloc() :- [Reallocation]:-

Reallocation is used to change the memory allocated by either malloc() (or) calloc() such that it enables the new size to a variable

syntax:- ptr = realloc(ptr, newsize);

new size indicates the size given for reallocation

Ex:- ptr = realloc(10* size-of(float));

free() :- It is used to de-allocate the given size of a variable

syntax:- free(ptr);

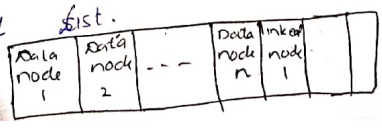
Linked Lists :-

It is a data structure that contains sequence of records stored at nodes where a node is a combination of 2 blocks.

(i) Data block (or) Data node (or) Data item

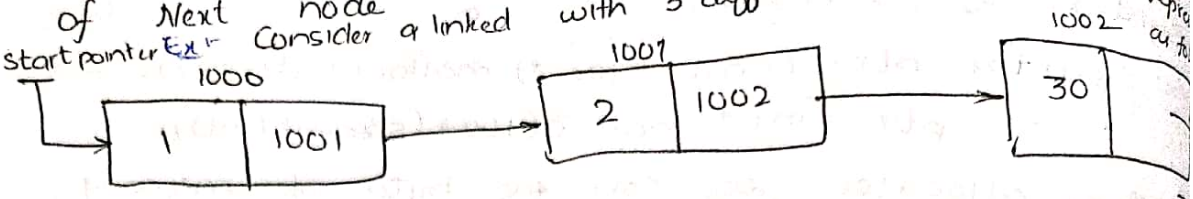
(ii) Address block (or) Address node (or) Link node

Syntax (or) Representation of Node structure in Linked list.



Linked list is the series of ... where uses the pointer node to store the address of Next node

Ex:- Consider a linked with 3 different data elements



Advantages of Linked List:-

* It allocates the memory dynamically.
* In these linked list the insertion & deletion of data can be done easily.

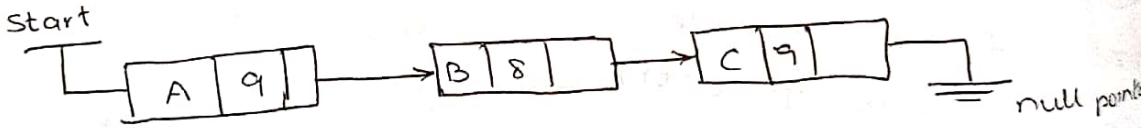
Types of Linked Lists:-

There are different types of linked lists.

1. Single Linked list:-

This is the data item where each node can have one (or) many nodes but only one link node

* The single linked list has only one link node as given below



Operations on Single Linked list:-

(1) Insertion
(2) Deletion
(3) Traversal

C.

Single linked list Operations:-

Creating of linked list :-

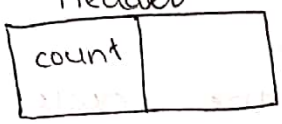
steps for creating a new node of single linked list :-

(1) Allocate the node

(2) Use the "getnode( )" of function as call getnode(x)

(3) The set the head link i.e of the particular list to null and also the list count to zero(0).

(4)



Here the count is used to store the no.of elements (or) no.of nodes in the single linked list.

(5) The second part of header is head which is used to create the new nodes.

(6) Insertion of Data :-

Insertion of Data can be done in three ways.

1. Inserting at Begining
2. Inserting at Ending (or)
3. Inserting at Middle.

Inserting at Begining:-

This is used to insert the node at the begining of the linked list such that by creating the very first node attached to the header node created

Input :- Here header is the pointer to the header node and the x is consider as the data of the node which is used to be inserted

Steps :-

```
new = getnode(node)
If(new = null) then
printf("memory underflow");
exit
else
new → link = header → link;
new → data = x;
header → link = new;
end if;
```

```
new - getnode (node)
if (new = null) th
printf (" memory
  exit
else
new → link = header → l
newdata = x;
header → link = new.
end if.
```

Insert the value 15 at the begining of the node

Insertion at Ending :-

This is used to insert the node at end of linked list.

Input :- Header is the pointer to the head node & x is the node we need to insert at the end

Steps :-

1) ptr = header; // move the position of curser(or) pointer get executed from starting of the node.
2) while (ptr → link ≠ NULL) // It checks whether the pointer link is null (or) not
3) ptr = ptr → link // change the pointer to the next node
4) end while
5) ptr → link = new; // change the link field of a last node as a pointer.
6) new → data = x; // change the content x into a new node
7) end;

Ex :-

Insertion at Middle :-

Always the insertion at the middle is done just based on the key element [i.e the element after which the node to be inserted]

Algorithm :-

```
new = Getnode (node)
if (new = null) then
printf ("Memory insufficient");
exit
```

```
else
ptr = Header
while (ptr → data ≠ key)  and (ptr → link ≠ Null) do
ptr ⇒ ptr → link
end while
if (ptr → link = Null) then
printf("key is not available in list");
exit
else
new → link = ptr → link
new → data = x
ptr → link = new
end if
stop
```

Deletion of Nodes from the Single linked list :-

The deletion can be made by 3 ways

(1) Delete at Begining
(2) Delete at End
(3) Delete at any position from the linked list.

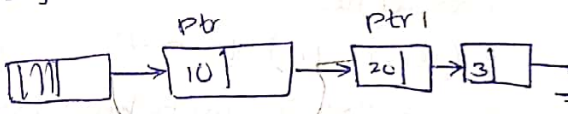
Delete at Begining :-

Here the input includes the header node.

Code :

```
ptr = Header → link
if (ptr = Null) then
printf("no deletion");
exit
Else
{ ptr1 = ptr → link
  header → link = ptr1
return node (ptr)
end if
exit | stop
```

Before Deletion



After Deletion

Deletion at Ending :-

This is used to delete last node of list & makes the previous node as last node

Steps :-

* Move from the header node
* Move to the last node to store the previous pointer & then move to the next node

Code :-

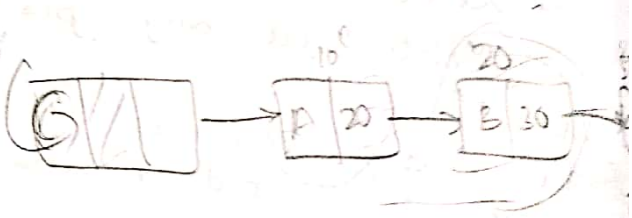
```
ptr = header → link
if (ptr → link = Null) then
printf("list is empty");
exit
else
while (ptr → link ≠ Null) do
```

```
ptr1 = ptr;
ptr = ptr → link
end while
ptr1 → link = NULL
return node (ptr)
```

Deletion at any position from the linked list :-

Code :-

```
while (ptr ≠ NULL) do
if (ptr → data ≠ key) then
ptr1 = ptr
ptr = ptr → link
else
ptr1 → link = ptr → link
return node (ptr)
exit
end if
end while
if (ptr = NULL) then
printf("Node with key doesn't Exist");
end if
stop
```



* Key element is the one after which the element is ~~inserted.~~ deleted

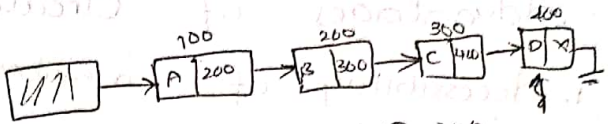
* In deletion the element which we need to delete is called Key Element, and the element before that is known as

Traversal:-

Traversal is used to display the elements or the data fields that are present in the given linked list from the begining of the node till it detects the null link.

code :-

```
struct node *P
P = start
while (P!=NULL)
{
printf(" P→data");
P = P→link;
}
```



P=100
100!=NULL (T)
A
P=200

P=200
200!=NULL(T)
B
P=300

P=300
300!=NULL(T)
C
P=400

P=400
400!=NULL(T)
D
P=NULL
NULL!=NULL(F)

Merging Two Different Linked list:-

* Combining of two lists as one can be done by the use of Merging Process.
* In this consider 2 Header nodes as Header-1 & Header-2 which are the pointers of both the Header nodes of 2 lists L1 & L2 then after merging the nodes delete the Header-2 node such that only one header node will exists.

Code:-

```
ptr = Header
while (ptr→link ≠ NULL) do
ptr = ptr→link
end while
ptr→link = Header 2→link
return node (Header 2)
Header = Header 1
stop.
```

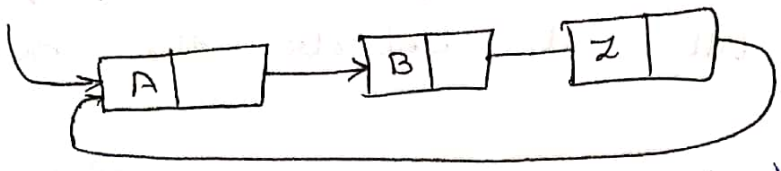
Disadvantages of Single Linked list:-

In single list the last nodes link field is left with the null link without storing any data also

*Also the starting nodes address will not be stored at any link field of the nodes. So circular linked list is used in order to overcome these disadvantages.

Circular linked list :-

General representation of circular linked list as follows



Advantages of Circular linked list :-

1. Accessibility of member nodes in the list is done efficiently.
2. All the nodes here acts as member nodes and can be access from the any node using Chaining Method.
2. NULL- Link problem

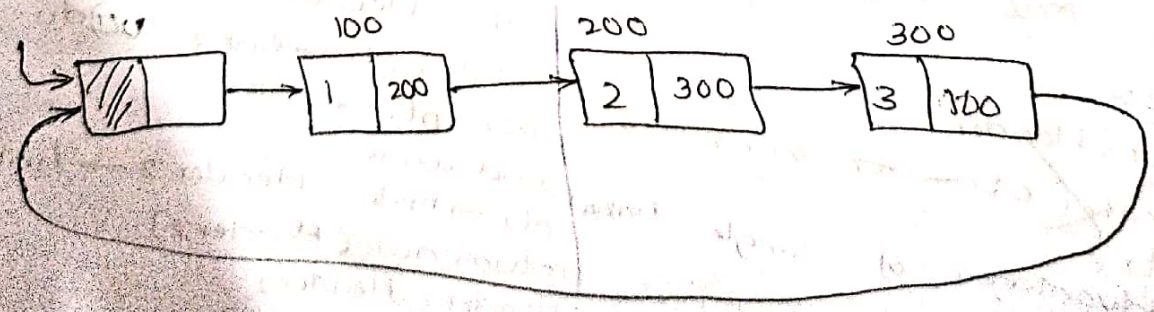
No NULL nodes are detected in the circular linked list

3. It is easy to operate the different operations on this linked list.

Disadvantages of Circular

This results in infinite loop problem becau we do not have a null link in the circular linked list

Ex :- Or.



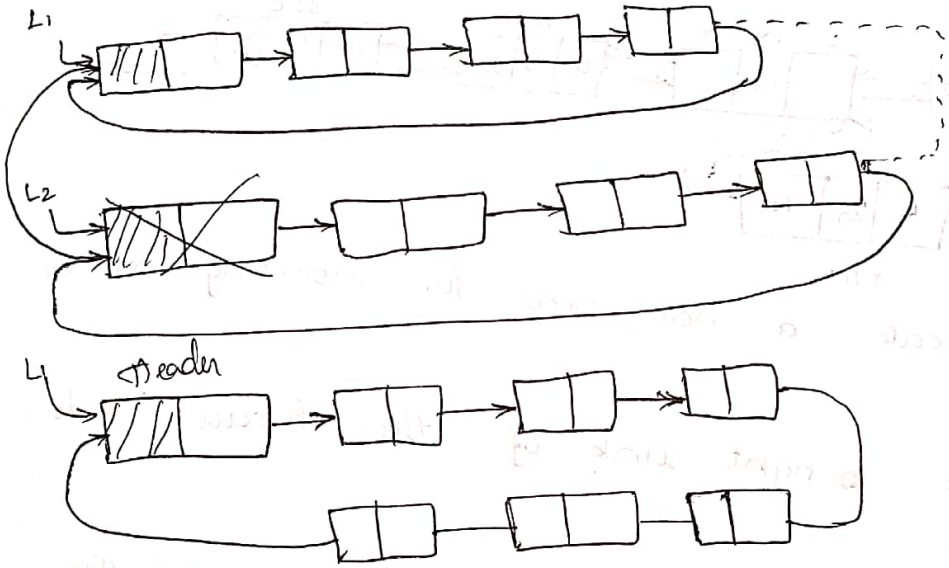
Concatenation of two circular linked lists :-

Consider two circular linked list and make those to be combined as a single circular linked list.

Code :-

```
if L1 ≠ NULL
{
if L2 ≠ NULL
{ temp = link(L1)
  link(L1) = link(L2)
  link(L2) = Temp
   L1 = L2
  Header1 = Header2
  Header = Header1
}
}
```

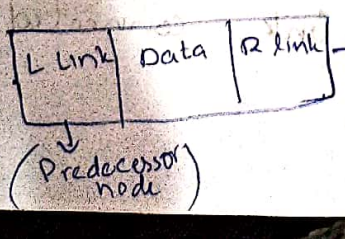
```
if L1 ≠ NULL
{
if L2 != NULL
{ temp = link(L1)
  Link(L1) = Link(L2)
  Link(L2) = temp
   L1 = L2
  Header1 = Head2
  Header2 = Header1
}
}
```



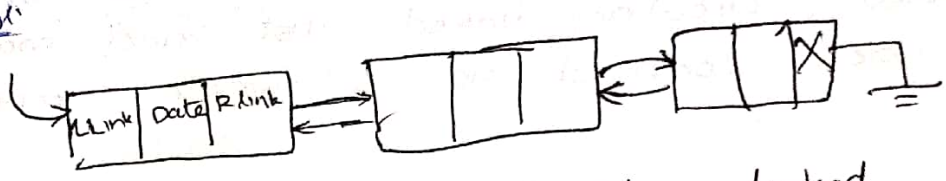
Double linked List :-

In the single linked list the nodes will be processed from left to right and hence it is called One Way list, and in this single linked list we will have only one link field connected towards the successive nodes.

The representation of Double linked list is.



Ex:



The operations of Double linked lists are

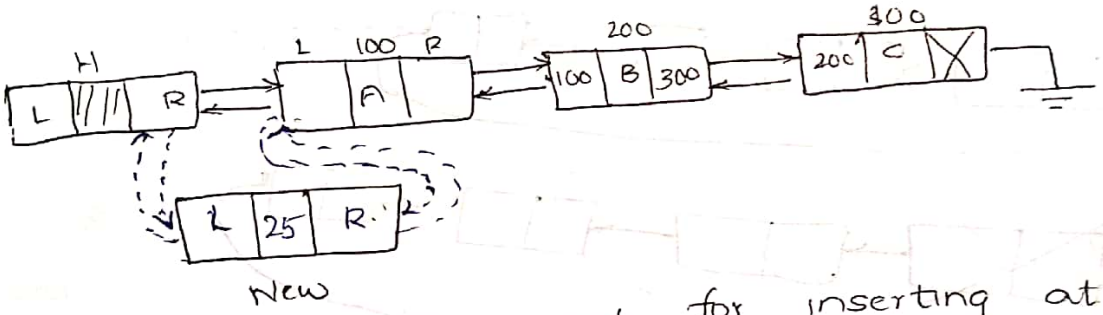
(1) Insertion
(2) Deletion &
(3) Traversal Mainly

Where as both the insertions & Deletions can done in 3 ways as at the beging, and at any position of the list.

Insertion of Elements in Double linked list.

Inserting at Beginning:

Let New be the node to be inserted.



* Initial create a new node for inserting at the begining.
* Make the right link of the header to be as a ptr
* Then create the new node to insert the data.
* Header is connected to the L-link of new node
* The new node is assigned to the R-link of the header.
* Connect the pointer to the new & nodes R link
* Make the pointer's L-link connected to the new.

Code :-

```
ptr = Header → R link
new = Get node (node)
if (new ≠ NULL) then
① new → L link = header
② header → R link = new
③ new → R link = ptr
④ ptr → L link = new
new → data = X
```

```
ptr = Header → R link
new = Get node (node)
if (new ≠ NULL) then
```

Insertion at End :-

To insert an element at the end

* First make down the pointer as the Header then check pointer's R-link is equal to NULL (or) not
* If true then insert the element at the position. (or) else make the pointer to move from one node to the next node.

Code :-

```
ptr = Header
while ( ptr → R link ≠ Null) do
ptr = ptr → R link
end while
new = Get node (node)
if (new ≠ null) ~~do~~ then
① new → L-link = ptr
② ptr → R-link = new
③ new → R-link = NULL
new → data = X
```

```
ptr = Header
while ( ptr → R link ≠ NULL) do
ptr = ptr → R link
end while
new = Get node (node)
if (new ≠ null) then
new → L-link = ptr
ptr → R-link = new
new → R-link = NULL
new → data = x
```

Insertion at Middle :-

To insert at the middle make the data to be inserted as x and pointer be the predecer node where you want to insert and ptr 1 be the successor node of the newly inserted data node.

code :-

```
while (ptr → data ≠ key) and (ptr → R link ≠ NULL)
ptr = ptr → R link
end while.
new = Getnode (node)
If(new = null) then
printf("memory not available");
exit
end if
If (ptr → R link = Null) then
new → lt link = ptr
ptr → R-link = new
new → R-link = Null
new → data = X
else
ptr1 = ptr → R-link
new → L-link = ptr
new → R-link = ptr1
ptr → R-link = new
ptr1 → L- Link = new
ptr = new
new → Data = X
end if
stop
```

* Even the key node can also consider after which you want to insert the data for that check whether the key node is present in the list or not.

```
while (ptr → data ≠ key)
    and (ptr → R link ≠ Null)
ptr = ptr → R link
end while
```

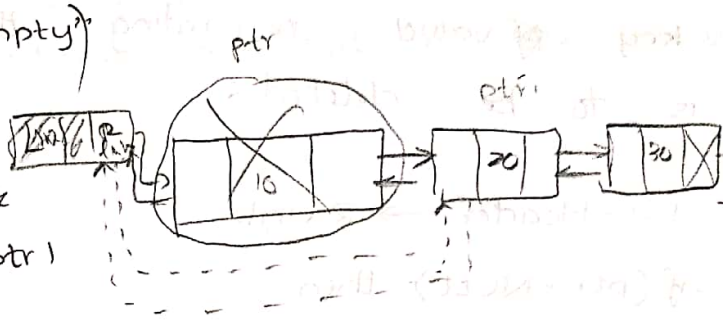
Note: If the key element is not found in the double linked list during the process of inserting at the middle then the elements gets to be inserted at the end of the list.

Deletion of Elements from Double linked list

Deleting at Begining :

Deleting at the front includes a pointer node given as right link of the header node

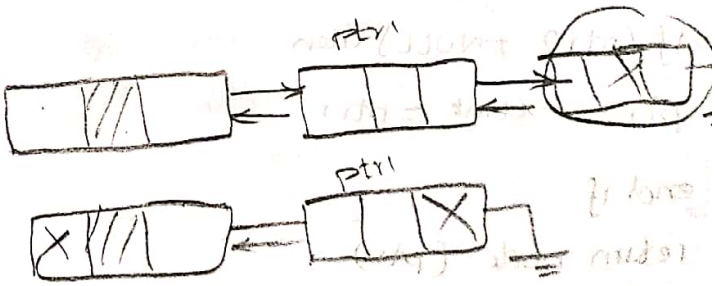
```
ptr = Header → Rlink
if (ptr = NULL) then
  print(" it is empty")
exit
else
  ptr1 = ptr → Rlink
  Header → Rlink = ptr1
  if (ptr1 ≠ NULL)
  ptr1 → Llink = Header
  end if
  returnnode (ptr)
stop
```



Deleting at End :

To delete at the end we find the last node to be pointed to a null node such that the null node will be connected to the new node created

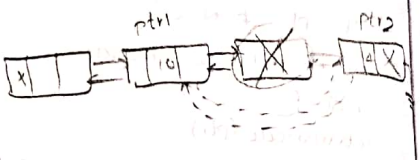
```
ptr = Header
while (ptr → Rlink ≠ NULL) do
ptr = ptr → Rlink
end while
if (ptr = header) then
printf("list is empty");
exit
else
ptr1 = ptr → L-link;
ptr1 → Rlink = NULL;
returnnode (ptr)
end if
```



Deleting at any position of the Double linked list:

In this case of deleting the data from any where of the list we make the search from the beging of the list i.e from header and we make the use of key value regarding the element which is to be deleted.

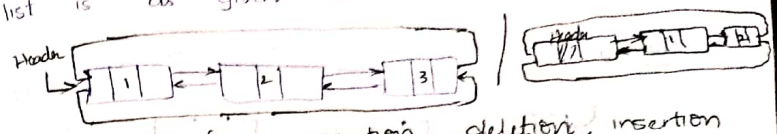
```
ptr = Header → Rlink
if (ptr = NULL) then
    print (empty)
    exit
end if
while (ptr → Data ≠ key)
        and (ptr → Rlink ≠ NULL) do
    ptr = ptr → Rlink
end which
if (ptr → data = key) then
    ptr1 = ptr → llink
    ptr2 = ptr → R link
    ptr1 → Rlink = ptr2
    if (ptr2 ≠ NULL) then
    ptr2 → llink = ptr1
    end if
    return node (ptr)
    else
print 'node not found'
end if
stop
```



Circular Double Linked List:

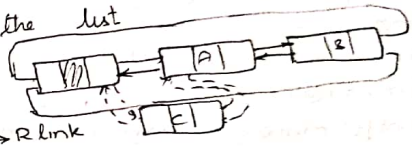
* It is a combination of circular linked list and double linked list.

Basic Representation of Circular Double linked list is as given below:-



* This also performs insertion deletion, insertion and Traversal operations as that of the circular double linked list.

* Here insertion and the deletion operations can be done in 3 ways as
  - At begining
  - At ending
  - Any where at the list



At begining :-

code:-

```
ptr = Header → Rlink
new = Getnode (node)
if (new ≠ NULL) then
new → llink = header
header → Rlink = new
new → Rlink = ptr
ptr → Llink = new
new → data = x
```

At End :-

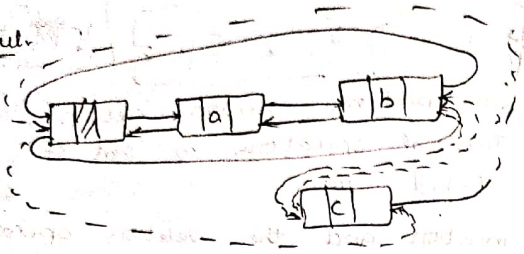
While inserting at the end the last node connected to the very node to the list will be modified and re-connected to the newly added node.

code:-

```
ptr = Header → if (ptr → rlink ≠ Header)
                { ptr = ptr → rlink }
ptr → Rlink = header }
header → llink = ptr
Header
while (ptr → Rlink != NULL)
{
  ptr = ptr → rlink;
}
```

new → llink = ptr,
new → rlink = header,
header → l link = new,
new → data = x.

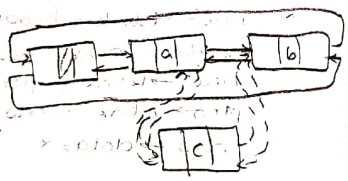
Output:



Insertion at Any Position:

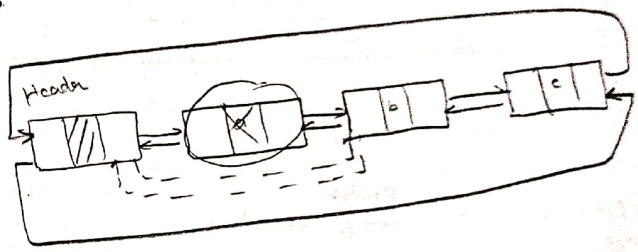
code:-

```
while(ptr → data ≠ key) and (ptr → rlink ≠ Header)
ptr = ptr → rlink
end while
new = Get node (node)
if(new = null) then
printf(" memory not available");
exit
end if
if(ptr → Rlink = Header) then
new → l-link = ptr
ptr → R-link = new
new → R-link = Header
new → data = x
else
ptr1 = ptr → R-link
new → l-link = ptr
new → R-link = ptr1
ptr → R-link = new
ptr1 → l-link = new
ptr = new
new → Data = x
end if
stop
```



Deletion at Beginning:

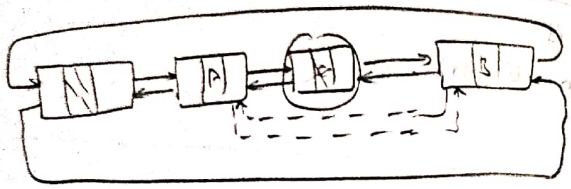
```
ptr = Header → R link
if(ptr = Header) then
printf(" It is empty");
exit
else
ptr1 = ptr → R link
Header → R link = ptr1
if(ptr1 ≠ Header)
ptr1 → l link = Header
end if
return node (ptr)
stop
```



Deleting at Any Position of the Circular Double linked list:-

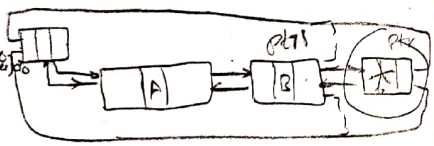
```
ptr = Header → R link
if(ptr = Header) then
print ("empty")
exit
end if
while (ptr → Data ≠ key) and (ptr → R link ≠ Header) do
ptr = ptr → R link
end which
if ptr1 = ptr → l link
ptr2 = ptr → R link
ptr1 → R link = ptr2
```

if (ptr2 ≠ Header) then
ptr2 → l-link = ptr1
end if
return node(ptr)
else
print("node not found");
end if
stop



Deleting at End :-

ptr = Header
while(ptr → rlink != Header)
ptr = ptr → Rlink
end while
if (ptr = header) then
printf("list is empty");
exit
else ptr1 = ptr → llink;
ptr1 → Rlink = header;
header → llink = ptr1;
return node(ptr);
end if



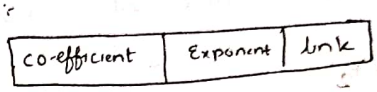
Applications of linked lists :-

* The linked lists can be used in various applications such as
  (i) Polynomial expression
  (ii) Addition of Polynomial Expression
* Sparce Matrix representation

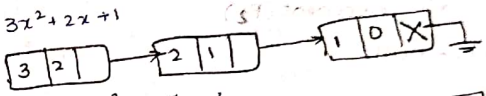
{Addition of} Polynomial Expression :-

The representation of the polynomial Expression used by list is given by

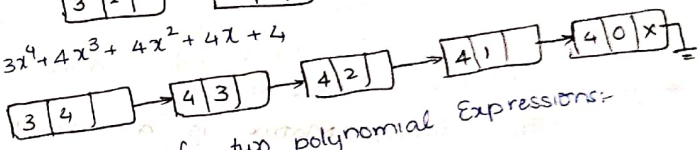
Syntax :-



Ex :- $3x^2 + 2x + 1$



$3x^4 + 4x^3 + 4x^2 + 4x + 4$



The Addition of two polynomial Expressions :-

Algo / Code :-

input → co-efficient & exponent values of given polynomial expressions
i.e Let P_1 & P_2 be the polynomial expressions

output The resultant polynomial expression P [the addition of the two polynomial expressions P_1 & P_2]

code :-

```
if[ ( (exponent(P1)) = ((exponent (P2)))
{
if(co-eff(P1) + coeff(P2)) ≠ 0 then
{
call Getnode(x);
coeff(x) = coeff(P1) + coeff(P2);
exponent(x) = exponent(P1);
link(x) = null;
}
```

```
If (exponent (P1) > exponent (P2))
 {
 Call getnode (x);
  coeff (x) = coeff (P1)
  exponent (x) = exponent (P1)
  link (x) = null
 }
If (exponent (P1) < exponent (P2))
 {
 call getnode (x);
 coeff (x) = coeff (P2)
  exponent (x) = exponent (P2)
  link (x) = null
 }
```

* Sparce Matrix Representation:

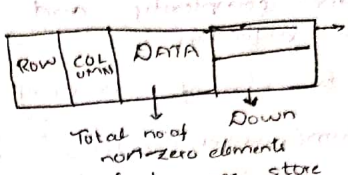
The Sparce Matrix is a matrix which contains maximum no. of zero elements.

Ex:- 4 x 5

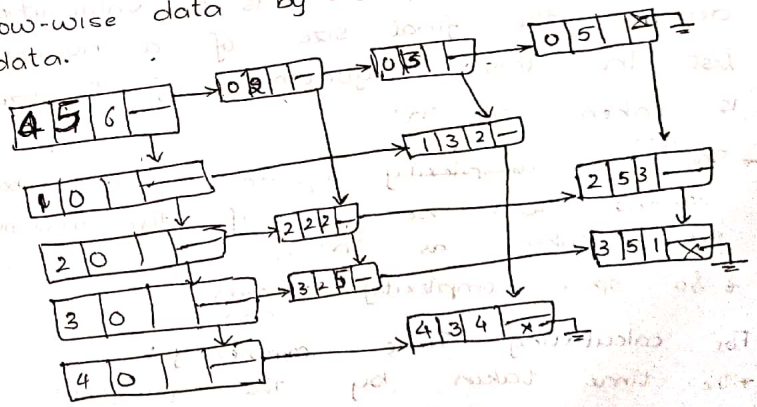
	1	2	3	4	5
1	0	0	(2)	0	0
2	0	(2)	0	0	(3)
3	0	(5)	0	0	(1)
4	0	0	(4)	0	0

$$\begin{bmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 2 & 0 & 0 & 3 \\ 0 & 5 & 0 & 0 & 1 \\ 0 & 0 & 4 & 0 & 0 \end{bmatrix}$$

* For representing the Sparce matrix we use array representation of elements.
* Here multiple linked list is used for the representation of Sparce matrix.
* The node structure of linked list uced for representing the sparce matrix



* From the right link we store the next non-zero elements and from the left link we store the non-zero elements of columns and rows. Here each field is connected with each other in the form of a linked list.
* Procedure:
* The very start node is taken as total no. of rows, columns, data and down link and the right link.
* Then mark the column wise data from the right link of the start node by disabling the row data.
* From the ~~row wise~~ down link mark the row-wise data by disabling the column data.



Calculating Time complexity and space complexity for example algorithm.

*Algorithm for sum of n-no.of digits / values

```
Algorithm sum(number, size)
{
result = 0.0;
for count = 1 to size do
result = result + number [count];
return result;
}
```

To calculate space complexity find 2 different components fixed component and variable component

* The fixed components are the components (or) the variables given in the program.
* In this algorithm result, count and size are the three different fixed components
* The variable component is a value which decides the final size of a number list. In this algorithm, if the size is taken as "n"
* The total space complexity required by a numbers variable will be 'n' if the maximum size is taken as 'n'.
* So space complexity = "3+n"

For calculating Time complexity :-

* The time taken by the program totally for compiling & execution is given as time complexity. So time = fixed time + instance time

Statement	Execution	Frequency	T T
1. Algorithm sum(number, size)	0	—	0
2. {	0	—	0
3. result = 0.0;	1	1	1
4. for count = 1 to size do	1	size+1	size+1
5. result = result + number [count];	1	size	size
6. return result;	1	1	1
7. }	0	—	0
			3+2size

Frequency is the total no. of times that each statement repeats

∴ The time complexity = 2size + 3

* Code for Searching a key

```
while (head != NULL)
{if (head → current == key)
{printf("key found");
}
head = head → next;
{
printf("key not found");
}
}
```

UNIT-2

STACKS AND QUEUES

Stacks:- Stack is a linear data structure in which all the operations such as insertion, deletion takes place at only one end called "TOP". Thus it is called Restrictive Data Structure.

* In this stack the insertion operation is called PUSH operation & Deletion operation is called POP operation.

* This follow a principle FILO[First in Last out] (or) LIFO[Last in First out].

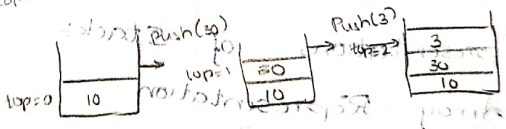
* Example:- The common examples of stack are
  1. Placing the books in the shelf.
  2. Bread Slices.
  3. Moving out of the bench.
  4.

Basic Operations of Stack are :-
1. PUSH operation
2. POP operation
3. Stack top Operation or Display operation.

PUSH Operation:-

Code :-

```
if (top == max-1)
{
printf("overflow");
}
else
{
 top++;
 st[top]=x;
}
```



*At a time only one element can be inserted (or) deleted from the stack.