Data structure :-

Data structure :-

A Data structure is a way of organising of storing Data in a computer, so that it can be accessed used efficiently. It refers to the logical or mathematical representation of Data as well as implementation.

Basic operations of Data structure :

1) Creation of Data
2) Insertion of Data
3) Modify Data
4) Delete Data
5) Searching
6) sorting

Types of Data structure :-

If we want to store the Data in sequential order is called Linear Data structure.

If we want to store the Data in non sequential order is called non-Linear Data structure.

Classification of Data structures :-

There are two types of Data structes

(1) Linear Data structures
(2) Non-Linear Data structures.

Linear Data structure :-

You can store the Data in sequential order (or) store continous memory allocations is called Linear Data structures.

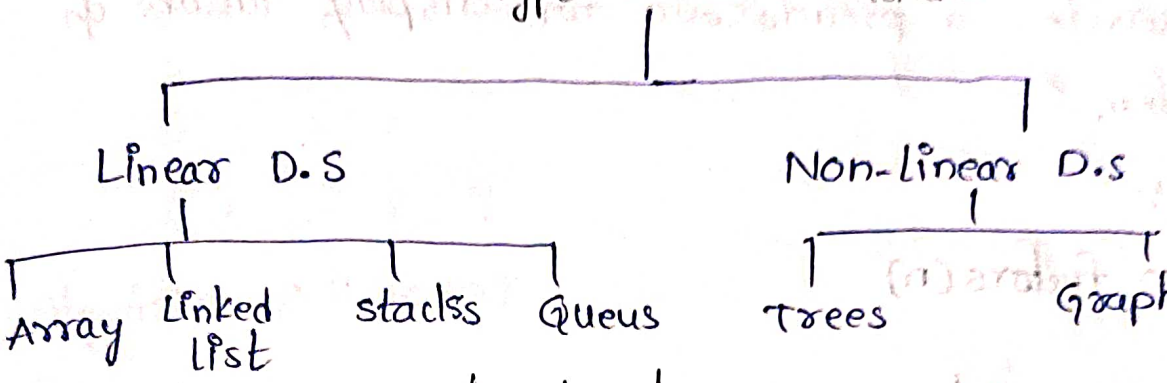
Examples :- Arrays, Linked list, stacks, queus.

Non Linear Data structure :-

You can store the Data in non-sequential order is called Non-Linear Data structure.

Examples:- Trees, Graphs.

Types of Data structures



Applications of Data structure:-

(1) Creating the Data Base.
(2) Creating Operating system.
(3) Creating computer graphics.
(4) Artificial Intelligence (AI).

Advantages of Data structure:-

1. Time complexity & space complexity should be less, it has more efficiency.
2. Flexibility (access any manner)
3. Re-usability
4. maintainability

27/09/2023

Algorithm :-

step by step process of a given problem.

⇒ Most of the cases Algorithm written in General English (User understandable language).

Pseudo Code :-

Pseudo code is almost same as Algorithm, it can ~~represent~~ written in mathematic process and logical representation.

1. algorithm name of algorithm (a, n)
2. well defined control statements
3. Input & output statements

4) Operators
5) return.

Example:- write a pseudocode for display factors of given number.

Pseudo code:-

```
Algorithm factors(n)
{
  // n is the integer
for (i<- 1 to n)
  if (n%.i=0)
   Print (i);
}
```

② write an Algorithm to print given number is prime (or) not.

Pseudo code:-

flag ① - prime
flag 0 - not prime

```
Algorithm prime(n)
{
 flag ← 1;
 for i ← 2 to n/2
   if (n%.i==0)
     {
       flag ← 0;
        break;
     }
  if (flag)
   print ("prime");
  else
   print ("not prime");
}
```

(3) Write an Algorithm to print following pyramid (4)

```
1
2 3
3 4 5
4 5 6 7
5 6 7 8 9
```

```
Algorithm pattern(n)
{
for i ← 1 to n
 {for j ← i to (2*i)-1 ;
    print (" ", j);
   print ("\n");
 }
}
```

(4) Write an Algorithm to print the following pyramid. (pascal triangle) 30/9/2023

```
      1
    1   1
  1   2   1
1   3   3   1
1 4   6   4 1
```

```
Algorithm pascal-triangle(n)
{
 for i ← 0 to n-1
   {
    for j ← 0 to i
     {
      element = fact(i) /(fact(i-j) * fact(j))
       print (" ", element);
     }
    print ("\n");
   }
}
```

```
Algorithm fact(x)
{
    product = 1,
    for i ← 1 to x/1.
        product = product i,
    return(product);
}
```

Linear Data structures :-

The Data stored in sequential order is called Linear Data structure.

⇒ There are four types of Linear Data structures :-

(i) Arrays
(ii) Linked list
(iii) stacks
(iv) Ques.

Arrays :- Collection of homogenous (or) similar data elements is called an Array.

Advantages of arrays :-

⇒ It is very flexible.
⇒ It will take the less time.
⇒ It can easily travels by using Index values.
⇒ Implementation of multidemensional elements is easy.

Disadvantage of Arrays :-

Arrays are fixed size memory allocations.

Syntax :- storage class datatype arrayname[index].

⇒ auto int a[10]

auto is default storage class.

Write an Algorithm to create an array with (6)
size 'n' and accepts n no. of elements to that array. Display
some of array elements.

```
Algorithm sum_array (n)
  //size of array.
    int a[n],sum=0;
  print ("array elements");
   for (i←0 ton)
    {
      read(a[i])
     sum+=a[i];
    }
  print ("sum is", sum);
}
```

write an algorithm to sort array of n Integers.

```
Algorithm sort array(a,n)        ← //a is an array
 int ~~a[i],i,j;~~                  //n is an size
 for i←0 to n-2
 { for j←i+1 ton-1
    {
      if (a[i]>a[j])
        {
          t=a[i];
          a[i]=a[j];
          a[j]=t;
        }
    }
}
```

Divide & Conquer :-

(1) Quick sort / partition exchange sort

(2) Merge sort.

Quick sort :-

0	1	2	3	4	5	6	7	8	
pv [25]	35	8	45	11	2	80	4	6	← stop.
25	6	8	45	11	2	80	4	35	← stop
25	6	8	4	11	2	80	45	35	
2	6	8	4	11	[25]	80	45	35	

Quick sort :-

Quick sort is the Divide & Concer in this entire list divide into 3 sub list :-

1. Left list
2. Sub list
3. Right List.

⇒ the basic condition for dividing elements is all the left list values must less than or equal to mid list.

⇒ All the right list values greater than mid list.

⇒ Quick sort also named as partition exchange sort.

Recursive Algorithm for quick sort :-

```
Algorithm Quick sort (a, lb, ub)
{
    // a - array of unsorted data
    // lb, ub are boundaries of given array.
  if (lb < ub)
  {
    sp = partition (a, lb, ub);
  }
```

```
Quick (a, lb, sp-1);
Quick (a, sp+1, ub);
}
}
```

Algorithm partition (a, lb, ub).

```
{
  p = a[lb];
  i = lb+1;
  j = ub;
while (j >= i)
{
  while (a[i] < p)
      i++;
  while (a[j] > p)
      j--;
  if (j > i)
  {
  // swap (a[i], a[j]);
    t = a[i]
    a[i] = a[j]
    a[j] = t;
  }
}
t = a[lb]
a[lb] = a[j]
a[j] = t
return j;
}
```

Quick sort program :-

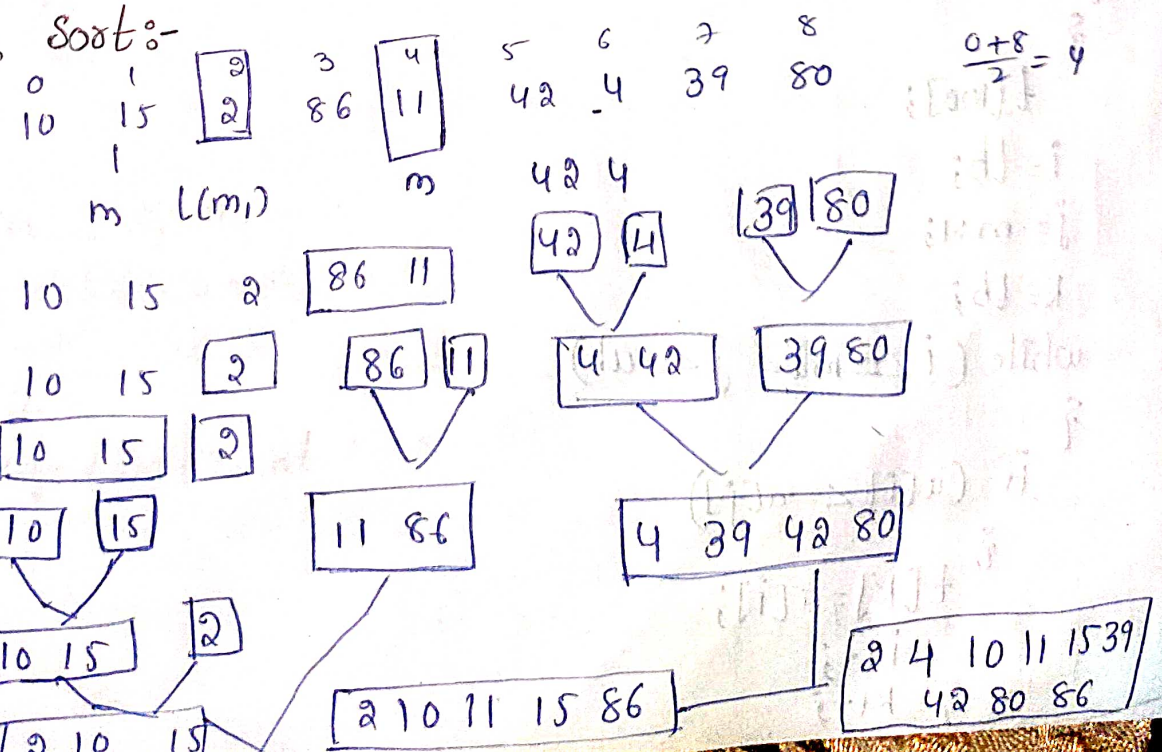
(9) 4/10/2023

```
#include <stdio.h>
main()
{
 void quick(int[], int, int);
 int partition(int[], int, int);
 int a[100], n, i;
 printf("enter size of array");
 scanf("%d", &n);
 Printf("enter array element");
  for(i=0; i<n; i++)
  {
   scanf("%d", &a[i]);
  }
  quick(a, 0, n-1);
   printf("sorted data is \n");
   for(i=0; i<n; i++)
     printf("%4d", a[i]);
  }
void quick(int a[100], int lb, int ub)
 { int sp;
   if(lb<ub)
   {
     sp= partition(a, lb, ub);
     quick(a, lb, sp-1);
     quick(a, sp+1, ub);
   }
 }
int partition(int a[100], int lb, int ub)
{ int i, j, p, t;
```

```
i = lb+1;
j = ub;
p = a[lb];
while (j >= i)
{
  while (a[i] < p)
      i++;
  while (a[j] > p)
      j--;
  if (j > i)
  {
    t = a[i];
    a[i] = a[j];
    a[j] = t;
  }
}
t = a[lb];
a[lb] = a[j];
a[j] = t;
return (j);
}
```

Merging Sort:-

Ex:-



```
quick (a, lb, ub)
{
  if (lb<ub)
  {
    m=(lb+ub)/2;
  quick sort (a, lb, m);
  quick sort (a, m+1, ub);
  merge (a, lb, m, m+1, ub);
}
```

5/10/2023

Algorithm Merge sort :-

```
Algorithm merge sort (a, lb, ub)
{
  if (lb<ub)
    {
      m= (lb+ub)/2
    mergesort (a, lb, m);
    mergesort (a, m+1, ub);
    mergesort (a, lb, m, ub);
  }
}
Algorithm merge (a, lb, m, ub)
  {
    t[100];
  i= lb;
  j= m+1;
  k= lb;
while (i<=m && j<=ub)
{
  if (a[i]<=a[j])
    {
      t[k]=a[i];
        i++;
    } k++;
```

```
else
 { t[k]=a[j];
     j++;
     k++;
 }
}
While (i<=m)
 {
  t[k]= a[i];
     i++;
     k++;
 }
 while (j<=ub)
   {
     t[k]= a[j];
     j++;
     k++;
   }

for (i=lb, i<=ub, i++)
 {
    a[i]= t[i]
  }
}
```

Merge sort program :-

```
#include <stdio.h>
 main()
 {
 void merge (int[], int, int);
```

Linked list:-

A linked list is a linear data structure that stores collection of data elements dynamically.

⇒ In linked list we represent all the data as nodes.

⇒ Each node contains two parts one is data: It can store required data of node and other one is address part. It can store the address of next node.

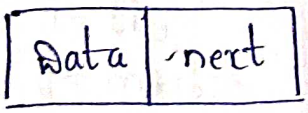
⇒ If there is no next node it can store the null point.

NOTE:-
For creating nodes of any linked list we need to use self refrential structure.

Self refrential structures:-
One of the structure memeber represents same structure that structures are called self referential structures.

Ex:- struct student
```
{
int no;
char name[100];
struct student *next;
}
```

NODE STRUCTURE:-




```
struct node
{
int data;
struct node *next;
}
typedf struct node Node; // — upto this only
Node *t;
```

t normal we use .
t.data (for normal)

t → data

memory allocation for Node :-

By using malloc statement we can allocate the memory for linked list nodes.

⇒ The prototype of malloc is <malloc.h>

⇒ syntax of malloc is pv= (casting *) malloc (sizeof (datatype))

```
int *p;
p=(int *) malloc (10* sizeof (int));
```

⇒ Raw bytes converted into desired data type using type casting

Example :-

```
Node *temp;
temp= (Node *) malloc( size of (Node));
```

Types of Linked List :-

there are 3 types of linked list.

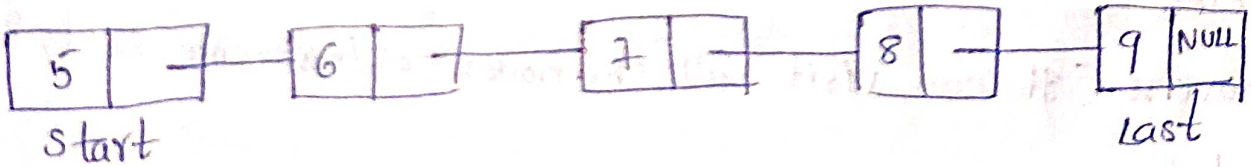
(1) single linked list.

(2) Double linked list :-

(3) Circular linked list (OR) circular single/double linked list.

Single Linked list :-

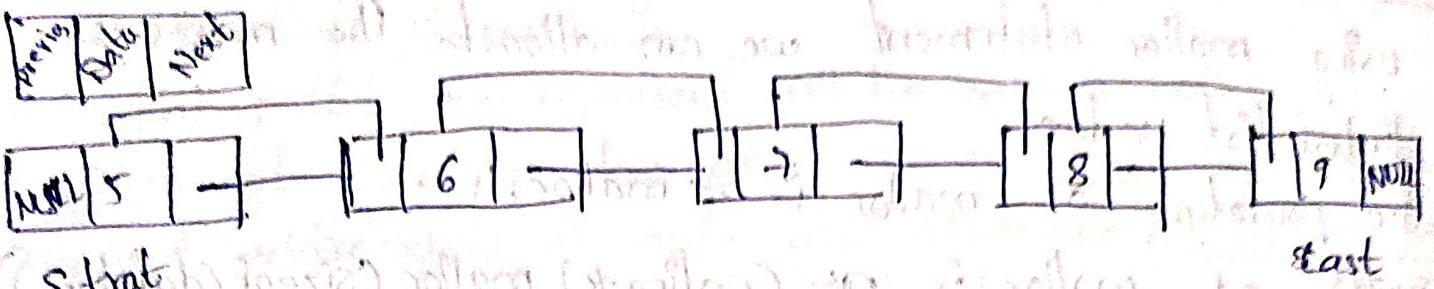
A single linked list is a common (or) standard type of list each In this list each node contains one data part and one address part. (line touch means address)



Double linked list :-

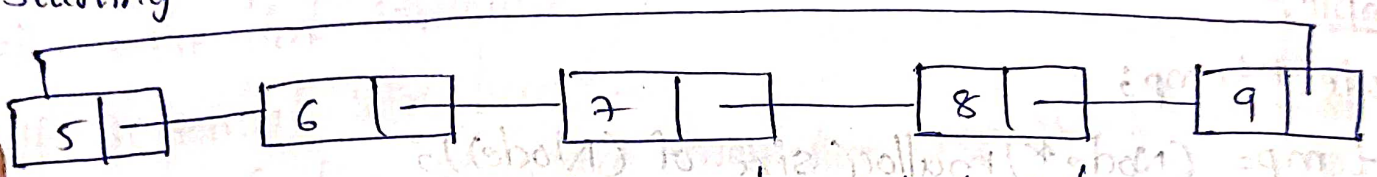
In double linked list there are two pointers and one data part; one pointer stores the address of previous node another pointer stores the address of next node.

If there is no previous node (or) next node it can store the null pointer.

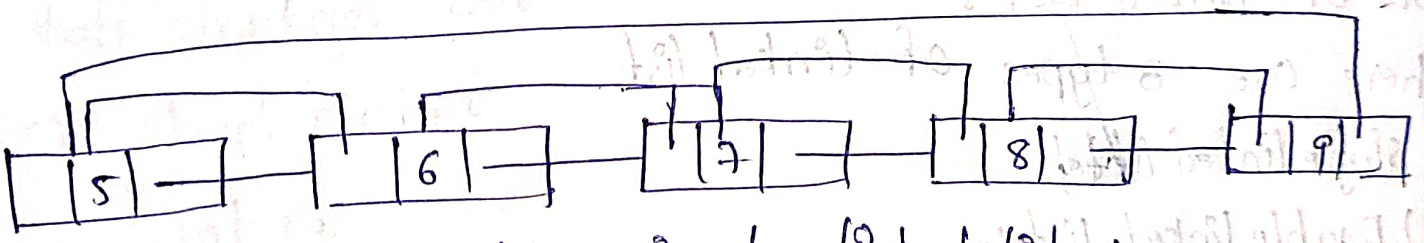


Circular Linked List :-

Circular linked list is extremely same as single linked list (or) double linked list the only difference is there is no starting node (or) ending node.



Single Circular linked List



double Circular linked list

Essential operations on linked list :-

⇒ There are 5 essential operations

(1) Create
(2) Insert
(3) Delete
(4) Traverse (It can visit all the nodes atleast once in its lifetime)
(5) sorted
(6) search

Implementation of single linked list :-

⇒ Node representation :-

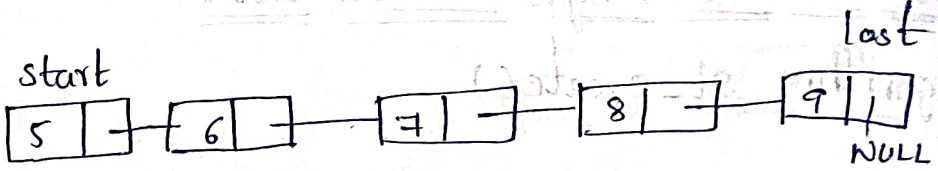
let us consider every node contains one data part as data and one address part as next, to store the address of next field.

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

Algorithm of Single Linked List :-

```
Algorithm sl_create()
{
 // start and last are two global identifiers initially you can store the null pointers.
 // slnode is user defined datatype it can represent the node structure.
 // temp is an identifier of type slnode, it can uses for temporary process.
 slnode *temp;
  int val;
while (1)
 {
  Printf ("enter value for node: if it is 0 end of list");
    read(val);
  if (val==0)
     break;
  else
  { temp=(slnode*) malloc(size of (slnode));
```

```
    temp->data = val;
     temp-> next= NULL;
    if (start= NULL)
     {
       start= temp;
        last= temp;
     }
    else
     {
      last -> next=temp;
       last= temp;
     }
  }
 }
}
```



~~temp=start;~~

~~ext~~

Algorithm sl_display()

{

// start and last are too global identifiers it represents first and last nodes of linked list.

// temp is an identifier of type slnode, it can uses for temporary purpose.

```
 temp=start;
 if (temp= NULL)
  {
   print (" list is empty");
  }
else
 {
 print ("list elements are");
 while (temp != NULL)
```

```
    print (temp -> data);
    temp = temp -> next;
  }
 }
}
```

Insert element into a Linked List :- 7/10/2022

```
Algorithm insert (key,)
{
// start and last are two global identifiers it represent
   first & last nodes of Linked list.
// You can insert the new node before the key value
   node if key is not identified in list, insert after
   the last node.
// temp is an identifier of type slnode, it can uses for
                                        temporary purpose.
 slnode, *temp, *prev, *current;
 temp= (slnode*) malloc (sizeof (slnode));
 print ("enter value for new node");
  read (temp->data);
  temp-> next= NULL;
   Current = previ= start;
while (current->data != key && current-> next != NULL);
   {
     prev= current;
    current = current -> next;
   }
if (Current = start)
{
    temp -> next = start;
  } start = temp;
```

```
else
    if (current == last && current->data != key)
    {
        Last->next = temp;
        Last = temp;
    }
else
    {
        prev->next = temp;
        temp->next = current;
    }
}
```

Delete element into a linked List :- 9/10/2023.

Algorithm delete(key)

// start & last are two global identifiers it represents first and last nodes of linked list.

// key

// temp is an identifier of type slnode.

```
slnode *prev, *current;
    prev = current = start;
    while (current->data != key && current->next != NULL)
    {
        prev = current;
        current = current->next;
    }
if (current == start) // remove first node
{
    start = start->next;
    current->next = NULL;
```

```
    free (current);
  }
else if (current == last && current -> data != key) // element not found
    printf ("element not found");
  else
    if (current == last) // last node is removed.
    {
      last = prev;
      last -> next = NULL;
      free (current);
    }
else  // remove middle node.
  {
    prev -> next = current -> next;
    current -> next = NULL;
    free (current);
  }
```

Program of single linked list.

```
#include <stdio.h>
# include < malloc.h>
 struct node
 {
   int data;
   struct node *next;
 };
 typedef struct node slnode;
  slnode *start = NULL, *last = NULL;
 void main ( )
  {
    int ch; key;   // ch = choice
    void create ();
    void insert(int);
```

```
void sl_node(int);
void search(int);
void display();

do
{
printf("\n\t\t MENU \n 1.Create \n 2. Insert \n");
printf ("3. Delete \n 4. Display \n 5. Search \n 6. Exit\n");
printf ("Enter your choice\n");
                                                    10/10/23
Scanf ("%d", &ch);
Switch (ch)
{
 Case 1: Create ();
         break;
 Case 2: Insert printf ("Enter which node before you have to Insert");
         Scanf ("%d", &key);
         insert (key);
          break;
Case 3: printf ("Enter which node to be delete");
          scanf ("%d", &key);
       sl_delete (key);
          break;
Case 4: display ();
         break;
Case 5: printf ("Enter element to be search");
          Scanf ("%d", &key);
            search (key);
              break;
}
```

```
} while (ch != 6);
}
void Create ()
{
  Slnode *temp;
  int val;
  While (1)
  {
  printf ("Enter value for new node");
  Scanf ("%d", &val);
  if (val == 0)
     break;
  else
   {
    temp= (slnode *) malloc (size of (sl node));
     temp→data= val;
     temp→ next= NULL;
  if (start==NULL)
   {
    start= temp;
    last = temp;
   }
   else
    {
     last → next= temp;
     last=temp;
    }
  }
 }
}
void insert (int key)
{
  slnode *cur, *prev, *temp;
```

```
temp=(slnode *)malloc(sizeof(slnode));
printf("enter value to be inserted node");
scanf("%d", &temp->data);
cur = start;
prev= start;
while (cur->data != key && cur->next != NULL)
{
  pre = cur;
  cur = cur->next;
}
if (cur == start)
{
 temp -> next=start;
  start =temp;
}
else if (cur==NULL && cur->data != key)
   {
     last->next=temp;
     last = temp;
else
   {
    prev-> next=temp;
     temp->next=cur;
   }
}
void sl_delete(int key)
{
 sl node *prev,*cur;
 while prev=start;
       cur= start;
```

```
while (current->data != key && current->next != NULL)
  {
    prev = cur;
    cur = cur->next;
  }
if (current == start)
  {
    start = start->next;
    cur->next = NULL;
    free (cur);
  }
else if (current == last && cur->data != key)
  printf ("element not found");
else if (cur == last)
  {
    last = prev;
    last->next = NULL;
    free (cur);
  }
else
  {
    prev->next = cur->next;
    cur->next = NULL;
    free (cur);
  }
}
void sldisplay()
{
  slnode *temp;
  temp = start;
if (temp == NULL)
```

```
{
 printf ("list is empty");
 }
else
 {
 printf ("In list data is:");
 while (temp!=NULL)
   {
   printf ("%d", temp->data);
    temp = temp->next;
   }
 }
}
void search (int key)
 {
  slnode *temp;
   temp=start;
 while (temp!=NULL)
  {
   if (temp->data==key)
    {
     printf ("element found");
       break;
    }
   temp = temp-> next;
  }
 if (temp==NULL)
    printf ("element not found");
 }
```

Double Linked List :-

In double linked list, it contains two pointers and one data part. It is a linear data structure.

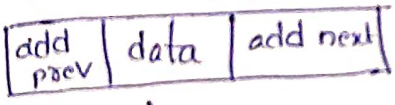
⇒ It process almost same as single linked list here only 1 difference double linked list stores the address of previous node & address of next node.

⇒ The main advantage of double linked list is it can easily process forward & backward traversing.

Operations on Double Linked List :- They are 5 types of operations:

1) Create
2) Insert
3) Delete
4) Display
5) Traverse (Backward & Forward)

Node structure :-



dl Node

```
Struct node
{
   int data;
  struct node *next, *prev;
};
typedef struct node dlnode;
```

Algorithm for Double Linked list (Create Algorithm)

```
Algroithm dl_create()
{
// start & last are two global identifies of dlnode.
   dl_node. which holds first & Last nodes.
```

// temp is an identifier of dl-node.

```
dl node * temp;
int val;
While(1)
{
printf("Enter value to be Insert if zero exit");
 read (val);
 if (Val==0)
    break;
else
  {
  temp = (dl node*) malloc(size of (dl node));
  temp →data=val;
   temp→prev = temp→next=NULL;
 if (start ==NULL)
  {
   start =last=temp;
  }
 else
  {
   last→next=temp;
   temp→prev= last;
   last=temp;
  }
}
}
```

Display Algorithm :-

Algorithm dL_node display( )

{

// start and last are two global identifiers of type dL_node, which identifies first & last nodes.

// temp is an identifier of type dL_node.

```
temp=start;
if (temp=NULL)
 {
  print ("List is empty");
 }
else
 {
```

print ("List elements are"); // Forward data of double linked list is

```
  temp=start;
  while (temp!=NULL)
  print (" temp→data);
  temp = temp→next;
 }
print (" Back traverse data of double linked list is");
  temp= last;
  while (temp!=NULL)
  print (" temp→data);
  temp= temp→prev;
}
```

Search Algorithm :-

Algorithm dL_node search (key)

{

// start & last are two identifiers of type dL_node which identifies first & last node.

```
temp=start;
while (temp!=NULL)
 {
  if (temp->data==key)
   {
    printf ("element found");
      break;
   }
 temp= temp->next;
 }
if (temp==NULL)
 print ("element not found");
}
```

Algorithm to Insert Node into Double linked list:-

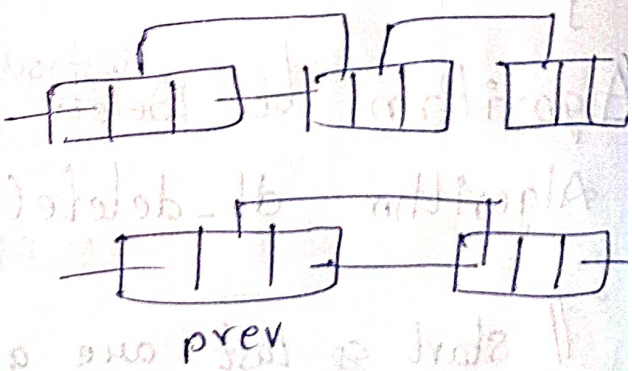
```
Algorithm dl-insert (key)
{
 // start & last are global identifiers which is used to
                identify first& Last nodes;
 // temp
  dlnode,* prev,*cur,*temp;
  temp = (dl node*) malloc (sizeof (dlnode));
  print ("enter value for new node");
read (temp-> data);
  temp ->next = temp ->prev= NULL
   prev=cur= start
 while ( cur
        temp ->data != key && curr ->next != NULL)
 {
    prev=cur;
    cur = cur -> next;
 }
```

```
if (current == start)
{
  current -> prev = temp;
    temp -> next = cur;
     start = temp;
}
else if (curr = last && cur->data != key)
  {
    cur -> next = temp;
     temp -> prev = cur;
        last = temp;
    }
else
  {
    prev -> next = temp;
    temp -> pre = prev;
    temp -> next = curr;
    cur -> pre = temp;
  }
}
```

Algorithm to Delete node from Double linked List:-

```
Algorithm dl_delete(key)
{
// start & last are global identifiers which is used to
identify first & last nodes.
  dlnode, * prev, * cur, //temp;
   temp = (dl node *) malloc(size of (dl node));
   print    prev = start;
            cur = start;
```

```
while (current -> data != key && cur -> next != NULL)
{
    prev = cur;
    cur = cur -> next;
}
if (cur == start)
{
start = start -> next;
    start -> pre = NULL;
    free (curr);
}
else if (curr == last && cur -> data != key).
print ("Element not found");
else if (curr = last)
{
    last = prev;
    last -> next = NULL;
    free (cur);
}
else
{
    prev -> next = cur -> next;
    cur -> next -> prev = prev;
    free(cur);
}
}
```



Circular linked list :-

Circular linked list is a linear data structure, is almost same as single linked list, here the only one difference is last node address pointer stores the address of first node.

⇒ Double circular linked list :-

Double circular linked list is a linear data structure, it almost same as double linked list, here the only difference is first node previous pointer holds the address of last node, Last node previous pointer holds the address of the first node.

12/10/2023

Method 1 :- (One pointer)

Creation of Circular linked list using single pointer.

Algorithm csl_create()

```
{
// start is a global node of data type slnode,
it can identifies the first node of single circular
linked list. Initially its value is NULL.

  slnode *temp;
  temp=(slnode*)malloc(sizeof(slnode));
  printf("enter value for new node");
  read(temp->data);
if (start==NULL)  // insert it as first node.
 {
  start=temp;
  temp->next=start;
 }
else      // identify last node.
 {
  last = start;
  while (last->next != start)
```

```
last = last -> next;
last -> next = temp;
temp -> next = start;
```

(OR)

```
temp -> next = last -> next;
last -> next = temp;
```

}

Method 2:- (2 pointers)

Creation of single circular linked list using two pointers.

Algorithm scl_create()

{

// start and last are two identifies of data type of sl_node, it can identifies first & last nodes of single circular linked list. Initially these two are NULL.

```
slnode *temp
temp = (slnode*) malloc( sizeof(slnode));
print("enter value of new node");
read(temp -> data);
if(start == NULL)
  { start = last = temp;
  } temp -> next = start;
else
  {
    last -> next = temp;
    last = temp;
    tast -> next = start;
  }
```

}

Display Algorithm for Single pointer circular linked list

Algorithm csl_display()

{

// start is a identifier of data type of SL node, it can identifies first node..

if (start = NULL)

print ("list is empty");

else

{

temp = start;

do

{ print (temp → data);

temp = temp → next;

} while (temp != start);

}

}

Display single circular linked list algorithm using two pointers

Algorithm scl_display()

{// start is a identifier of data type

if(start = NULL)

print ("list is empty");

else

{ temp = start;

do

{ print (temp → data);

temp = temp → next;

} } } while (temp != start);

insert & delete try.

Stack:-

Stack is a linear data structure it can process all the insertions & deletions at one end is called top.

⇒ stacks can follow the property of last in first out (LIFO) [or] first in last out [FILO].

Applications:-

1) Conversion of expression.
2) Evaluation of expressions.
3) Implementing function or procedure
4) Solve the problem of tower of funnel.

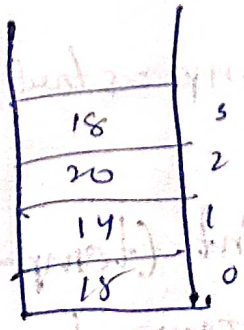
Stack Terminology :-

⇒ s represents stack with maximum size of 100 (Size(100]).
⇒ size identifies maximum locations used in this process.
⇒ top represents top of the stack.
⇒ Overflow - if stack is full we can write it as overflow.
⇒ Underflow - if stack is empty we can represents underflow
⇒ push() — Insert element into the stack.
⇒ pop() — we can delete element from stack.

Operations on stack:-

There are three types of operations

(i) push
(ii) pop
(iii) Display



Push:-

Push Algorithm :-

```
Algorithm push (ele)
{ // If s is static array represents stack of size, size.
  // top is a global identifier it represents top of stack.
if (top == size)
  print(" Stack is overflow");
else
  {
    s [top++] = ele;
    print ("element pushed into the stack");
  }
```

Pop :-

```
Algorithm pop ()
{
  // s is static array represents stack of size, size.
  // top is global indentifier it represents top of stack.
  if (top = 0)
  print (" stack underflow");
  else
    {
      top--; // removes top.
    return (s[top]);
    }
}
```

Display :-

```
Algorithm display()
{//
```

```
if (top=0)
 print ("stack underflow");
else
  {
  for (i=top-1;i>=0; i--)
  print (s[i]);
  }
}
```

13/10/2023

Program using arrays :-

```
#include <stdio.h>
int s[100], top=0;size=5;
void main()
 {
  void push (int);
  int pop();
  void disp();
  int ch, val;

do
 {
printf("\n\t\t MENU\n 1. PUSH\n 2. POP \n 3. DISPLAY
        \n 4. EXIT \n");
printf ("Enter one option from above menu \n");
scanf ("%d", &ch);
switch (ch)
{
 Case 1: printf ("Enter value to be push\n");
         Scanf ("%d", &val);
         push (val);
         breaks;
```

```
case 2: val = pop();  → if (val == -1) printf("List is empty\n");
                      → else printf("deleted element is %d", val);
        break;
case 3: disp();
        break;
}
} while (ch != 4);
}
void push (int ele)
{
  if (size == top)
    printf ("stack is overflow\n");
  else
    {
    s[top++] = ele;
    printf ("element is inserted");
  }
}
int pop()
{
  if (size == top == 0)
    return (-1);
  else
    {
      top--;
    return (s[top]);
  }
}
void disp()
{ int i;
  if (size == 0)
    printf ("stack is empty \n");
  else
    {
      printf ("stack data is \n");
```

```
    for (int i=top-1; i>=0; i--)
      printf("%d ", s[i]);
  }
}
```

Program of stack using List :-

```
#include<stdio.h>
#include<malloc.h>
 struct node
   { int data;
     struct node *next;
 };
 typedef struct node strode;
  stnode *top=NULL, *temp;
void main()
 {
   void push(int);
    int pop();
    void disp();
    int ch,val;
 do
{
 printf("\n\t\t MENU\n 1. PUSH \n 2. POP \n 3. DISPLAY
        \n 4. EXIT \n");
 printf("Enter one option from above menu\n");
 scanf("%d",&ch);
 switch(ch)
{
   case1: printf("Enter value to be push");
          scanf("%d",& val);
          push(val);
          break;
```

```
case 2: val= pop();
        if (val == -1)
        printf("list is empty");
    else
     printf("deleted element is %d", val);
            break;

case 3:  disp();
            break;

  }

}while (ch!=4);

}
void push (int ele)
{
  temp = (stnode*) malloc (size of (stnode));
   temp -> data = ele;
   temp -> next = NULL;
   if (top == NULL)
      top=temp;
   else
    {
       temp-> next = temp;
        top= temp;
    printf ("element is inserted");
   }
}
int pop()
{ int x;
  temp= top;
  if (top==NULL)
   return (-1);
else
  {
     x= temp -> data;
```

```
        top = top->next;
        free(temp);
        return(x);
    }
}
void disp()
{
    int i;  temp=top;
    if      if(top==NULL)
        printf("stack is empty\n");
    else
    {
        printf("stack data is\n");
        while(temp!=NULL)
        {
            printf("%d", temp->data);
            temp=temp->next;
        }
    }
}
```

17/10/2023

Queues :-

Queue is a linear data structure in this all insertions done at rear positions and delete the element from front position. It follows the property of first in first out (fifo).

Unable to all ready processed location drawback

Applications :-

1. Ready queue in computer processor.
2. Graph traversal.
3. Identify the path b/n source & destination.

Terminology :- peak :- Identify the front

1. Front :- Identify the queue front location.
2. rear :- Identifies last position of the queue.
3. Quefull & Queueempty.
4. Insert :- Insert element into the queue (nqueue)
5. Delete :- Delete element from queue. (dequeue)
6. Traversal :- Display the content from front to real.

** peak returns the queues front location value. (2)

Alogorithm of Insert :-

Algorithm insert (ele)
{
// "q" is global ~~identifier~~ array with the size of ~~queue~~ size. and rear is also global variable represents ↓ last position of the queue.
current

```
if (rear == size)
  {
  printf (" queue is full");
  }
else
  {
    q [rear++] = ele ;
  printf (" element inserted");
  }
}
```

18/10/2023

Algorithm Delete :-

q | 5 | 6 | 7 | 8 |

front
rear

```
Algorithm q_delete ( )
{
  if (front == rear)
    return (-1);
```

```
else
 {
    element = q [front];
       front ++;
    return (ele);
  }
}
```

Algorithm Display :-

```
Algorithm display ( )
{
  if (front == rear)
   print ("queue is empty");
  else
    { print (" Queue data is : ");
      for (i = front; i < rear; i++)
          print (q[i]);
    }
}
```

Queue Implementation using Linked list :-

```
struct node
{ int data;
  struct node *next;
};
typedef struct node qnode;
Algorithm insert_qlist (ele)
{
  // front & rear are two pointers to identify qfront
  & qrear Initially these pointer values are NULL.
```

```
qnode *temp;
 temp = (qnode *) malloc (sizeof (qnode));
 temp ->data = ele;
 temp ->next =NULL;
if (rear == NULL)
{
  rear = front = temp;
 }
else
  {
   rear ->next =temp;
      rear=temp;
   }
 }
```

Algorithm delete_qlist()

```
{
 if (front == NULL)
     return (-1);
  else
     {
      temp = front;
      front = front ->next;
       temp -> next = NULL;
    ele = temp ->data;
      free (temp);
    return (ele);
  }
}
```

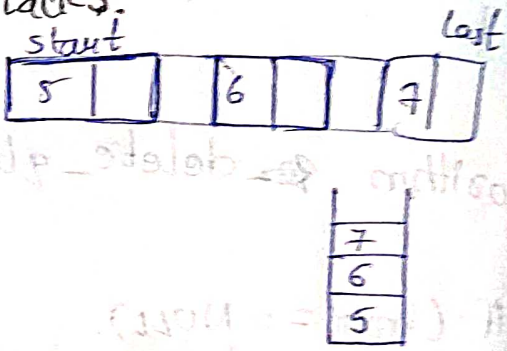
Algorithm display_qlist()

```
{
   if (front==NULL)
     print ("q is empty");
```

```
else
 {
   temp = front;
     while (temp != NULL)
   {
    print ("temp -> data");
     temp = temp -> next;
    }
 }
}
```

Applications of stack:-

① write an Algorithm to display single linked list data in Inverse order using stacks.




```
Algorithm display_stack( )
{
 // start & last   (wrong)
   temp=start;
```

```
Algorithm display SLlist_reverse( )
{
 // Assume that single linked list already existed represented
    by start & last
     int s[100], top=0;
     Snode *top;
      temp = start;
     while (temp != NULL)
     {
       s[top++] = temp->data;
         temp = temp->next;
      }
    for (i=top-1; i>=0; i--)
    }  printf (s[i]);
```

(8) Convert infix notation to postfix notation.

Every string must write to stack top operator

least — push
" — same pop

19/10/2023

a + b * c \0

S.No	ch	stack	o/p
1.	a	#	a
2.	+	# +	a
3.	b	# +	ab
4.	*	# + *	ab
5.	c	# + *	abc
6.	\0	pop()	abc * + \0

a * b + c \0

1.	a	#	a
2.	*	# *	a
3.	b	# *	ab
4.	+	pop()	ab *
5.	-	push # +	ab *
5.	c	# +	ab * c
6.	\0	pop	ab * c +

- 0
C - 1
-, + - 2
*, / - 3

⇒ C — push into stack,) — pop all characters from stack until "C"
open brace

Rules for Convert Infix Expressions to Postfix Expressions:-

1. Print operands as they arrive.
2. If the stack is empty (or) contains a left paranthesis on the top push the incoming operator into the stack.
3. If the incoming symbol is left paran thesis push it on the stack.

(C)

4. If the the incoming symbol is right paranthesis pop the sta print the operators until you see a left paranthesis. Discord the pair of parathesis.

5. If incoming symbol has highest preceedence than the top of the stack then push it on to the stack.

6. If the incoming symbol has equal preceedence with top of the stack use association. (First in first come)

7. If the incoming symbol has lower preeedence than the symbol on the top of the stack pop the stack and print the top operator. Then test the incoming operator with new top of the stack.

8. At the end of the expression pop() and print all the operators from the stack.

9. Display the output.

Ex:- A*(B+C*D)+E

S.No	Char	stack	O/P
1	A	#	A
2.	*	#*	A
3	(	#*(	A
4.	B	#*(	AB
5	+	#*(+	AB
6.	C	#*(+	ABC
7.	*	#*(+*	ABC
8.	D	#*(+*	ABCD
9	)	#*(+*)	ABCD*+

#*(+) pop' pop() discarded

10	+	#+*	ABCD*+
11	E	#+*	ABCD*+*E
12	\0	#+	ABCD*+*E+
		pop*	

Ex:-2

(A+B*C)+(D*E+F)\0

S.No	Char	stack	o/p.
1.	(	#(	
2.	A	#(	A
3.	+	#(+	A
4.	B	#(+	AB
5.	*	#(+*	AB
6.	C	#(+*	ABC
7.	)	pop upto (	ABC*+
8.	+	#+	ABC*+
9.	(	#+(	ABC*+
10.	D	#+(	ABC*+D
11.	*	#+(*	ABC*+D
12	E	#+(*	ABC*+DE
13	+	#+(+	ABC*+DE*
14.	F	#+(+	ABC*+DE*F
15.	)	#+(+)	ABC*+DE*F+
		pop(+)	
16	\0	#+	ABC*+DE*F++

Algorithm to convert infix to postfix :-

```
Algorithm Infix_postfix
{
 read Infix;
 push (#);
 while ((ch = infix [i]) != '\0')
 {
  if (ch == '(')
    push (ch);
  else if (isalphanum (ch))
     postfix [k++] = ch;
  else if (ch == ")")
  {
    while (s [top] != '(')
      postfix [k++] = pop();
    elee = pop();
  }
  else
    {
      while (pr (ch) < pr (s [top]))
        postfix [k++] = pop();
      push (ch);
    }
 }
}
```