EXPERIMENT: 1

Write program to implement array operation

1. Find avg of 10 numbers using array.

→

```
#include <stdio.h>
int main(){
int num[10];
int i, sum=0;
float avg;

printf("Enter 10 numbers: ");

for(i=0; i<10; i++)
{ scanf("%d", &num[i]);
  sum += num[i];
}

avg = sum/10;

printf("Average of 10 numbers = %.2f", avg);

return 0;
}
```

OUTPUT

Enter 10 numbers: 45 78 38 98 47 69 0 7 9 34

Average of numbers = 42.00.

2. Display the following pattern

```
*
# #
* * *
# # # #
```

→

```
#include <stdio.h>
int main(){

int i, j;

for(i=1; i<=4; i++){
  for(j=1; j<=i; j++){

    if(i%2==0){
      printf("# ");
    }
    else {
      printf("* ");
    }
  }
  printf("\n");
}
return 0;
}
```

OUTPUT

```
*
# #
* * *
# # # #
```

3. Find first repeating element of array.

→

```
# include <stdio.h>

int main()
{
int arr[] = { 4, 2, 1, 3, 2, 5, 1 };
int n = sizeof(arr) / sizeof(arr[0]);
int found = 0;

for( i=0; i<n-1 ; i++) {

for (j = i+1 ; j<n ; j++ ) {

  if (arr[i] == arr[j]) {

printf (" First repeating element is : %d \n", arr[
found = 1;
break;
}
}

if (found){
  break;
  }
else {
printf (" No repeating element found. ");
  }
  return 0;
  }
```

OUTPUT

First repeating element is: 2

4. Find greatest and smallest element in array.

→

```
# include <stdio.h>
int main()
{
  int arr[] = { 45, 3, 88, 12, 78, 34, 23);
  int n = sizeof(arr) / sizeof (arr[0]);
  int max = arr[0];
  int min = arr[0];

 for(int i=1; i<n ; i++)
 {
   if (arr[i] > max) {
       max = arr[i];
     }
   if (arr[i] < min) {
      min = arr[i];
     }
  }
 printf ("Greatest element = %d ", max);
 printf ("Smallest element = %d ", min);

  return 0;
  }
```

OUTPUT

Greatest element = 88

Smallest element = 3.

5. write a program squaring the odd position elements

→

```
#include <stdio.h>
int main()
{
int arr[] = {7, 6, 9, 2, 8, 4, 3};
int n = sizeof(arr) / sizeof(arr[0]);
int i;

printf("Original array: \n");

for(i=0; i<n; i++)
{ printf("%d \n", arr[i]);
 }

for(i=1; i<n; i+=2) {
 arr[i] = arr[i] * arr[i];
 }

printf("New array: \n");
for(i=0; i<n; i++)
{
 printf("%d \n", arr[i]);
 }
 return 0;
 }
```

OUTPUT

Original array: 7, 6, 9, 2, 8, 4, 3.
New array: 7, 36, 9, 4, 8, 16, 3

* EXTRA QUESTIONS.

1) WAP to print the following pattern.

```
*
# #
$ $ $
? ? ? ?
```

→

```
#include <stdio.h>
int main()
{ int i, j;

for(i=1; i<=5; i++)
{ for(j=1; j<=i; j++)
{ switch(i) {
   case 1:
   printf("*");
   break;

   case 2:
   printf("#");
   break;

   case 3;
   printf("$");
   break;

   case 4;
   printf("?");
   break;
  } }
  printf("\n");
  }
  return 0;
  }
```

OUTPUT

```
*
# #
$ $ $ $
? ? ? ?
```

2
→

```
#include <stdio.h>
int main()
{
  int i,j;

for(i=1;i<=5;i++)
{ for(j=1;j<=5;j++)
  {
    printf("%i");
  }
  printf("\n");
  }
  return 0;
  }
```

OUTPOT

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

3.

```
#include <stdio.h>
int main() {

int i,j;

for(i=1;i<=5;i++)
{
for(j=1;j<=5;j++)
{
  printf("%d ",j);

  }
  printf("\n");
  }
  return 0;
  }
```

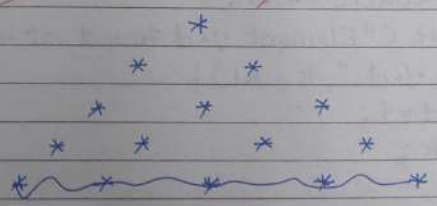
OUTPUT

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

4.
```
# include <stdio.h>
int main()
{
int i, j;

for(i = 4; i >= 1; i--)
{
for(j = 1; j <= 4-i; j++) {

  printf(" ");
}
for(j = 1; j <= 2*i-1; j++) {

  printf("*");
}
printf("\n");
}
return 0;
}
```

OUTPUT

5.
```
# include <stdio.h>
int main() {
int i, j;

for(i = 1; i <= 4; i++) {
 for(j = 1; j <= 4-i; j++)
{
 printf(" ");
}

for(j = 1; j <= i; j++) {

  printf("* ");
}
printf("\n");
}
return 0;
}
```



EXPERIMENT-2

1) Search data using linear search, consider following list.
56, 36, 89, 57, 1, 0, 67, 59.

→
```
#include <stdio.h>
int main()
{
int data[] = { 56, 36, 89, 57, 1, 0, 67, 59};
int k, i;
int size = sizeof(data) / sizeof(data[0]);
int found = 0;

printf("Enter the number to be searched:");
scanf("%d", &k);

for(i=0; i< size ; i++) {

if ( data[i] == k ){
printf("Element %d found at index
    %d ", k, i);
found = 1;
break;
}
}

if (k != found){
printf("Element not found");
}
return 0;
}
```

OUTPUT

```
Enter the element to be searched: 1
Element 1 found at index 4.

Eter Enter the element to be searched: 55
Element not found.
```

2) Search data using binary search.

```
#include <stdio.h>
int main(){

int m, k, l, h, i;
int arr[9];

for(i=0; i<9; i++)
{
printf("Enter element %d", i+1);
scanf("%d\n", &k);
}

printf("Enter the element to be searched: ");
scanf("%d", &k);

l=0, h=8;

while(l<=h){
 m=(l+h)/2;

 if(arr[m]==k){
  printf("element %d found at index
         %d", k, m);
  break;
 }
 else {
 if(arr[m]<k){
    l=m+1;
 }
 if(arr[m]>k){
    h=m-1;
 }
 }
}

if(l>h){
 printf("Element not found");
}
return 0;
}
```

OUTPUT

```
Enter element 1 : 1
"       "   2 : 2
            3 : 3
            4 : 4
            5 : 5
            6 : 6
            7 : 7
            8 : 8

Enter the element to be searched: 7
Element 7 found at index 6.
```

3) Compare linear and binary search.

→

LINEAR SEARCH	BINARY SEARCH
• It works on both unsorted and sorted arrays.	• It works only on sorted arrays.
• Its algorithm type is sequential.	• Its algorithm type is divide and conquer.
• It is simple to implement.	• It is more complex to implement.
• Time complexity is (n)	• Time complexity is (log n) ~~tons~~
• It is used when data is small or unsorted	• It is used when data is large & unsorted.

4) State limitations of linear search in terms of time complexity.

→ Linear search has limitations in terms of time complexity, especially for large data sets. Its worst case and avg. case time complexity is O(n) meaning it may need to scan every element in the array to determine if its present or not. It makes it slower than binary search which operates in O(log n) time on sorted data. In linear data doesn't need sorted arrays leading to poor performance as data size grows.

* EXTRA QUESTIONS

1) write a program to copy the elements of array into another array in a reverse order.

```
#include <stdio.h>
int main()
{
int original[100], reversed[100];
int i, n;

printf("Enter numbers of elements in array:");
scanf("%d", &n);

for(i=0; i<n; i++)
{
printf("Enter %d element:", i+1);
scanf("%d", &original[i]);
}

for(i=0; i<n; i++)
{
reversed[i] = original[n-i-1];
}
printf("Elements of reversed array:\n");
for(i=0; i<n; i++)
{
printf("%d\n", reversed[i]);
}
return 0;
}
```

OUTPUT

```
Enter number of elements in the array: 2
Enter 1 element: 4
Enter 2 element: 6

Elements of reversed array:
6
4.
```

(2) WAP in C to count total number of dupl. elements in array.

```
#include <stdio.h>
int main()
{
int arr[100], n, i, j, count = 0;

printf("Enter size of array: ");
scanf("%d", &n);

for(i=0; i<n; i++)
{printf("Enter element: ");
 scanf("%d", &arr[i]);
}

int visited[100] = {0};

for(i=0 ; i<n ; i++)
 {
  for(j=i+1 ; j<n ; j++)
   {
    if(a[i] ==a[j])
     {
      count ++;
      break;
     }
   }
 }
printf("\n Total Duplicate elements : %d", count);
 return 0;
}
```

OUTPUT

```
Enter size of array: 4
Element 1 : 2
Element 2 : 1
Element 3 : 1
Element 4 : 2

No Total Duplicate elements: 2
```

(3) WAP in C to print all unique elements in array.

```
# include <stdio.h>

int main() {
 int arr[100], n, i, j, count=0;

printf("Enter no of elements: ");
scanf("%d", &n);
for(i=0; i<n; i++) {
printf("Element %d", i+1);
scanf("%d", arr[i]);
}

for(i=0; i<n; i++)
{
 for(j=0; j<n; j++)
 {
   if (arr[i] == arr[j]) {
      count++;
   }
 }
printf("Unique elements in array are: ");
 if (count == 1) {
    printf("%d", arr[i]);
 }
 return 0;
}
```

OUTPUT

```
Enter no of elements: 6
Element 1 = 3
Element 2 = 5
Element 3 = 3
Element 4 = 2
Element 5 = 7
Element 6 = 5

Unique elements in array are:
2 7
```

(4) WAP to seperate odd and even integers in two seperate arrays.

```
#include <stdio.h>
int main()
{
int arr[10], arr1[ ], arr2[ ]
int i

for(i=0; i<10; i++)
{
  printf("Element %d = ", i+1);
  scanf("%d", &arr[10]);
  }

for(i=0; i<n; i++)
{
  if (arr[i] % 2 == 0)
    {
      arr[i] = arr1[int evencount++];
    }
  else
    {
      arr[i] = arr2[int oddcount++];
    }
  }

printf("Even numbers:");
 for (i=0; i<evencount; i++)
  {
   printf("%d ", arr1[i]);
  }

printf("Odd numbers:");
 for(i=0; i<oddcount; i++)
  {
    printf("%d", arr2[i]);
  }
 return 0;
}
```

OUTPUT

arr = [10, 21, 4, 45, 66, 93, 1, 9, 81, 12]

Even numbers: 10, 4, 66, 12
Odd numbers: 21, 45, 93, 1, 81, 9

(5) WAP to Find second smallest element in an array.

```
#include <stdio.h>
int main()
{
int arr[] = {10, 5, 8, 20, 2};
int n = 5;
int i,j, temp;

for(i=0;i<n;i++)
 {
  for(j=i+1;j<n;j++)
   {
    if(arr[i] > arr[j])
    {
     temp = arr[i];
     arr[i] = arr[j];
      arr[j] = temp;
    }
  }
}

for(i=1;i<n;i++)
{
  if(arr[i] != arr[0])
  {
   printf("second smallest element: %d",
                              arr[i]);
   return 0; break;
  }
else
 {
 printf("## No second smallest element");
 }
return 0;
}
```

OUTPUT

Second smallest element : 5

(6) WAP in C to count the total no of words in a string

```
#include <stdio.h>
int main()
{
 char str[100];
 int i, words = 1;
 printf("Enter a string: ");
 scanf("%[^\n]", str);

for(i=0; str[i] != '\0'; i++)
 {
  if (str[i] == ' ')
  {
   words++i;
  }
 }

printf("Total no of words: %d \n", words);
 return 0;
}
```

OUTPUT

Enter a string: I Love coding

Total no of words: 3

EXPERIMENT : 3

(1) Sort elements in ascending order using BubbleSort.

```
#include <stdio.h>
 int main()
{
 int a[50], n, i, j, temp;

 printf("\n Enter no of elements: ");
 scanf("%d", &n);

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

for(i=0; i<n-1; i++)
 {
   for(j=0; j<n-i-1; j++)
    {
      if (a[j] > a[j+1])
      {
        temp = a[j];
        a[j] = a[j+1];
          a[j+1] = temp;
      }
    }
 }
```

```
printf("\n Sorted array: ");
for (i=0 ; i<n ; i++)
  {
   printf ("%d ", a[i]);
  }
return 0;
}
```

OUTPUT

```
Enter number of elements : 5
Enter 5 elements : 8 6 5 4 1
Sorted array: 1 4 5 6 8
```

(2) Sort elements in descending order using selection sort

```
#include <stdio.h>
int main()
{
 int a[50], n, i, j, max, temp;
 printf ("\n Enter number of elements: ");
 scanf ("%d", &n);

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

for (i=0 ; i<n-1 ; i++)
  {
   max = i;
  for (j= i+1 ; j<n ; j++)
    {
     if (a[j] > a[max])
      {
       max = j;
      }
    }
  temp = a[i];
   a[i] = a[max];
   a[max] = temp;
  }
printf (" Sorted array : ")
  for (i=0; i<n ; i++)
  {
    printf ("%d", a[i]);
  }
  return 0;
}
```

OUTPUT

```
Enter number of elements : 5
Enter 5 elements : 8 6 3 78 98
Sorted array : 98 78 8 6 3
```

3. Find the number of comparison required in bubble sort method of the following list having 5 number: 100, 200, 300, 400, 500.

Pass: 1

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

Pass: 2

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

Pass: 3

100 200 300 400 500

100 200 300 400 500

100 200 300 400 500

Pass: 4

100 200 300 400 500

100 200 300 400 500

Total comparison

$4 + 3 + 2 + 1 = 10$.

4. Sort the given array in ascending array using selection sort method and show diagramatic representation of every iteration.

500, -20, 30, 14, 50.

Pass: 1

500 -20 30 14 50

-20 500 30 14 50

-20 500 30 14 50

-20 500 30 14 50

-20 500 30 14 50

Pass: 2

-20 500 30 14 50

-20 30 500 14 50

-20 14 500 30 50

-20 14 500 30 50

Pass: 3

-20 14 500 30 50

-20 14 30 500 50

-20 14 30 500 50

Pass: 4

-20 14 30 500 50

-20 14 30 50 500

Page No.
Date

EXPERIMENT: 4

1. Sort element in ascending order using insertion sort.

```
#include <stdio.h>
int main()
{
  int n, a[50], temp;
  printf("Enter no of array elements: ");
  scanf("%d", &n);
  printf("Enter elements: \n", n);
  for(i=0; i<n; i++)
  {
    scanf("%d", &a[i]);
  }
  for(i=1; i<n; i++)
  {
    for(int j=0; j<i; j++)
    {
      if(a[i] < a[j])
      { temp = a[i];
        for(int k=i; k>=j; k--)
        {
          a[k+1] = a[k];
        }
        a[j] = temp;
        break; }}
  printf("After pass: %d", i);
    for(int m=0; m<n; m++) {
```

```
printf("%d", a[m]);
}
printf("\n");
}
```

o/p

2. Sort elements

```
# include <stdio.h>
int main()
{
 int a[50], output[50], count[10], int n,
 j, i, max = 0, place = 1;
 printf("Enter number of elements: ");
 scanf("%d", &n);
 printf("Enter array elements: ");
 for(i=0; i<n; i++)
 {
   scanf("%d", &a[i]);
   if (a[i] > max)
    {  max = a[i];
    }
 }
 while (max/place > 0)
 { for(i=0; i<10; i++)
   {  count[i] = 0;
   for (i=0; i<n; i++)
    {  count[(a[i]/place) %10] ++;
   for (i=1; i<10; i++)
    {  count[i] += count[i-1];
    }
    for (i = n-1; i>=0; i++)
    {
   int digit = (a[i]/place) %10;
    output (--count[digit]-a[i];
    }
```

```
for(i=0; i<n; i++)
{ a[i] = output(i);
  place *= 10;
}
printf("Sorted array: ");
for(i=0; i<n; i++)
{
  printf("%d", a[i]);
}
return 0;
}
```

3] What is the output of the insertion sort after the 2nd iteration for the following.

7, 3, 1, 9, 4, 8, 6.

1st iteration:

7, 3, 1, 9, 4, 8, 6

3, 7, 1, 9, 4, 8, 6

2nd iteration:

~~3 7~~

3, 7, 1, 9, 4, 8, 6

1, 3, 7, 9, 4, 8, 6

4) Sort the following using radix sort.

i] 100, 225, 390, 4130, 956, 99, 5431

→

	0	1	2	3	4	5	6	7	8	9
100	100									
225						225				
390	390									
4130	4130									
956									956	
0099										99
5431						5431				

	0	1	2	3	4	5	6	7	8	9
100	100									
390										390
4130				4130						
5431				5431						
225			225							
956							956			
99										99

	0	1	2	3	4	5	6	7	8	9
0100		100								
0225			225							
4130		4130								
5431						5431				
0956										956
0390				390						
0099	99									

	0	1	2	3	4	5	6	7	8
0099	099								
0100	100								
4130					4130				
0225	225								
0390	390								
5431						5431			
0956	956								

Sorted Array: 99, 100, 225, 390, 956, 4130, 5431

ii] 256, 99, 145, 239, 20, 18.

	0	1	2	3	4	5	6	7	8	9
256							256			
99										99
145						145				
239									239	
20	20									
18									18	

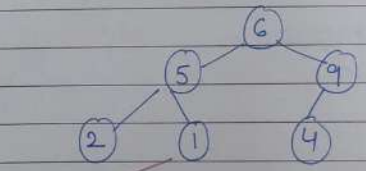
	0	1	2	3	4	5	6	7	8	9
20			20							
145					145					
256							256			
18	18									
99										
239										99
				239						

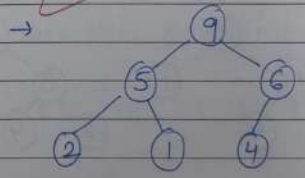
	0	1	2	3	4	5	6	7	8	9
18	18									
20	20									
239									239	
145						145				
256									256	
99	99									

SORTED ARRAY - 18, 20, 99, 145, 239, 256

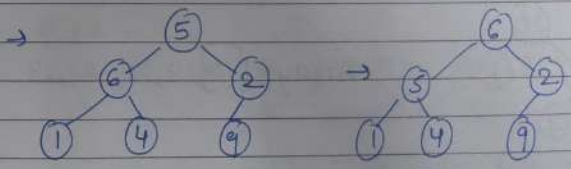
5) Sorted Array using heap sort.

i] 6, 5, 9, 2, 1, 4.

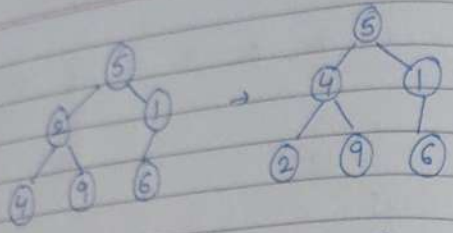




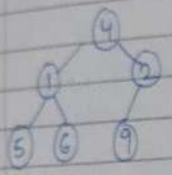
Array - [9] 5, 6, 2, 1, 4



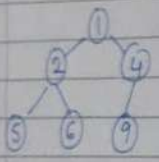
Array - [6, 5, 2, 1, 4, 9]



Array - [5], 4, 1, 2, 9 6, 9

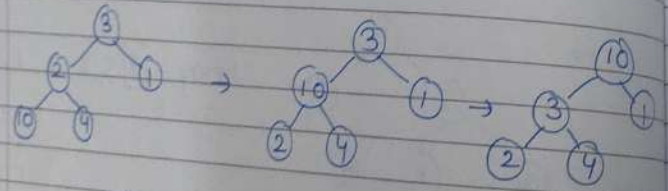


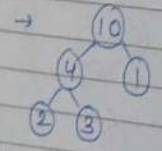
Array - [4], 1, 2, 5, 6, 9



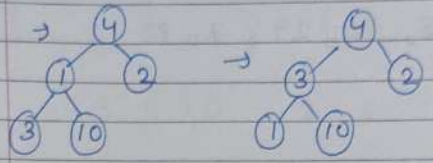
Sorted Array - 1, 2, 4, 5, 6, 9

ii] 3, 2, 1, 10, 4.

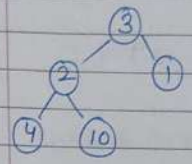




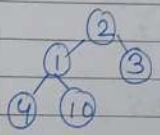
Array - [10], 4, 1, 2, 3



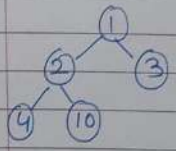
Array - [4], 3, 2, 1, 10



→ Array - [3], 2, 1, 4, 10



→ Array - 2, 1, 3, 4, 10



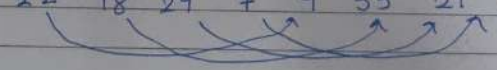
→ Sorted Array - 1, 2, 3, 4, 10

6) Sort the following using Shell Sort.

i] 22, 18, 29, 7, 9, 55, 21, 8.

→ n = 8

n/2 = 4



{22, 9}, {18, 55}, {29, 21}, {7, 8}

→ 9, 22, 18, 55, 21, 29, 7, 8.

n/4 = 2

{9 22 18 55} {21 29 7 8}

→ 9, 18, 22, 55, 7, 8, 21, 29

n/8 = 1

{9 18 22 55 7 8 21 29}

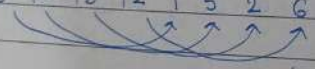
→ 7, 8, 9, 18, 21, 22, 29, 55.

ii] 3, 10, 15, 12, 1, 5, 2, 6.

n = 8.

n/2 = 4.

3 10 15 12 1 5 2 6.



{3 1} {10 5} {15 2} {12 6}

1, 3, 5, 10, 2, 15, 6, 12.

n/4 = 2

{1, 3, 5, 10} {2, 15, 6, 12}

1 3 5 10 2 6 12 15

n/8 = 1.

{1, 3, 5, 10, 2, 6, 12, 15}

Sorted Array - 1, 2, 3, 5, 6, 10, 12, 15.

EXPERIMENT-5

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

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

struct Node * head = NULL;
void insertAtBeginning();
void insertAtEnd();
void insertAtPosition();
void deleteFirst();
void deleteLast();
void deleteAtPosition();
void display();
void createlist();
void reverse();
void search();

int main(){
  int choice;

while (1) {
 printf("\n-- SINGLY LINKED LIST (FULL OPERATIONS) -- \n");
 printf("1. Create List \n");
      ("2. Insert at Beginning \n");
      ("3. Insert at End \n");
      ("4. Insert at Position \n");
      ("5. Delete First Node \n");
      ("6. Delete Last Node \n");
      ("7. Delete From Position \n");
      ("8. Display List \n");
      ("9. Reverse List \n");
      ("10. Search Node \n");
      ("11. Exit \n");
printf ("Enter your choice: ");
scanf ("%d", &choice);

 switch (choice) {
  case 1:
    createList();
    break;
  case 2:
    insertAtBeginning();
    break;
  case 3:
    insertAtEnd();
    break;
  case 4:
    insertAtPosition();
    break;
  case 5:
    deleteFirst();
    break;
```

```
case 6:
deleteLast();
break;
case 7:
deleteAtPosition();
break;
case 8:
display();
break;
case 9:
reverse();
break;
case 10:
search();
break;
case 11:
printf("Existing...\n");
exit(0);
default:
printf("Invalid choice!\n");
}
}
return 0;
}

void createList() {
    if (head != NULL) {
        printf("List already exists!\n");
        return;
    }
    struct Node * newNode = (structNode *) malloc
        (sizeof (struct Node));
    printf("Enter data for first node: ");
    scanf("%d", &newNode -> data);
    newNode -> next = NULL;
    head = newNode;
    printf("List created with node: %d\n", newNode -> data);
}

void insertAtBeginning () {
    struct Node * newNode = (struct Node *) malloc
    (sizeof (struct Node));
    printf("Enter data to insert at Beginning: ");
    scanf("%d", &newNode -> data);
    newNode -> next = head;
    head = newNode;
    printf("Inserted %d at the beginning\n", newNode -> data);
}

void insertAtEnd () {
    struct Node * newNode = (struct Node *) malloc (sizeof
    (struct Node));
    printf("Enter data to insert at end:");
    scanf("%d", &newNode -> data);
    newNode -> next = NULL;

    if (head == NULL) {
        head = newNode;
        printf("Inserted %d as first node\n", newNode
        -> data);
    }
```

```
else
{ struct Node * temp = head;
  while (temp -> next != NULL) {
      temp = temp -> next;
  }
  temp -> next = newNode;
printf("Inserted %d at the end \n", newNode->data);
 }
}

void insertAtPosition()
{ int pos;
 printf("Enter position to insert at (1-based): ");
 scanf("%d", &pos);

 if(pos < 1) {
  printf("Position must be >=1 ! \n");
  return;
 }

 struct Node * newNode = (struct Node *) malloc
  (sizeof(struct Node));
 printf("Enter data for new node: ");
 scanf("%d", &newNode -> data);

 if (pos == 1) {
   newNode -> next = head;
    head = newNode;
  printf("Inserted %d at position 1 \n", newNode->data);
  return;
 }
```

```
 struct Node * temp = head;
  for(int i = 1; i < pos - 1; i++)
   {  temp = temp -> next;
      if (temp == NULL)
       { printf("Position out of bounds! \n");
         free(newNode);
         return;
       }
   }
 if (temp == NULL) {
  printf("Position out of bounds! \n");
   free(newNode);
   return;
 }

 newNode -> next = temp -> next;
  temp -> next = newNode;
 printf("Inserted at position %d \n", pos);
 }

 void deleteFirst() {
   if (head == NULL) {
    printf("List is empty ! \n");
    return;
   }

 struct Node * temp = head;
  head = head -> next;
 printf("Deleted first node ");
  free(temp);
 }
```

```
void deleteLast() {
    if (head == NULL) {
    printf("List is empty! \n");
    return;
}

if (head->next == NULL) {
printf("Deleted last node \n");
 free(head);
 head = NULL;
 return;
}

struct Node *temp = head;
while (temp->next->next != NULL) {
     temp = temp->next;
 }

printf("Deleted last node. \n");
free(temp->next);
temp->next = NULL;
}

void deleteAtPosition() {
  int pos;
printf("Enter position to delete : ");
scanf("%d", &pos);

if (pos<1 || head == NULL)
{
printf("Invalid position or empty list! \n");
 return;
}
if (pos == 1) {
     deleteFirst();
     return;
 }

struct Node * temp = head;
for(int i = 1; i<pos-1; i++) {
     temp = temp -> next;
     if (temp == NULL || temp -> next == NULL)
  { printf(" Position out of bounds");
     return;
     }
   }
if (temp -> next == NULL)
{ printf(" Position out of bounds!");
   return;
}

struct Node * nodeToDelete = temp -> next;
 temp -> next = nodeToDelete -> next;
printf(" Deleted node at position %d with data
     %d \n", pos, nodeToDelete -> Data);
     free (nodeToDelete);
}
```

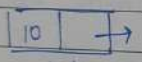
```
void display() {
  if (head == NULL) {
  printf("List is empty \n");
  return;
}

struct Node *temp = head;
printf("List: ");
while (temp != NULL) {
  printf("%d -> ", temp -> data);
  temp = temp -> next;
}
printf("NULL \n");
}

void reverse() {
  if (head == NULL) {
  printf("List is empty! Cannot reverse \n");
  return;
}

Struct Node * prev = NULL;
struct Node * current = head;
struct Node * next;

while (current != NULL)
{ next = current -> next;
  current -> next = prev;
  prev = current;
  current = next;
}

  head = prev;
  printf(" List reversed successfully \n");
}

void search() {
  if (head == NULL)
  { printf("List is empty! \n");
    return;
  }
  int value;
  printf(" Enter value to search: ");
  scanf("%d", &value);

  struct Node * temp = head;
   int pos = 1;

  while (temp != NULL) {
     if (temp -> data == value)
     { printf("Value found ");
       return;
     }
     temp = temp -> next;
      pos ++;
  }
  printf("Value not found in the list. \n");
  }
```

g) Difference between Array and Linked List

ARRAY	LINKED LIST
Resizing not possible	Resizing is possible
Memory allocation is static	Memory can change
Stored in contiguous memory.	Stored in non-contiguous memory.
Memory usage is efficient.	Uses extra memory for pointers.
O(1)	O(n)

g) Basic Terminology of Linked List.

a) Node: Basic building block of Linked List. It contains data & next (address of next node).

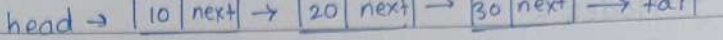


Node

b) Head: The head is a reference or pointer to first node of list Null or none.

c) Tail: Last node where next = NULL

d) Next: Each node has a next reference to the following node.



e) Null Pointer: Special value that indicates that pointer does not have object / memory.

f) Empty Linked List: singly linked list with no nodes in it.

Head → Null.

g) Circular Linked List.

A circular linked list is a type of Linked List where the last node points back to the first node, forming a continuous loop. Unlike a standard singly linked list, there is no NULL pointer at the end to indicate termination.

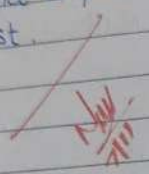
Key characteristics:

→ Circular structure -
The defining feature is that the next pointer of the last node references the head (first) node, creating a closed loop.

→ No Termination -
There is no NULL pointer, meaning traversal can continue indefinitely unless a specific stopping condition is implemented.

→ Nodes -
Each nodes typically contains two parts :

- Data :
  The value or information stored within the node.
- Next Pointer :
  A reference or pointer to the subsequent node in the list.



Doubly Linked List.

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

void create();
void display();
void insert_begin();
void insert_end();
void insert_position();

struct node {
   int info;
   struct node* next;
   struct node* prev;
};

struct node* start = NULL;

int main() {
   int choice;
   while (1) {
      printf("\n MENU ");
      printf("\n 1. CREATE");
      printf("\n 2. DISPLAY");
      printf("\n 3. INSERT_BEGIN");
      printf("\n 4. INSERT_END");
      printf("\n 5. INSERT_POSITION");
      printf("\n 6. EXIT");
      printf("\n Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            create();
            break;
         case 2:
            display();
            break;
         case 3:
            insert_begin();
            break;
```

```
            case 4:
                insert_end();
                break;
            case 5:
                insert_position();
                break;
            case 6:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice, please try again.\n");
        }
    }
    return 0;
}

void create() {
    struct node* temp;
    struct node* ptr;

    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    printf("\n Enter data: ");
    scanf("%d", &temp->info);
    temp->next = NULL;
    temp->prev = NULL;

    if (start == NULL) {
        start = temp;
    } else {
        ptr = start;
        while (ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = temp;
        temp->prev = ptr;
    }
    printf("Node created and added at the end.\n");
}
```

```
void display() {
    struct node* ptr;
    if (start == NULL) {
        printf("\n Empty List\n");
        return;
    }
    ptr = start;
    printf("\n List elements are: ");
    while (ptr != NULL) {
        printf("%d ", ptr->info);
        ptr = ptr->next;
    }
    printf("\n");
}

void insert_begin() {
    struct node* temp;
    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
    printf("\n Enter data to insert at beginning: ");
    scanf("%d", &temp->info);

    temp->next = start;
    temp->prev = NULL;

    if (start != NULL)
        start->prev = temp;

    start = temp;
    printf("Node inserted at the beginning.\n");
}

void insert_end() {
    struct node* temp;
    struct node* ptr;

    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL) {
        printf("Memory allocation failed\n");
        return;
    }
```

```
    printf("\n Enter data to insert at end: ");
    scanf("%d", &temp->info);
    temp->next = NULL;
    temp->prev = NULL;

    if (start == NULL) {
        start = temp;
    } else {
        ptr = start;
        while (ptr->next != NULL) {
            ptr = ptr->next;
        }
        ptr->next = temp;
        temp->prev = ptr;
    }
    printf("Node inserted at the end.\n");
}

void insert_position() {
    struct node* temp;
    struct node* ptr;
    int pos, i;

    temp = (struct node*)malloc(sizeof(struct node));
    if (temp == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    printf("\n Enter data to insert: ");
    scanf("%d", &temp->info);
    temp->next = NULL;
    temp->prev = NULL;

    printf(" Enter position to insert node (starting from 1): ");
    scanf("%d", &pos);

    if (pos == 1) {
        temp->next = start;
        temp->prev = NULL;
        if (start != NULL)
            start->prev = temp;
        start = temp;
        printf("Node inserted at position 1.\n");
```

```
        return;
    }

    ptr = start;
    for (i = 1; i < pos - 1 && ptr != NULL; i++) {
        ptr = ptr->next;
    }

    if (ptr == NULL) {
        printf("Position out of bounds.\n");
        free(temp);
    } else {
        temp->next = ptr->next;
        temp->prev = ptr;

        if (ptr->next != NULL)
            ptr->next->prev = temp;

        ptr->next = temp;
        printf("Node inserted at position %d.\n", pos);
    }
}
```

Page No.
Date

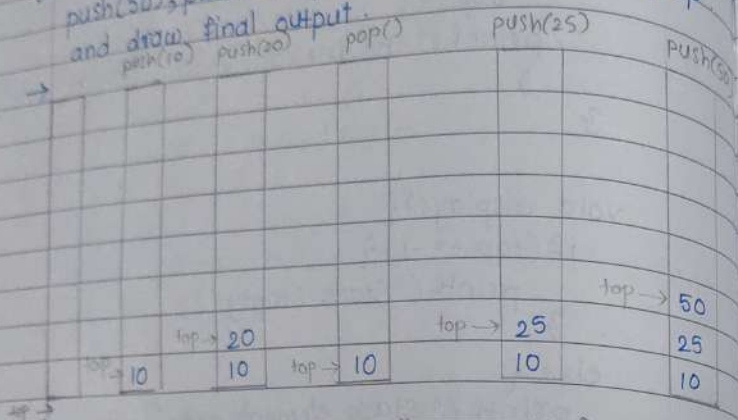
EXPERIMENT : 7

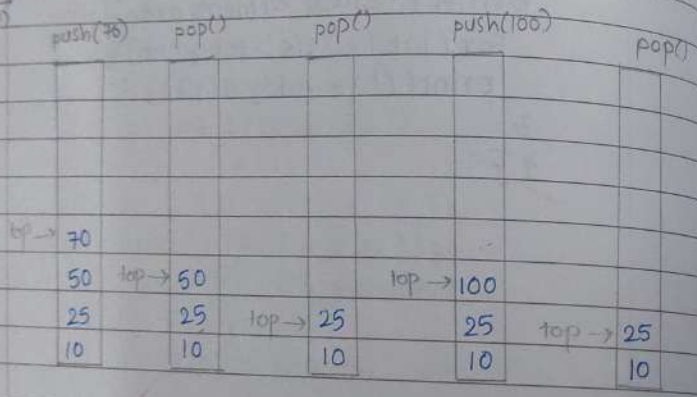
i] Primitive operations on stack.

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

void push();
void pop();
void display();

int s[Max];
int top = -1;

int main() {
 int choice;
    printf("\n MENU: ");
    printf("\n 1. Push");
    printf("\n 2. Pop");
    printf("\n 3. Display");
    printf("\n 4. Exit");
    do {
    printf("\n Enter your choice:");
    scanf("%d", &choice);

   switch(choice) {
       case 1: push();
           break;
       case 2: pop();
           break;
```

```
case 3: display();
        break;
case 4: exit(0);
        break;
default:
printf("Invalid choice!");
}
}
while (choice != 4);
return 0;
}

void push() {
  int element;
  if (top == Max-1) {
     printf("Overflow");
  }
  else {
  printf("Enter value");
  scanf("%d", &element);
  top = top + 1;
  s[top] = element;
  }
}

void pop() {
  int item;
  if (top == -1) {
     printf("Underflow");
  }
  else {
    item = s[top];
     top--;
    printf("The deleted element is %d", item);
  }
}

void display() {
  if (top == -1) {
     printf("Stack Empty");
  }
  else {
   printf("Stack elements are:");
    for (int i=0; i<=top; i++) {
     printf("\t %d", s[i]);
  }
}}
```

Q) Stack size-8 for push(10), push(20), pop, push(25), push(50), push(70), pop, pop, push(100), pop and draw final output.





EXPERIMENT - 8

i) Convert Infix to Postfix Expression

```
#include <stdio.h>
#include <ctype.h>
char stack[100];
int top = -1;
void push(char x)
{  stack[++top] = x;
}
char pop()
{ if (top == -1)
    return -1;
  else
    return stack[top--]
}
int priority (char x)
{   if (x == '(')
      return 0;
    if (x == '+' || x == '-')
      return 1;
    if (x == '*' || x == '/')
      return 2;
    return 0;
}
int main() {
  char exp[100];
  char *e, x;
  printf("Enter the expression");
```

```
scanf("%s", &exp);
printf("\n");
e = exp;

while (*e != "\0")
{ if (isalnum(*e))
  printf("%c", *e);

else if (*e == '(')
push(*e);
else if (*e == ')')
{
while ((x = pop()) != '(')
  printf("%c", x);
}
else
{
  while (priority(stack[top]) >= priority(*e))
  printf("%c", pop());
   push(*e);
  }
  e++;
}
while (top != -1)
{
  printf("%c", pop());
}
return 0;
}
```

iii] Evaluate postfix expression.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define SIZE 40

int pop();
void push(int);

char postfix[SIZE];
int stack[SIZE], top = -1;

int main() {
   int i, a, b, result, pEval;
   char ch;
for (i=0; i< SIZE; i++)
{
   stack[i] = -1;
}
printf("\n Enter a postfix expression: ");
scanf("%s", postfix);

for (i=0; postfix[i] != '\0'; i++)
  {  ch = postfix[i];

   if (isdigit(ch))
    {
       push(ch - '0');
    }
```

```
else if (ch=='+' || ch=='-' || ch=='*' ||
         ch=='/')
    b = pop();
    a = pop();

switch (ch)
case '+':
    result = a+b;
    break;
case '-':
    result = a-b;
    break;
case '/':
    result = a/b;
    break;
case '*':
    result = a*b;
    break;
case '%':
    result = a%b;
    break;
}
push (result);
}
}
pEval = pop();
printf ("\n The postfix evaluation is : %d\n" pEval);
return 0;
}
```

```
void push (int n)
{  if (top < SIZE - 1)
   {
      stack [++top] = n;
   }
   else {
      printf ("Stack is full! \n");
      exit(-1);
   }
}
int pop()
{  int n;
   if (top > -1)
   {
      n = stack [top];
      stack [top--] = -1;
      return n;
   }
   else
   {  printf ("Stack is empty! \n");
      exit (-1);
   }
}
```

iii] conversion of infix to prefix.

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

I/P	Stack	O/P
		H
A	*	H
+	*(	H
(	*(	GHG
B	*(*	HG
*	*(*(	HG
C		HGF
-	*(*(^	HGF
(	*(*(^	HGFE
^	*(*(/	HGFE^
D	*(*(/	HGFE^D
/	*(*(/	HGFE^D/
E	*(*-	HGFE^D/*
^	*(-	HGFE^D/*C
F	*(-*	HGFE^D/*C
)	*(-*	HGFE^D/*CB
*	*(-*	HGFE^D/*CB*-
G	*+	HGFE^D/*CB*-*
)	+	HGFE^D/*CB*-*A+
*		
H		

+A*-*BC*/D^EFGH

iv] Evaluate prefix expression : +-*+12/421$42

I/P	Op1		Op2	Result	Stack
2					2
4					2,4
$	4	$	2	$4^2=16$	16
1					16,1
2					16,1,2
4					16,1,2,4
/	4	/	2	2	16,1,2
2					16,1,2,2
1					16,1,2,2,1
+	1	+	2	3	16,1,2,3
*	3	*	2	6	16,1,6
-	6	-	1	5	16,5
+	5	+	16	21	21

v] Algorithm for infix to postfix conversion

~~Step 1~~

i] Read the expression from left to right.

ii] If an operand (A-Z, 0-9) is encountered add it directly to the output.

iii] If an operator is encountered:

→ Pop operators from the stack to the output if they have higher or equal precedence.

→ Push the current operator onto the stack.

iv] If an opening paranthesis '(' is found, push it into the stack.

v] If a closing paranthesis ')' is found, pop operators from the stack to the output until an opening paranthesis is encountered.

vi] After scanning the entire expression, pop any remaining operators from the stack to the output.

vi] Algorithm for Infix to Prefix ~~expressio~~ conversion

i] Reverse the infix string. Note that while reversing the string you must interchange left and right paranthesis.

ii] Obtain the postfix expression of the infix expression obtained in step 1.

iii] Reverse the postfix expression to get the prefix expression.

vii] Algorithm to evaluate Prefix expression

i] Start from the last element of the expression.

ii] Check the current element.

iii] If it is an operand, push it to the stack.
If it is an operator, pop two operands from the stack. Perform the operation and push the elements back to the stack.

iv] Do this till all elements of expression are traversed and return the top of stack which will be the result of the operation.

viii] Algorithm For Postfix expression evaluation.

i] Scan the elements of string from left to right.

ii] ~~the~~ If the element is an operand, push it into the stack.

iii] If the element is an operator, pop two operands from the stack.

iv] The first pop is second operand & second pop is first operand.

v] Perform arithmetic operation & push result back into stack.

vi] When input expression is completely traversed pop operand stack & return the value.

EXPERIMENT: 9

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5

void enqueue();
void dequeue();
void display()
int queue[SIZE];
int front = -1, rear = -1;
int main(){
  int ch;
  printf("1. Enqueue \n 2. Dequeue \n 3. Display
      \n 4. Exit \n");
  do{
  printf("\n Enter your choice: ");
  scanf("%d", &ch);
  switch(ch){
  case 1: enqueue();
    break;
  case 2: dequeue();
    break;
  case 3: display();
    break;
  case 4: exit(0);
  default:
    printf("\n Invalid choice! \n");
}
while (ch != 4)
  return 0;
}
void enqueue(){
   int element;
   if (rear == SIZE-1){
     printf("\n Queue is full...");
   }
   else{
   printf("\n Enter element to insert:");
   scanf("%d", &element);
   if (front == -1)
    front = 0;
    rear ++;
    queue[rear] = element;
  }
}
void dequeue(){
   if (front == -1 || front > rear){
     printf("\n Queue is empty...");
   }
  else{
    printf("\n Deleted element: %d", queue[front]);
    front ++;
  }
}
void display(){
  if (front == -1 || front > rear){
     printf("\n Queue is empty...");
   }
```

```
else {
printf("In queue element are: \n");
for (int i = front; i<=rear ;i++) {
    printf("%d\t", queue[i]);
}
printf("\n");
}
}
```

Q Perform following operation on linear queue in array size (10).

i) insert (10)

10

Front = 0 Rear = 0

ii) Insert (50)

10	50

Front = 0 Rear = 1

iii) Delete

50

Front = 1 Rear = 1

iv) Insert (100)

50	100

Front = 1 Rear = 2

v) Insert (20)

50	100	20

Front = 1 Rear = 3

vi) Delete

100	20

Front = 2 Rear = 3

vii) Insert (25)

100	20	25

Front = 2 Rear = 4

viii) Insert (200)

100	20	25	200

Front = 2 Rear = 5

Q) Explain concept of priority queue in detail.

→ A priority queue is a type of data structure similar to a regular queue but each element is assigned a priority. Instead of following the FIFO rule, the element with highest priority is served first. If two elements have the same priority, the one that entered first is usually served first.

Q) Algorithm for insertion in linear queue

- Check the overflow
- Check if queue is empty. In case it is, then both FRONT and REAR are set to zero. So that new value can be stored at the 0th location.
- Otherwise, if queue already has some values then REAR is incremented such that it points to the next location in array.
- The value is stored in the queue at the location pointed by REAR.
- Exit.

Deletion

- We check for underflow condition. An underflow occurs if FRONT = -1 or FRONT > REAR.
- If queue has some values, the FRONT is incremented so that it now points to next value in queue.
- Exit.

Q) Applications of Queue

+ Serving requests on a single share resource, like printer, CPU task scheduling etc.
- In real life scenario, call center phone system uses queues to hold people calling them in an order, until a service representative is free.
- Handling interrupts in real-time systems. The interrupts are handled in same order as they arrive i.e, FCFS.
- FCFS job scheduling.
- Online banking Fund transfer requests.

EXPERIMENT: 10

```
#include <stdio.h>
#include <stdlib.h>
#define size 5

void enqueue();
void dequeue();
void display();

int queue [size];
int front=-1, rear=-1;
int main()
{ int ch;
  printf("\n 1. Enque \n 2. Dequeue \n 3.
          Display \n 4. Exit \n");
Do{
  printf(" Enter your choice : ");
  scanf("%d", &ch);

  switch (ch){

   case 1: enqueue();
      break;
   case 2: dequeue();
      break;
   case 3: display();
      break;
   case 4: exit(0);
   default:
      printf(" Invalid Choice! ");
}
}
while (ch!= 4);
 return 0;
}

void enqueue()
{
  int value;
  if ((FRONT == 0 & REAR = SIZE-1) ||
       (rear + 1 == Front))
   {
   printf("\n Queue is Full \n");
   }
  else
   {
   printf(" \n Enter value :");
   scanf("%d", & value);
   if (front == -1)
    front = 0;

  rear = [rear +1 ]/size;
  queue [rear] = value;
 }
}

void dequeue()
{
```

```
{
 if ( Front == -1)
 {
  printf ("Queue is empty ");
 }
 else
 {
 printf ("\n Deleted %d from the queue
  \n", queue[front]);

if (front==rear)
{
  Front = rear -1;
}
else
{
  front = (Front + 1) / size;
 }
}
}

void display()
{
 if (Front == -1)
 {
  printf ("\n Queue is empty! ");
 }
 else
 {
  int i = Front;
  printf ("\n queue elements are : ");
  while (1) {
   printf ("%d ", queue [i]);
    if (i == rear)
      break;

    i =(i +1) / SIZE;
  }
  printf ("\n");
  }
}
```

(Q) Insertion Algorithm.

- If front = 0 and rear = max - 1 the queue will overflow.
- If front is rear+1 it will also overflow queue.
- IF Front = -1 and rear = -1, Front = rear = 0.
- Else if rear = max - 1 and Front != 0.
  Rear = 0.
- Else, rear = rear + 1.
  End.
- queue [rear] = value
- Exit.

* Deletion.

- If Front = -1 write underflow.
- Go to step 4 at end of if.
- Set val = queue[front].
- If Front = rear, set Front = rear = -1.
- Else, if Front = max-1, set Front = 0.
- Else, set Front = Front + 1.
- Exit.

2) * Applications of priority queue.

→

- Traffic light control -
Manages signal priorities based on real-time traffic sensor data.
- Operating system algorithms -
Used for scheduling process to execute high priority first.
- Huffman codes -
Priority queue helps build huffman trees for efficient data compression.

EXPERIMENT - 11

1)

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

struct node{
  int item;
  struct node * left;
  struct node * right;
};

struct node * createNode (int item){
  struct node * newNode = (structnode *)
    malloc (sizeof (structnode));
  newNode -> item = item;
  newNode -> left = NULL;
  newNode -> right = NULL;
  return newNode;
}

struct node * insertAtLeft (struct node * root,
    int item){
  root -> left = createNode (item);
  return root -> left;
}

struct node * insertAtRight (struct node * root,
  int item){
    root -> right = createNode (item);
    return root -> right;
}
```

```
void inorder(struct node *root){
  if(root==NULL)
  return;
  inorder(root->left);
  printf("%d", root->item);
  inorder(root->right);
}

void preorder(struct node *root){
  if(root==NULL)
  return;
  printf("%d", root->item);
  preorder(root->left);
  preorder(root->right);
}

void postorder(struct node *root){
  if(root==NULL)
  return;
  postorder(root->left);
  postorder(root->right);
  printf("%d", root->item);
}

int main(){
  struct node *root = createNode(40);
  insertAtLeft(root, 20);
  insertAtRight(root,60);
  insertAtLeft(root->left,10);
  insertAtRight(root->left, 30);
  insertAtLeft(root->right,50);
  insertAtRight(root->right,70);

  printf("\nInorder traversal:");
  inorder(root);
  printf("\nPreorder traversal:");
  preorder(root);
  printf("\nPostorder traversal:");
  postorder(root);
  printf("\n");
  return 0;
}
```

2)

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

struct node {
  int item;
  struct node *left;
  struct node *right;
};

struct node * createNode(int item)
{ struct node * newNode = (struct node *)
    malloc (sizeof (struct node));
    newNode -> item = item;
    newNode -> left = NULL;
    newNode -> right = NULL;
    return newNode;
}

struct node * insertAtLeft (struct node * root,
  int item)
{ root -> left = createNode (item);
  return root -> left;
}

struct node * insertAtRight ( struct node * root,
  int item)
{ root -> right = createNode (item);
  return root -> right;
}
```

```
int main()
{ struct node * root = createNode (40);
  insertAtLeft ( root, 20);
  insertAtRight (root, 60);
  insertAtLeft ( root -> left, 10);
  insertAtRight ( root -> left, 30);
  insertAtLeft (root -> right, 50);
  insertAtRight ( root -> right, 70);
}
```

EXPERIMENT-12

```
#include <stdio.h>
#define V 5

void init(int arr[][V])
{  int i,j;
   for(i=0;i<V;i++)
      for(j=0;j<V;j++)
         arr[i][j]=0;
}

void insertedge(int arr[][V],int i,int j){
    arr[i][j] = 1;
    arr[j][i] = 1;
}

void printadjmatrix(int arr[][V]){
   int i,j;
   printf("Adjacency Matrix:\n");
   for(i=0;i<V;i++){
      printf("%d",i);
      for(j=0;j<V;j++){
         printf("%d",arr[i][j]);
      }
      printf("\n");
   }
}
```

```
int main()
{  int adjmatrix[V][V];
   init(adjmatrix);

insertedge(adjmatrix, 0,1);
"          "          ,0,3)
"          "          ,0,2)
"          "          ,0,4)
"          "          ,1,3)
"          "          ,2,0)
"          "          ,2,3)
"          "          ,2,4)
"          "          ,3,0)
"          "          ,3,1)
"          "          ,3,2)
"          "          ,3,4)
"          "          ,4,0)
"          "          ,4,2)
"          "          ,4,3)

printadjmatrix(adjmatrix);
return 0;
}
```

2)

```
#include <stdio.h>
#include <stdlib.h>
#define V 5
struct Node {
  int dest;
  struct Node * next;
};

struct graph {
  struct Node * adj[V];
};
struct Node * newNode(int dest) { struct Node * temp
= (struct Node *) malloc (sizeof (struct Node));
temp -> dest = dest;
temp -> next = NULL;
return temp;
}

struct Graph * createGraph() { struct Graph * graph
= (struct Graph *) malloc (sizeof (struct Graph));
for (int i = 0; i < V; i++)
   graph -> adj[i] = NULL;
return graph;
}

void addEdge (struct Graph * graph, int src, int dest)
{ struct Node * temp = newNode(dest);
  temp -> next = graph -> adj[src];
  graph -> adj[src] = temp;
  temp = newNode(src);
  temp -> next = graph -> adj[dest];
  graph -> adj[dest] = temp;
}

void printGraph (struct Graph * graph)
{ printf("Adjacency List: \n");
  for (int i = 0; i < V; i++) {
    printf("%d -> ", i);
    struct Node * temp = graph -> adj[i];
  while (temp) {
    printf("%d", temp -> dest);
    temp = temp -> next;
  }
  printf("\n");
} }
int main() {
  struct Graph * graph = createGraph();
  addEdge(graph, 0, 1);
          "      0, 2);
          "      0, 3);
          "      0, 4);
          "      1, 3);
          "      2, 3);
          "      2, 4);
          "      3, 4);

printGraph(graph);
return 0;
}
```

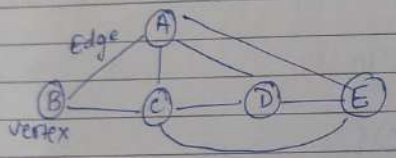
Q) Explain terminologies-

i) Vertex: Each node of graph is represented as a vertex. In the examples, labelled circle is vertex.

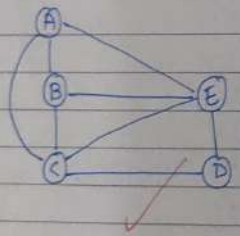
ii) Edge: Edge represents a path between two vertices or a line between two vertices.

iii) Adjacency: Two node or vertices are adjacent if they are connected to each other through an edge.

iv) Path: Path represents a sequence of edges between the two vertices.



Q)

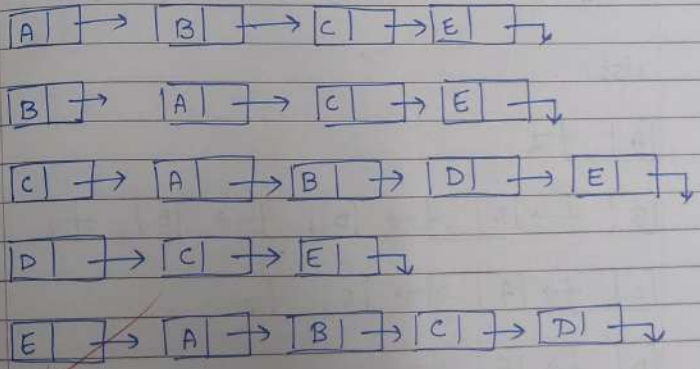


→

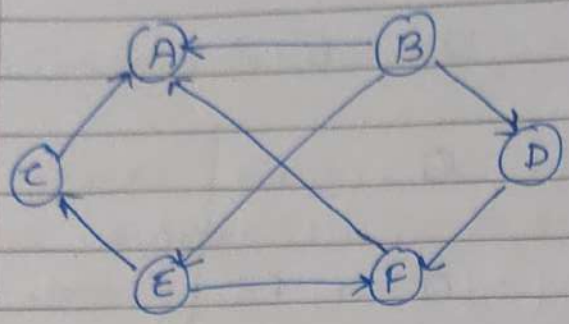
Matrix:

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	1
C	1	1	0	1	1
D	0	0	1	0	1
E	1	1	1	1	0

List:



Q)



→

Matrix:

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	1	0	0	1	1	0
C	1	0	0	0	1	0
D	0	0	0	0	0	1
E	0	0	1	0	0	1
F	1	0	0	0	0	0

List: