WEEK-1

Date:18-06-2025

List of programs:

1. Write a C program to find the factorial of given number using a recursive function.
2. Write a C program to find the GCD of two numbers using a recursive function.
3. Write a C program to generate Fibonacci series using a recursive function.
4. Write a C program to demonstrate passing of an array to a C function.
5. Write a C program to implement the Towers of Hanoi problem using recursion.

1. **Aim:** To write a C program to find the factorial of given number using a recursive function.

Program:

```
#include<stdio.h>
int fact(int n)
{
    if(n==1)
            return 1;
    else
            return n*fact(n-1);
}
void main()
{
    int n,factorial;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    factorial=fact(n);
    printf("Factorial of %d is %d\n",n,factorial);
```

```
}
```

Output:

```
Enter a positive integer: 4
Factorial of 4 is 24
```

2.Aim: To Write a C program to find the GCD of two numbers using a recursive function.

Program:

```
#include<stdio.h>
int gcd(int a,int b)
{
    if(a==0)
            return b;
    else if(b==0)
            return a;
    else if(a<b)
            return gcd(a,b%a);
    else
            return gcd(b,a%b);
}
void main()
{
    int x,y,GCD;
    printf("Enter two integers: ");
    scanf("%d%d",&x,&y);
    GCD=gcd(x,y);
    printf("GCD of %d and %d is %d\n",x,y, GCD);
}
```

Output:

```
Enter two integers: 55
65
GCD of 55 and 65 is 5
```

3.Aim: Write a C program to generate Fibonacci series using a recursive function.

Program:

```
#include<stdio.h>
int fib(int n)
{
    if(n==1)
        return 0;
    else if(n==2)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
void main()
{
    int num,fibonacci;
    printf("Enter an integer: ");
    scanf("%d",&num);
    fibonacci=fib(num);
    printf("%d\n ",fibonacci);
}
```

Output:

```
Enter an integer: 6
5
```

4.Aim: Write a C program to demonstrate passing of an array to a C function.

Program:

```
#include <stdio.h>
void printArray(int arr[], int size)
 {
   printf("Array elements are: ");
   for(int i = 0; i < size; i++)
  {
      printf("%d ", arr[i]);
   }
   printf("\n");
}
int sumArray(int arr[], int size)
 {
   int sum = 0;
   for(int i = 0; i < size; i++)
 {
      sum += arr[i];
   }
   return sum;
}
int main()
 {
   int arr[5] = {10, 20, 30, 40, 50};
   int size = 5;
   printArray(arr, size);
   int sum = sumArray(arr, size);
```

```c
    printf("Sum of array elements = %d\n", sum);


  return 0;
 }
```

Output:

```
Array elements are: 10 20 30 40 50
Sum of array elements = 150
```

5.Aim:

Write a C program to implement the Towers of Hanoi problem using recursion.

Program:

```
#include<stdio.h>
void TOH(int n,char source,char aux,char dest)
{
    if(n==1)
    {
            printf("Move disk 1 fron %c to %c\n",source,dest);
    }
    else
    {
            TOH(n-1,source,dest,aux);
            printf("Move disk %d fron %c to %c\n",n,source,dest);
            TOH(n-1,aux,source,dest);
    }
}
void main()
{
    int n;
    printf("Enter number of disks: ");
    scanf("%d",&n);
    TOH(n,'A','B','C');
}
```

Output:

```
Enter number of disks: 4
Move disk 1 fron A to B
Move disk 2 fron A to C
Move disk 1 fron B to C
Move disk 3 fron A to B
Move disk 1 fron C to A
Move disk 2 fron C to B
Move disk 1 fron A to B
Move disk 4 fron A to C
Move disk 1 fron B to C
Move disk 2 fron B to A
Move disk 1 fron C to A
Move disk 3 fron B to C
Move disk 1 fron A to B
Move disk 2 fron A to C
Move disk 1 fron B to C
```

Inferences:

- Recursion is a powerful concept where a function calls itself to solve a problem step by step.
- Problems like factorial, GCD, Fibonacci series, and Towers of Hanoi are classic examples suited for recursive solutions.
- Recursive programs require a base case (stopping condition) and a recursive case (function calling itself).
- Factorial and Fibonacci demonstrate recursion in mathematical sequences.
- GCD using recursion highlights the efficiency of Euclid’s algorithm.
- Towers of Hanoi illustrates recursion in problem-solving using divide-and-conquer.
- Passing arrays to functions shows that arrays are passed by reference in C, allowing functions to access/modify array elements.
- These programs improve understanding of functions, recursion, parameter passing, and modular programming in C.

WEEK-2

Date:25-06-2025

List of programs:

1. Write a C program to search an element in the given list using recursive Linear Search technique.
2. Write a C program to search an element in the given list using non-recursive Linear Search technique.
3. Write a C program to search an element in the given sorted list using recursive Binary Search technique.
4. Write a C program to search an element in the given sorted list using recursive Binary Search technique.

1. **Aim:** To write a C program to search an element int the given list using recursive Linear Search technique.

Program:

```
#include<stdio.h>
int LinearSearch(int arr[],int size,int key,int index)
{
if(index>=size)
    return -1;
if(arr[index]==key)
    return index;
return LinearSearch(arr,size,key,index+1);
}
int main()
{
int arr[]={5,3,8,4,2};
int size=sizeof(arr)/sizeof(arr[0]);
int key=4;
```

```
int result=LinearSearch(arr,size,key,0);
if(result!=-1)
    printf(“element found at index = %d\n”,result);
else
    printf(“element not found in the list\n”);
return 0;
}
```

Output:

```
element found at index = 3


=== Code Execution Successful ===
```

2. **Aim:** To Write a C program to search an element in the given list using non-recursive Linear Search technique.

Program:

```
#include<stdio.h>
int LinearSearch(int arr[],int size,int key)
{
 int i;
   for(i=0;i<size;i++)
    {
     if(arr[i]==key)
    {
       return i;
     }
    }
   return -1;
}
int main()
{
int arr[]={5,3,8,4,2};
int size=sizeof(arr)/sizeof(arr[0]);
int key=2;
int result=LinearSearch(arr,size,key);
if(result!=-1)
     printf("element found at index = %d\n",result);
else
     printf("element not found in the list\n");
return 0;
```

```
}
```

Output:

```
element found at index = 4


=== Code Execution Successful ===
```

3. **Aim:** Write a C program to search an element in the given sorted list using recursive Binary Search technique.

Program:

```
#include<stdio.h>
int BinarySearch(int arr[],int left,int right,int x)
{
if(right>=left)
{
  int mid=(left+right)/2;
  if(arr[mid]==x)
      return mid;
  if (arr[mid]>x)
     return BinarySearch(arr,left,mid-1,x);
 return BinarySearch(arr,mid+1,right,x);
 }
return -1;
}
int main()
{
int arr[]={2,3,4,10,40};
int n=sizeof(arr)/sizeof(arr[0]);
int x=4;
int result=BinarySearch(arr,0,n-1,x);
if(result!=-1)
     printf("element found at index = %d\n",result);
else
     printf("element not found in the list\n");
```

```
return 0;

}
```

Output:

```
element found at index = 2


=== Code Execution Successful ===
```

4. **Aim:** Write a C program to search an element in the given sorted list using non-recursive Binary Search technique.

Program:

```
#include<stdio.h>
int BinarySearch(int arr[],int size,int target)
{
int left=0;
int right=size-1;
while(left<=right)
{
  int mid=(left+right)/2;
  if(arr[mid]==target)
      return mid;
  if (arr[mid]<target)
     left=mid+1;
 else
     right=mid-1;
 }
return -1;
}
int main()
{
int arr[]={2,3,4,10,40};
int size=sizeof(arr)/sizeof(arr[0]);
int target=4;
int result=BinarySearch(arr,size,target);
if(result!=-1)
```

```
        printf("element found at index = %d\n",result);
else
        printf("element not found in the list\n");
return 0;
}
```

Output:

```
element found at index = 3


=== Code Execution Successful ===
```

Inferences:

- Linear search is a method of checking each element of the array **sequentially** until the element is found or the list ends.
- Recursive linear search is simple to implement and works on unsorted array.
- Non-recursive linear search uses a simple **loop** to scan through the array.
- In this non-recursive linear search, No recursion overhead but still **O(n)** time complexity.
- Binary search is a method Works only on **sorted arrays**. Repeatedly divide the array into halves and check the middle element.
- Recursive binary search has time complexity **O (log n)** that is much faster than linear search and has clear recursive structure.
- Non recursive binary search is implemented using a **while loop** (iterative approach).
- No recursion overhead (stack saving) and still **O (log n)** and efficient.

WEEK-3

Date:02-07-2025

List of programs:

1. Write a C program to sort a given list of integers using Bubble Sort Technique.
2. Write a C program to sort a given list of integers using Selection Sort Technique
3. Write a C program to sort a given list of integers using Insertion Sort Technique

1. **Aim:** To write a C program to sort a given list of integers using Bubble Sort Technique.

Program:

```
#include<stdio.h>
int main()
{
int i,j,n,a[100],temp;
printf("enter the number of elements");
scanf("%d",&n);
printf("enter %d elements\n",n);
for (i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for (i=0;i<n;i++)
{
for (j=0;j<n-1;j++)
{
if (a[j]>a[j+1])
{
temp= a[j];
a[j]=a[j+1];
```

```
a[j+1]=temp;
}
}
}
printf("elements after sorting:\n");
for (i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
return 0;
}
```

Output:

```
enter the number of elements4
enter 4 elements
12
4
13
24
elements after sorting:
4
12
13
24


=== Code Execution Successful ===
```

2. **Aim:** Write a C program to sort a given list of integers using Selection Sort Technique.

Program:

```
#include<stdio.h>
int main()
{
int i,j,n,a[100],t,min,k;
printf("enter the number of elements");
scanf("%d",&n);
printf("enter %d elements\n",n);
for (i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for (i=0;i<n;i++)
{
min=i;
for (j=i+1;j<n;j++)
{
if (a[min]>a[j])
{
min=j;
if(min!=k)
{
t=a[i];
a[i]=a[min];
a[min]=t;
}
```

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

Output:

```
enter the number of elements5
enter 5 elements
12
23
1
13
4
elements after sorting:
4
1
13
12
23


=== Code Execution Successful ===
```

3. **Aim:** Write a C program to sort a given list of integers using Insertion Sort Technique.

Program:

```
#include <stdio.h>
int main()
{
    int i, j, n, a[100], temp;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i = 1; i < n; i++)
    {
        temp = a[i];
        for (j = i - 1; j >= 0 && a[j] > temp; j--)
        {
            a[j + 1] = a[j];
        }

        a[j + 1] = temp;
    }

    printf("Elements after sorting:\n");
    for (i = 0; i < n; i++)
    {
```

```
        printf("%d\n", a[i]);
    }
    return 0;
}
```

Output:

```
Enter the number of elements: 4
Enter 4 elements:
23
56
3
11
Elements after sorting:
3
11
23
56


=== Code Execution Successful ===
```

Inferences:

- Bubble sort working principle is Repeatedly compares adjacent elements and swaps them if they are in the wrong order.

- Bubble Sort is **easy but inefficient**. It is stable, adaptive (with optimization), and good for learning or very small datasets, but not used in practice for large-scale sorting.

- Selection sort working priciple is **Working Principle** Finds the **minimum (or maximum)** element from the unsorted part of the array. Places it at the correct position by swapping with the first unsorted element. Repeats until the array is sorted.

- Selection Sort is simple and requires fewer swaps than Bubble Sort, but it is **always O(n^2)** and **not adaptive**. It is **inefficient for large datasets**, but sometimes used when **swapping is expensive**.

- Insertion Sort is **better than Bubble and Selection Sort** for **small or nearly sorted arrays**. It is **stable, adaptive, and simple**, but still **O(n^2)** for large unsorted datasets, making it impractical for large-scale sorting.

WEEK-4

Date:16-07-2025

List of programs:

1. Write a C program to sort a given list of integers using Quick Sort Technique.
2. Write a C program to sort a given list of integers using Merge Sort Technique

2. Aim: To write a C program to sort a given list of integers using Quick Sort Technique.

Program:

```
#include <stdio.h>

void quicksort(int A[], int low, int high);

int partition(int A[], int low, int high);

void quicksort(int A[], int low, int high)

{

   if (low < high)

   {

      int m = partition(A, low, high);

      quicksort(A, low, m - 1);

      quicksort(A, m + 1, high);

   }

}


int partition(int A[], int low, int high)

{

   int pivot = A[low];

   int i = low + 1;

   int j = high;

   int temp;

   while (i < j)
```

```
    {
        while ( A[i] <= pivot)
            i++;
        while (A[j] > pivot)
            j--;

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    }
    A[low] = A[j];
    A[j] = pivot;
    pivot=temp;
    return j;
}

int main()
{
    int i, n, A[20];
    printf("Enter how many elements: ");
    scanf("%d", &n);

    printf("Enter %d elements: ", n);
    for (i = 0; i < n; i++)
    {
```

```
        scanf("%d", &A[i]);
    }

    quicksort(A, 0, n - 1);

    printf("Elements after sorting:\n");
    for (i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter how many elements: 4
Enter 4 elements: 12
2
4
8
Elements after sorting:
2 4 8 12


=== Code Execution Successful ===
```

2. **Aim:** Write a C program to sort a given list of integers using Merge Sort Technique.

Program:

```
#include<stdio.h>
int main()
{
   int a[100],b[100],c[100],m,n,i,j,k;
   printf("how many elements do you want to read into list 1:");
   scanf("%d",&m);
   printf("how many elements do you want to read into list 2:");
   scanf("%d",&n);
   printf("enter %d elements in list 1:\n",m);
   for(i=0;i<m;i++)
   {
      scanf("%d",&a[i]);
   }
   printf("enter %d elements in list 2:\n",n);
   for(j=0;j<n;j++)
   {
      scanf("%d",&b[j]);
   }
   i=j=k=0;
   while(i<m&&j<n)
   {
      if(a[i]<b[j])
           c[k++]=a[i++];
     else
          c[k++]=b[j++];
```

```
    }
    while(i<m)
        c[k++]=a[i++];
    while(j<n)
        c[k++]=b[j++];
    printf("elements after merging:\n");
    for(k=0;k<m+n;k++)
    {
        printf("%d\n",c[k]);
    }
    return 0;
}
```

Output:

```
how many elements do you want to read into list 1:3
how many elements do you want to read into list 2:3
enter 3 elements in list 1:
12
15
27
enter 3 elements in list 2:
1
34
55
elements after merging:
1
12
15
27
34
55


=== Code Execution Successful ===
```

Inferences:

- Quick sort working principle is **Divide and Conquer** approach. It picks a **pivot** element, partitions the array into two halves (elements less than pivot & elements greater than pivot), and recursively sorts them.
- Quick Sort is **efficient, fast, and widely used in practice** with an average of **O(n log n)**. However, it is **not stable**, and poor pivot selection can lead to **$O(n^2)$** performance.
- Merge sort working principle is **Divide and Conquer** strategy. Recursively splits the array into halves until single elements remain, then **merges** them back in sorted order.
- Merge Sort is a **stable, guaranteed O(n log n)** sorting algorithm, excellent for **large datasets** and **linked lists**, but it needs **extra space (O(n))**. Unlike Quick Sort, its performance doesn't degrade to $O(n^2)$.

WEEK-5

Date:23-07-2025

List of programs:

1. Write a C program to insert a node at the beginning in a single linked list.
2. Write a C program to insert a node at the end in a single linked list.
3. Write a C program to insert a node after a given node(middle case) in a single linked list.
4. Write a C program to delete a node at the beginning in a single linked list.
5. Write a C program to delete a node at the end in a single linked list
6. Write a C program to delete a node after a given node(middle case) in a single linked list.

1. **Aim:** To write a C program to insert a node at the beginning in a single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL,*temp,*new=NULL;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
return temp;
```

```
}
void insert_begin()
{
if(head==NULL)
   head=new;
else
{
new->next=head;
head=new;
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
}
printf("null\n");
}
```

```
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=new;
temp=head;
}
else
{
temp->next=new;
temp=new;
}
}
printf("enter the data to insert at beginning");
scanf("%d",&x);
new=getnode(x);
insert_begin();
printf("the list after inserting at beginning\n");
display();
return 0;
```

```
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:100
enter the data in 2 node:200
enter the data in 3 node:300
enter the data in 4 node:400
enter the data in 5 node:500
enter the data to insert at beginning10
the list after inserting at beginning
10->100->200->300->400->500->null


=== Code Execution Successful ===
```

2. **Aim:** Write a C program to insert a node at the end in a single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL,*temp,*new=NULL;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
return temp;
}
void insert_end(int x)
{
new=getnode(x);
if(head==NULL)
{
head=new;
temp=head;
}
else
{
```

```
temp->next=new;
temp=new;
}
}
void display()
{
if(head==NULL)
{
printf("list is empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("null\n");
}
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
```

```
printf("enter the data in %d node:",i+1);

scanf("%d",&x);

insert_end;

}

printf("the list after insertion:\n");

display();

return 0;

}
```

Output:

```
Enter the number of nodes: 4
Enter the data for node 1: 11
Enter the data for node 2: 22
Enter the data for node 3: 33
Enter the data for node 4: 44
The list after insertion:
11->22->33->44->NULL


=== Code Execution Successful ===
```

3. **Aim:** Write a C program to insert a node after a given node(middle case) in a single linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node *next;

};

struct node *head=NULL,*temp,*new;

struct node *getnode(int x)

{

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

temp->data=x;

temp->next=NULL;

return temp;

}

void insert_middle(int val,int y)

{

struct node *temp=head;

while(temp->data!=val)

{

temp=temp->next;

}

struct node * new=getnode(y);
```

```
new->next=temp->next;
temp->next=new;
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("null\n");
}
}
int main()
{
int n,i,x,val,y;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
```

```
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=new;
temp=head;
}
else
{
temp->next=new;
temp=temp->next;
}}
printf("the list before insertion:\n");
display();
printf("enter the value in which node after you want to insert:");
scanf("%d",&val);
printf("enter the value you want to insert");
scanf("%d",&y);
insert_middle(val,y);
printf("the list after insertion:\n");
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:9
enter the data in 2 node:18
enter the data in 3 node:27
enter the data in 4 node:36
enter the data in 5 node:45
the list before insertion:
9->18->27->36->45->null
enter the value in which node after you want to insert:18
enter the value you want to insert20
the list after insertion:
9->18->20->27->36->45->null


=== Code Execution Successful ===
```

4. **Aim:** To write a C program to delete a node at the beginning in a single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL,*temp,*new,*last;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
return temp;
}
void delete_begin()
{
if(head==NULL)
{
printf("deletion is not possible");
}
else
{
```

```
temp=head;
head=head->next;
free(temp);
printf("node deleted from beginning");
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
printf("null\n");
}
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
```

```
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
last->next=new;
last=new;
}
}
printf("the list after creation:\n");
display();
delete_begin();
printf(" list after deleting first node:\n");
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:3
enter the data in 2 node:6
enter the data in 3 node:9
enter the data in 4 node:12
enter the data in 5 node:15
the list after creation:
3->6->9->12->15->null
node deleted from beginning list after deleting first node:
6->9->12->15->null



=== Code Execution Successful ===
```

5. **Aim:** Write a C program to delete a node at the end in a single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL,*temp=NULL,*new;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
return temp;
}
void delete_end()
{
if(head==NULL)
{
printf("deletion is not possible");
}
else if(head->next==NULL)
{
temp=head;
head=NULL;
```

```
free(temp);
}
else
{
struct node*temp1=NULL;
temp=head;
while(temp->next!=NULL)
{
temp1=temp;
temp=temp->next;
}
temp1->next=NULL;
free(temp);
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
```

```
}
printf("null\n");
}
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
}
}
```

```
printf(" list after deleting end node:\n");

delete_end();

display();

return 0;

}
```

Output:

```
enter the number of node:5
enter the data in 1 node:2
enter the data in 2 node:4
enter the data in 3 node:6
enter the data in 4 node:8
enter the data in 5 node:10
 list after deleting end node:
2->4->6->8->null


=== Code Execution Successful ===
```

6. **Aim:** Write a C program to delete a node after a given node(middle case) in a single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head=NULL,*temp,*new;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
return temp;
}
void delete_middle(int val)
{
if(head==NULL)
{
printf("deletion is not possible");
}
else if(head->next==NULL)
{
printf("deletion from middle is not possible");
```

```
}
else
{
struct node*temp1=NULL;
temp=head;
while(temp->data!=val)
{
temp1=temp;
temp=temp->next;
}
temp1->next=temp->next;
free(temp);
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
```

```
printf("null\n");
}
}
int main()
{
int n,i,x,val;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
}
}
printf("enter the value that you want to delete:\n");
```

```
scanf("%d",&val);
printf(" list after deleting middle node:\n");
delete_middle(val);
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:4
enter the data in 2 node:8
enter the data in 3 node:12
enter the data in 4 node:16
enter the data in 5 node:20
enter the value that you want to delete:
12
 list after deleting middle node:
4->8->16->20->null


=== Code Execution Successful ===
```

Inferences:

- List size is **dynamic** that is nodes created at runtime using `malloc`.
- Nodes are **not stored in contiguous memory** unlike arrays.
- Traversal is **one-way only** that is forward, from `head` to `NULL`.
- Last node's `next` pointer is always **NULL** → end of list.
- **Head pointer is important** because losing it means losing the list.
- Insertion and deletion are **easier and faster** than arrays.
- Searching/traversing is **slower** → O(n) time complexity.
- Each node requires **extra memory** for the pointer (`struct node* next`).
- Cannot access elements randomly. It has only sequential access.
- Commonly used in **stacks, queues, graphs, and dynamic memory structures**.

WEEK-6

Date:30-07-2025

List of programs:

1. Write a C program to insert a node at the beginning in a Circular single linked list.
2. Write a C program to insert a node at the end in a Circular single linked list.
3. Write a C program to insert a node after a given node(middle case) in a Circular single linked list.
4. Write a C program to delete a node at the beginning in a Circular single linked list.
5. Write a C program to delete a node at the end in a Circular single linked list
6. Write a C program to delete a node after a given node(middle case) in a Circular single linked list.

1. Aim: To write a C program to insert a node at the beginning in a Circular single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node*next;
};
struct node*head=NULL,*new,*temp,*last=NULL;
struct node*getnode(int x)
{
   struct node*temp=(struct node*)malloc(sizeof(struct node));
```

```
    temp->data=x;
    temp->next=NULL;
    return temp;
}
void insert_begin()
{
    if(head==NULL)
    {
        head=new;
        last=new;
        new->next=head;
    }
    else
    {
        new->next=head;
        head=new;
        last->next=head;
    }
}
void display()
{
    if(head==NULL)
    {
        printf("list is empty\n");
    }
    else
    {
        temp=head;
```

```
        do
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }
        while(temp!=head);
        printf("head");
    }
}
int main()
{
    int n,x,i;
    printf("enter no.of nodes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter data value in %d node:",i+1);
        scanf("%d",&x);
        new=getnode(x);
        if(head==NULL)
        {
            head=new;
            last=new;
            new->next=head;
        }
        else
        {
            last->next=new;
```

```
        last=new;
        last->next=head;
      }
   }
   printf("list after creation:\n");
   display();
   printf("\n enter the values to insert at beginning:\n");
   scanf("%d",&x);
   new=getnode(x);
   insert_begin();
   printf("list after insertion:\n");
   display();
   return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:10
enter data value in 2 node:15
enter data value in 3 node:20
enter data value in 4 node:25
enter data value in 5 node:30
list after creation:
10->15->20->25->30->head
 enter the values to insert at beginning:
5
list after insertion:
5->10->15->20->25->30->head


=== Code Execution Successful ===
```

2. **Aim:** Write a C program to insert a node at the end in a Circular single linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

   int data;

   struct node*next;

};

struct node*new,*last=NULL,*head=NULL,*temp;

struct node*getnode(int x)

{

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

   temp->data=x;

   temp->next=NULL;

   return temp;

}

void insert_end()

{

   if(head==NULL)

   {

      head=last=new;

      last->next=head;

   }

   else

   {

      last->next=new;
```

```
        last=new;
        last->next=head;
    }
}
void display()
{
    if(head==NULL)
    {
        printf("list is empty\n");
    }
    else
    {
        struct node*temp=head;
        do
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }
        while(temp!=head);
        printf("head\n");
    }
}
int main()
{
    int n,x,i;
    printf("enter no.of nodes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
```

```
    {
        printf("enter data value in %d node:",i+1);
        scanf("%d",&x);
        new=getnode(x);
        if(head==NULL)
        {
            head=last=new;
            last->next=head;
        }
        else
        {
            last->next=new;
            last=new;
            last->next=head;
        }
    }
    printf(" The list after creation:\n");
    display();
    printf(" Enter data to insert at end:");
    scanf("%d",&x);
    new=getnode(x);
    insert_end();
    printf(" The list after insertion:\n");
    display();
    return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:6
enter data value in 2 node:12
enter data value in 3 node:18
enter data value in 4 node:24
enter data value in 5 node:30
 The list after creation:
6->12->18->24->30->head
 Enter data to insert at end:36
 The list after insertion:
6->12->18->24->30->36->head



=== Code Execution Successful ===
```

3. Aim: Write a C program to insert a node after a given node(middle case) in a Circular single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node*next;
};
struct node*new,*last,*head=NULL,*temp;
struct node*getnode(int x)
{
   struct node*temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
   temp->next=NULL;
   return temp;
}
void insert_middle(int x,int val)
{
   temp=head;
   new=getnode(x);
   if(head==NULL)
   {
      printf("insertion is not possible \n");
   }
   else
```

```
      {
        while(temp->data!=val&&temp->next!=head)
        {
            temp=temp->next;
        }
        if(temp->data==val)
        {
            new->next=temp->next;
            temp->next=new;
        }
      }
  }
  void display()
  {
      if(head==NULL)
      {
          printf("list is empty\n");
      }
      else
      {
          struct node*temp=head;
          do
          {
              printf("%d->",temp->data);
              temp=temp->next;
          }
          while(temp!=head);
          printf("head\n");
```

```
    }
  }
  int main()
  {
    int n,x,i,val;
    printf("enter no.of nodes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
      printf("enter data value in %d node:",i+1);
      scanf("%d",&x);
      new=getnode(x);
      if(head==NULL)
      {
        head=new;
        last=new;
        last->next=head;
      }
      else
      {
        last->next=new;
        last=new;
        last->next=head;
      }
    }
    printf(" The list after creation:\n");
    display();
    printf("enter the value after which node you want to insert:");
```

```
    scanf("%d",&val);
    printf(" Enter data to insert at end:");
    scanf("%d",&x);
    insert_middle(x,val);
    printf(" The list after insertion:\n");
    display();
    return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:7
enter data value in 2 node:14
enter data value in 3 node:21
enter data value in 4 node:28
enter data value in 5 node:35
 The list after creation:
7->14->21->28->35->head
enter the value after which node you want to insert:14
 Enter data to insert at end:17
 The list after insertion:
7->14->17->21->28->35->head


=== Code Execution Successful ===
```

4. **Aim:** To write a C program to delete a node at the beginning in a Circular single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node*next;
};
struct node*new,*last,*head,*temp;
struct node*getnode(int x)
{
   struct node*temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
   temp->next=NULL;
   return temp;
}
void delete_begin()
{
   if(head==NULL)
   {
      printf("deletion is not possible \n");
   }
   else if(head->next==head)
   {
```

```
        temp=head;
        head=NULL;
        free(temp);
    }
    else
    {
      temp=head;
      head=head->next;
      last->next=head;
      free(temp);
    }
}
void display()
{
    if(head==NULL)
    {
        printf("list is empty\n");
    }
    else
    {
        struct node*temp=head;
        do
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }
        while(temp!=head);
        printf("head\n");
```

```
    }
}
int main()
{
    int n,x,i;
    printf("enter no.of nodes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter data value in %d node:",i+1);
        scanf("%d",&x);
        new=getnode(x);
        if(head==NULL)
        {
            head=new;
            last=new;
            last->next=head;
        }
        else
        {
            last->next=new;
            last=new;
            last->next=head;
        }
    }
    printf(" The list after creation:\n");
    display();
    delete_begin();
```

```
    printf(" The list after deletion at begin:\n");
    display();
    return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:8
enter data value in 2 node:16
enter data value in 3 node:24
enter data value in 4 node:32
enter data value in 5 node:40
 The list after creation:
8->16->24->32->40->head
 The list after deletion at begin:
16->24->32->40->head


=== Code Execution Successful ===
```

5. **Aim:** Write a C program to delete a node at the end in a Circular single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
  int data;
  struct node*next;
};
struct node*new,*last=NULL,*head,*temp;
struct node*getnode(int x)
{
  struct node*temp=(struct node*)malloc(sizeof(struct node));
  temp->data=x;
  temp->next=NULL;
  return temp;
}
void delete_end()
{
  if(head==NULL)
  {
    printf("deletion is not possible \n");
  }
  else if(head->next==head)
  {
    temp=head;
    head=last=NULL;
```

```
        free(temp);
      }
      else
      {
       temp=head;
       struct node*temp2=NULL;
       while(temp->next!=head)
       {
          temp2=temp;
          temp=temp->next;
       }
       last=temp2;
       last->next=head;
       free(temp);
      }
   }
   void display()
   {
      if(head==NULL)
      {
        printf("list is empty\n");
      }
      else
      {
        struct node*temp=head;
        do
        {
          printf("%d->",temp->data);
```

```
          temp=temp->next;
        }
        while(temp!=head);
        printf("head\n");
    }
}
int main()
{
    int n,x,i;
    printf("enter no.of nodes:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter data value in %d node:",i+1);
        scanf("%d",&x);
        new=getnode(x);
        if(head==NULL)
        {
            head=new;
            last=new;
            last->next=head;
        }
        else
        {
           temp=head;
           while(temp->next!=head)
           {
               temp=temp->next;
```

```
            }
            temp->next=new;
            last=new;
            last->next=head;
          }
      }
      printf(" The list after creation:\n");
      display();
      delete_end();
      printf(" The list after deletion at end:\n");
      display();
      return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:10
enter data value in 2 node:20
enter data value in 3 node:30
enter data value in 4 node:40
enter data value in 5 node:50
 The list after creation:
10->20->30->40->50->head
 The list after deletion at end:
10->20->30->40->head


=== Code Execution Successful ===
```

6. Aim: Write a C program to delete a node after a given node(middle case) in a Circular single linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node*next;
};
struct node*head,*new,*temp,*last=NULL;
struct node*getnode(int x)
{
   struct node*temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
   temp->next=NULL;
   return temp;
}
void display()
{
   if(head==NULL)
   {
    printf("list is empty\n");
   }
   else
   {
      temp=head;
```

```
        do
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }
        while(temp!=head);
        printf("head");
    }
}
void delete_middle(int val)
{
    if(head==NULL)
    {
        printf("list is empty\n");
    }
    temp=head;
    struct node *temp1=NULL;
    while(temp->data!=val)
    {
        if(temp->next==head)
        {
            printf("node%not found\n",val);
        }
        temp1=temp;
        temp=temp->next;
    }
    if(temp==head||temp->next==head)
    {
```

```
        printf("node value%d is not a middle node value",val);
      }
      temp1->next=temp->next;
      free(temp);
    }
    int main()
    {
      int n,x,i,val;
      printf("enter no.of nodes:");
      scanf("%d",&n);
      for(i=0;i<n;i++)
      {
        printf("enter data value in %d node:",i+1);
        scanf("%d",&x);
        new=getnode(x);
        if(head==NULL)
        {
          head=new;
          last->next=head;
        }
        else
        {
          temp=head;
          while(temp->next!=head)
          {
            temp=temp->next;
          }
          temp->next=new;
```

```
            last=new;
            last->next=head;
          }
        }
        printf("Enter the value to delete at middle:\n");
        scanf("%d",&val);
        printf("The list after deletion:");
        delete_middle(val);
        display();
        return 0;
    }
```

Output:

```
enter the number of node:5
enter the data in 1 node:4
enter the data in 2 node:8
enter the data in 3 node:12
enter the data in 4 node:16
enter the data in 5 node:20
enter the value that you want to delete:
12
 list after deleting middle node:
4->8->16->20->null


=== Code Execution Successful ===
```

Inferences:

- In circular linked list, the **last node does not point to NULL** – it points back to the **head node**.
- Traversal can be done starting from **any node** since the list is circular.
- There is **no natural end** (need a condition to stop traversal, e.g., when pointer reaches head again).
- **Head pointer is still important**, but even if head is lost, list can still be traversed from any known node.
- **Efficient for round-robin scheduling** (CPU scheduling, buffering, etc.).
- Insertion at the **beginning or end** can be done in **O(1)** time if a tail pointer is maintained.
- Same as singly linked list, requires **extra memory for next pointer**.
- Searching is still **O(n)** since traversal may need to cover the whole list.
- More flexible than linear singly linked list in **repeated traversals** (no need to restart at head).

WEEK-7

Date:06-08-2025

List of programs:

1. Write a C program to insert a node at the beginning in a Double linked list.
2. Write a C program to insert a node at the end in a Double linked list.
3. Write a C program to insert a node after a given node(middle case) in a Double linked list.
4. Write a C program to delete a node at the beginning in a Double linked list.
5. Write a C program to delete a node at the end in a Double linked list
6. Write a C program to delete a node after a given node(middle case) in a Double linked list.

1. Aim: To write a C program to insert a node at the beginning in a Double linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node *next;

struct node *prev;

};

struct node *head,*temp,*new,*last=NULL;

struct node *getnode(int x)

{

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

temp->data=x;

temp->next=NULL;
```

```
temp->prev=NULL;
return temp;
}
void insert_begin(int x)
{
new=getnode(x);
if(head==NULL)
   head=last=new;
else
{
new->next=head;
head->prev=new;
head=new;
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
```

```
}
}
printf("null\n");
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
new->prev=temp;
last=new;
```

```
}
}
printf("enter the data to insert at beginning");
scanf("%d",&x);
printf("the list after inserting at beginning\n");
insert_begin(x);
display();
return 0;
}
```

Output:

```
enter the number of node:4
enter the data in 1 node:13
enter the data in 2 node:26
enter the data in 3 node:39
enter the data in 4 node:42
enter the data to insert at beginning3
the list after inserting at beginning
3->13->26->39->42->null


=== Code Execution Successful ===
```

2. **Aim:** Write a C program to insert a node at the end in a Double linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
struct node *prev;
};
struct node *head,*temp,*new,*last;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
temp->prev=NULL;
return temp;
}
void insert_end(int x)
{
new=getnode(x);
if(head==NULL)
   head=last=new;
else
{
temp=head;
```

```
while(temp->next!=NULL)
{
temp=temp->next;
}
last=temp;
last->next=new;
new->prev=last;
last=new;
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
}
printf("null\n");
}
int main()
```

```
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
new->prev=temp;
last=new;
last->next=NULL;
}
}
printf("enter the data to insert at end:");
scanf("%d",&x);
```

```
printf("the list after inserting at end:\n");
insert_end(x);
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:14
enter the data in 2 node:28
enter the data in 3 node:42
enter the data in 4 node:56
enter the data in 5 node:70
enter the data to insert at end:84
the list after inserting at end:
14->28->42->56->70->84->null


=== Code Execution Successful ===
```

3. Aim: Write a C program to insert a node after a given node(middle case) in a Double linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node *next;

struct node *prev;

};

struct node *head,*temp,*new,*last=NULL;

struct node *getnode(int x)

{

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

temp->data=x;

temp->next=NULL;

temp->prev=NULL;

return temp;

}

void insert_middle(int x,int val)

{

new=getnode(x);

if(head==NULL)

   head=last=new;

else if(last->data==val)

{

last->next=new;
```

```
new->prev=last;
last=new;
}
else
{
temp=head;
while(temp->data!=val&&temp!=NULL)
{
temp=temp->next;
}
new->next=temp->next;
(temp->next)->prev=new;
temp->next=new;
new->prev=temp;
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
```

```
temp=temp->next;
}
}
printf("null\n");
}
int main()
{
int n,i,x,val;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
new->prev=temp;
```

```
last=new;
last->next=NULL;
}
}
printf("enter the value of node after which you want to insert:\n");
scanf("%d",&val);
printf("enter the data to insert at end:");
scanf("%d",&x);
printf("the list after inserting at end:\n");
insert_middle(x,val);
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:15
enter the data in 2 node:30
enter the data in 3 node:45
enter the data in 4 node:60
enter the data in 5 node:75
enter the value of node after which you want to insert:
45
enter the data to insert at end:55
the list after inserting at end:
15->30->45->55->60->75->null


=== Code Execution Successful ===
```

4. **Aim:** To write a C program to delete a node at the beginning in a Double linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
struct node *prev;
};
struct node *head,*temp,*new,*last;
struct node *getnode(int x)
{
struct node *temp=(struct node*)malloc(sizeof(struct node));
temp->data=x;
temp->next=NULL;
temp->prev=NULL;
return temp;
}
void delete_begin()
{
if(head==NULL)
 {
   printf("list is empty");
 }
else if(head->next==NULL)
{
```

```
temp=head;
head=last=NULL;
free(temp);
}
else
{
temp=head;
head=head->next;
head->prev=NULL;
free(temp);
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
}
printf("null\n");
```

```
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
new->prev=temp;
last=new;
last->next=NULL;
}
}
```

```
printf("the list after deleting at begin:\n");
delete_begin();
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:16
enter the data in 2 node:32
enter the data in 3 node:48
enter the data in 4 node:64
enter the data in 5 node:80
the list after deleting at begin:
32->48->64->80->null


=== Code Execution Successful ===
```

5. **Aim:** Write a C program to delete a node at the end in a Double linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node *next;

struct node *prev;

};

struct node *head,*temp,*new,*last=NULL;

struct node *getnode(int x)

{

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

temp->data=x;

temp->next=NULL;

temp->prev=NULL;

}

void delete_end()

{

if(head==NULL)

 {

  printf("list is empty");

 }

else if(head->next==NULL)

{

 temp=head;
```

```
head=last=NULL;
free(temp);
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp=last;
last=last->prev;
last->next=NULL;
free(temp);
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
```

```
temp=temp->next;
}
}
printf("null\n");
}
int main()
{
int n,i,x;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
temp=temp->next;
}
temp->next=new;
new->prev=temp;
```

```
last->next=NULL;
}
}
printf("the list after deleting at end:\n");
delete_end();
display();
return 0;
}
```

Output:

```
enter no.of nodes:5
enter data value in 1 node:10
enter data value in 2 node:20
enter data value in 3 node:30
enter data value in 4 node:40
enter data value in 5 node:50
 The list after creation:
10->20->30->40->50->head
 The list after deletion at end:
10->20->30->40->head


=== Code Execution Successful ===
```

6. Aim: Write a C program to delete a node after a given node(middle case) in a Double linked list.

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node *next;

struct node *prev;

};

struct node *head,*temp,*new,*last=NULL;

struct node *getnode(int x)

{

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

temp->data=x;

temp->next=NULL;

temp->prev=NULL;

}

void delete_middle(int val)

{

if(head==NULL)

 {

   printf("list is empty");

 }

else if(head->next==NULL)

{
```

```
 temp=head;
head=last=NULL;
free(temp);
}
else
{
temp=head;
while(temp!=NULL&&temp->data!=val)
{
temp=temp->next;
}
if(temp==NULL)
{
printf("node with value %d not found \n",val);
}
(temp->prev)->next=temp->next;
(temp->next)->prev=temp->prev;
free(temp);
}
}
void display()
{
if(head==NULL)
{
printf("list empty\n");
}
else
{
```

```
temp=head;
while(temp!=NULL)
{
printf("%d->",temp->data);
temp=temp->next;
}
}
printf("null\n");
}
int main()
{
int n,i,x,val;
printf("enter the number of node:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("enter the data in %d node:",i+1);
scanf("%d",&x);
new=getnode(x);
if(head==NULL)
{
head=last=new;
}
else
{
temp=head;
while(temp->next!=NULL)
{
```

```
temp=temp->next;
}
temp->next=new;
new->prev=temp;
last=new;
last->next=NULL;
}
}
printf("enter the value to delete at middle:\n");
scanf("%d",&val);
printf("the list after deleting at middle:\n");
delete_middle(val);
display();
return 0;
}
```

Output:

```
enter the number of node:5
enter the data in 1 node:17
enter the data in 2 node:34
enter the data in 3 node:51
enter the data in 4 node:68
enter the data in 5 node:85
enter the value to delete at middle:
51
the list after deleting at middle:
17->34->68->85->null


=== Code Execution Successful ===
```

Inferences:

- Each node contains **three fields** → `data`, `prev` pointer, and `next` pointer.
- Traversal is possible in **both directions** (forward and backward).
- **Insertion and deletion** are easier compared to singly list (can be done from both ends efficiently).
- **More memory required** per node (extra `prev` pointer).
- Searching is still **O(n)** in worst case.
- **Head node's** `prev` **= NULL** and **last node's** `next` **= NULL**.
- Losing head pointer still causes list inaccessibility, but tail pointer helps in reverse traversals.
- Better suited for applications where **bidirectional traversal** is required (like undo/redo in editors, navigation systems).
- Compared to singly linked list, it provides **more flexibility**, but at the cost of **extra memory** and **slightly more complex operations**.

WEEK-8

Date:03-09-2025

List of programs:

1. Write a C program to implement Stack operations using arrays.
2. Write a C program to implement Stack operations using linked list.

1. Aim: To write a C program to implement Stack operations using arrays.

Program:

```
#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 5

void push(int x);

void pop();

void display();

 int stack[MAXSIZE];
 int top=-1;
void main( )

{

  int ch,num;

  while(1)

  {

     printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n\nEnter your choice:");

     scanf("%d",&ch);

      switch(ch)

     {

       case 1: printf("\n\nENTER THE STACK ELEMENT :  ");


                scanf("%d",&num);
               push(num);
```

```
                break;
          case 2:  pop();
                   break;
          case 3:  display();
                   break;
        case 4:  exit(0);
                break;
        default:printf("Invalid Choice :");

        }

    }

  }

  void push(int item)

  {

    if(top==MAXSIZE-1)

    {

                printf("\nSTACK FULL");
                return;
        }

        else

        {

                    top++;
                    stack[top]=item;
                    printf("ELEMENT INSERTED\n");
        }
   }
 void pop()

 {

    int x;

    if(top==1)

     {

        printf("STACK EMPTY");

         return;

        }
```

```
        else
         {
           x=stack[top];
           printf("DELETED ELEMENT is %d\n",x);
           top--;
          }
}
void display()
{
          int i;
          if(top==-1)
          {
                    printf("\nSTACK EMPTY");
                    return;
          }
          else
          {
                    printf("\nSTACK ELEMENTS ARE...\n");
                    for(i=top;i>=0;i--)
                    {
                              printf("\n%d ",stack[i]);
                              if(i==top)
                              printf("---->TOP");
                    }
          }
     }
```

Output:

```
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:1


ENTER THE STACK ELEMENT :  10
ELEMENT INSERTED

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:1


ENTER THE STACK ELEMENT :  20
ELEMENT INSERTED

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:1


ENTER THE STACK ELEMENT :  30
ELEMENT INSERTED

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:3
```

```
STACK ELEMENTS ARE...

30 ---->TOP
20
10
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:2
DELETED ELEMENT is 30

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:3

STACK ELEMENTS ARE...

20 ---->TOP
10
```

```
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter your choice:4


=== Code Execution Successful ===
```

2. Aim: To Write a C program to implement Stack operations using linked list.

Program:

```
#include<stdio.h>
 #include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *top=NULL;
void push(int);
void pop();
void display();
void main()
{
 int ch,num;
 printf("\n:: Stack using Linked List ::\n");
 while(1)
 {
   printf(" \n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n\nEnter Your Choice:");
   scanf("%d",&ch);
   switch(ch)
   {
     case 1: printf("\n\nENTER THE STACK ELEMENT : ");
             scanf("%d",&num);
             push(num);
             break;
```

```
        case 2:  pop();
                 break;
        case 3:  display();
                 break;
        case 4:  exit(0);
                 break;
default: printf("Invalid Choice : ");
        }
    }
}
void display()
{
    struct node *temp = top;
     if(temp==NULL)
         printf("\nSTACK IS EMPTY\n");
       else
        {
          while(temp!=NULL)
           {
              printf("%d-->",temp->data);
              temp=temp->next;
             }
              printf("NULL");
       }
}
void push(int num)
{
  struct node *newNode = (struct node*)malloc(sizeof(struct node));
```

```
 newNode->data = num;
 newNode->next = NULL;
 if(top == NULL)
      top=newNode;
else
 {
   newNode->next = top;
   top = newNode;
  }
  printf("\nELEMENT IS INSERTED\n");
}
void pop()
{
  if(top == NULL)
     printf("\nSTACK IS EMPTY\n");
  else
  {
    temp = top;
    printf("\nDELETED ELEMENT IS %d\n",temp->data);
    top = temp->next;
    free(temp);
  }
}
```

Output:

```
:: Stack using Linked List ::

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:1
ENTER THE STACK ELEMENT : 100

ELEMENT IS INSERTED

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:1
ENTER THE STACK ELEMENT : 200

ELEMENT IS INSERTED

1.PUSH
2.POP
3.DISPLAY
```

```
4.EXIT

Enter Your Choice:1
ENTER THE STACK ELEMENT : 300

ELEMENT IS INSERTED

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:3
300-->200-->100-->NULL
```

```
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:2

DELETED ELEMENT IS 300

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:3
200-->100-->NULL
1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter Your Choice:4


=== Code Execution Successful ===
```

Inferences:

- A stack can be implemented using an array where elements are added/removed from one end (top).
- Advantages: Simple, fixed-size memory allocation.
- Disadvantages: Fixed size can lead to overflow; resizing can be costly.
- Stacks using Linked Lists Implementation: Nodes are dynamically allocated; top points to head node.

WEEK-9

Date:10-09-2025

List of programs:

1. Write a C program to convert an infix expression into postfix expression using Stacks.
2. Write a C program to evaluate postfix expression using Stacks.

1. **Aim:** Write a C program to convert an infix expression into postfix expression using Stacks.

Program:

```
#include<stdio.h>

#include<string.h>

#include <ctype.h>

#define SIZE 50

char s[SIZE];

int top=-1;

void push(char ele)

{

   s[++top]=ele;

}

char pop()

{

   return(s[top--]);

}

int priority(char ele)

{

   switch(ele)

   {

   case '$': return 0;
```

```
    case '(': return 1;
    case '+': return 2;
    case '-': return 2;
    case '*': return 3;
    case '/': return 3;
    }
}
void main()
{
    char infx[50],postfx[50],ch;
    int i=0,k=0;
    printf("Read the Infix Expression : ");
    scanf("%s",infx);
   push('$');
   while( (ch=infx[i++]) != '\0')
   {
    if( ch == '(')
          push(ch);
    else
          if(isalnum(ch))
              postfx[k++]=ch;
          else
               if( ch == ')')
                 {
                    while( s[top] != '(')
                           postfx[k++]=pop();
                     pop();
                 }
```

```
            else
            {
               while( priority(s[top]) >= priority(ch) )
                    postfx[k++]=pop();
                    push(ch);
              }
   }
   while( s[top] != '$')
     postfx[k++]=pop();
   postfx[k]='\0';
   printf("Given Infix Expn: %s  \n Postfix Expn: %s\n",infx,postfx);
}
```

Output:

```
Read the Infix Expression : a+b*c/d
Given Infix Expn: a+b*c/d
 Postfix Expn: abc*d/+


=== Code Exited With Errors ===
```

2. **Aim:** Write a C program to evaluate postfix expression using Stacks.

Program:

```
#include<stdio.h>
#include <stdlib.h>
#include<ctype.h>
int s[50];
int top=-1;
void push(int elem)
{
 s[++top]=elem;
}
int pop()
{
 return(s[top--]);
}
void main()
{
 char pofx[50],ch;
 int i=0,op1,op2;
 printf("\n\nEnter the Postfix Expression ");
 scanf("%s",pofx);
 while( (ch=pofx[i++]) !='\0')
 {
  if(isdigit(ch))
     push(ch-'0');
 else
  {
```

```
   op2=pop();
   op1=pop();
   switch(ch)
   {
   case '+':push(op1+op2);
            break;
   case '-':push(op1-op2);
           break;
   case '*':push(op1*op2);
            break;
   case '/':push(op1/op2);
            break;
   }
  }
 }
 printf(" Given Postfix Expn: %s\n",pofx);
 printf(" Result after Evaluation: %d\n",s[top]);
}
```

Output:

```
Enter the Postfix Expression 123*+
 Given Postfix Expn: 123*+
 Result after Evaluation: 7
```

Inferences:

- The working principle for infix to postfix conversion is Converts an **infix expression** (e.g., A + B) to **postfix** (A B +) using a stack. Operators are pushed to the stack; operands go directly to the output. Operator precedence and parentheses are managed via stack operations.
- Postfix evaluation is **fast (O(n))**, **simple**, and **stack-based**, making it ideal for **expression evaluation in compilers and calculators**.It avoids complexities of parentheses and operator precedence, unlike infix evaluation.

WEEK-10

Date:17-09-2025

List of programs:

1. Write a C program to implement Queue operations using arrays.
2. Write a C program to implement Queue operations using linked list.

1. Aim: To write a C program to implement Queue operations using arrays.

Program:

```
#include<stdio.h>

#include<stdlib.h>

#define size 4

int Queue[size];

int front=-1,rear=-1;

void enqueue(int x)

{

   if(rear==size-1)

   {

      printf("queue is full");

   }

   else

   {

      if(front==-1&&rear==-1)

      {

         front=rear=0;

      }

      else

      {
```

```
        rear++;
      }
      Queue[rear]=x;
    }
}
void dequeue()
{
    if(front==-1)
    {
      printf("queue is empty");
    }
    else
    {
      printf("deleted element is %d",Queue[front]);
      if(front==rear)
      {
        front=rear=-1;
      }
      else
      {
        front++;
      }
    }
}
void display()
{
    if(front==-1)
    {
```

```
            printf("queue is empty");
        }
        else
        {
            for(int i=front;i<=rear;i++)
            {
                printf("%d\n",Queue[i]);
            }
        }
    }
    void main()
    {
    int ch,num;
    printf("\n:: queue using arrays ::\n");
    while(1)
    {
    printf("\nMAIN MENU:\n1.enqueue\n2.dequeue\n3.DISPLAY\n4.EXIT\n\nENTER YOUR 
CHOICE:");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1: printf("ENTER THE QUEUE ELEMENT : ");
             scanf("%d",&num);
            enqueue(num);
            break;
    case 2: dequeue();
             break;
    case 3: display();
```

```
            break;
case 4:exit(0);
            break;
default:printf("Invalid Choice : ");
}
}
}
```

Output:

```
:: queue using arrays ::

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 12

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 24

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT
```

```
ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 36

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:3
12
24
36

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:2
deleted element is 12
MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:3
24
36

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:4


=== Code Execution Successful ===
```

2. Aim: To Write a C program to implement Queue operations using linked list.

Program:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *front=NULL,*rear=NULL,*new,*temp;
void enqueue(int x)
{
   new=(struct node*)malloc(sizeof(struct node));
   new->data=x;
   new->next=NULL;
   if(front==NULL&&rear==NULL)
   {
      front=rear=new;
   }
   else
   {
      rear->next=new;
      rear=new;
   }
}
void dequeue()
{
```

```
    if(front==NULL&&rear==NULL)
    {
        printf("queue is empty");
    }
    else
    {
        printf("deleted element is %d",front->data);
        temp=front;
        if(front==rear)
        {
            front=rear=NULL;
        }
        else
        {
            front=front->next;
        }
        free(temp);
    }
}
void display()
{
    temp=front;
    if(front==NULL)
    {
        printf("queue is empty");
    }
    else
    {
```

```
        while(front!=NULL)
        {
          printf("%d->",front->data);
          front=front->next;
        }
        printf("NULL");
      }
    }
    void main()
    {
    int ch,num;
    printf("\n:: queue using linked list ::\n");
    while(1)
    {
    printf("\nMAIN MENU:\n1.enqueue\n2.dequeue\n3.DISPLAY\n4.EXIT\n\nENTER YOUR
CHOICE:");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1: printf("ENTER THE QUEUE ELEMENT : ");
            scanf("%d",&num);
            enqueue(num);
            break;
    case 2: dequeue();
            break;
    case 3: display();
            break;
    case 4:exit(0);
```

```
            break;
default:printf("Invalid Choice : ");
}
}
}
```

Output:

```
:: queue using linked list ::

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 10

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 20

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT
```

```
ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 30

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:2
deleted element is 10
MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:3
20->30->NULL
MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT
```

```
ENTER YOUR CHOICE:4


=== Code Execution Successful ===
```

Inferences:

- Working principle for queues using arrays is Queue follows **FIFO (First In, First Out)** order. Elements are inserted at the **rear** and removed from the **front**.
- It is Easy to implement and it has constant time insertion and deletion.
- The main advantage of queues using linked list is:

    1. **Dynamic size** → no overflow (unless memory is full).
    2. No need for shifting elements (as in arrays).
    3. Efficient memory utilization.

WEEK-11

Date:24-09-2025

List of programs:

1. Write a C program to implement Circular Queue operations using arrays.
2. Write a C program to implement Recursive Binary Tree Traversals(In-Order, Pre-Order, Post-Order).

1.Aim: Write a C program to implement Circular Queue operations using arrays.

Program:

```
#include<stdio.h>
#include<stdlib.h>
#define size 3
int Queue[size];
int front=-1,rear=-1;
void enqueue(int x)
{
   if(front==(rear+1)%size)
   {
      printf("circular queue is full");
   }
   else
   {
      if(front==-1&&rear==-1)
      {
         front=rear=0;
      }
      else
      {
```

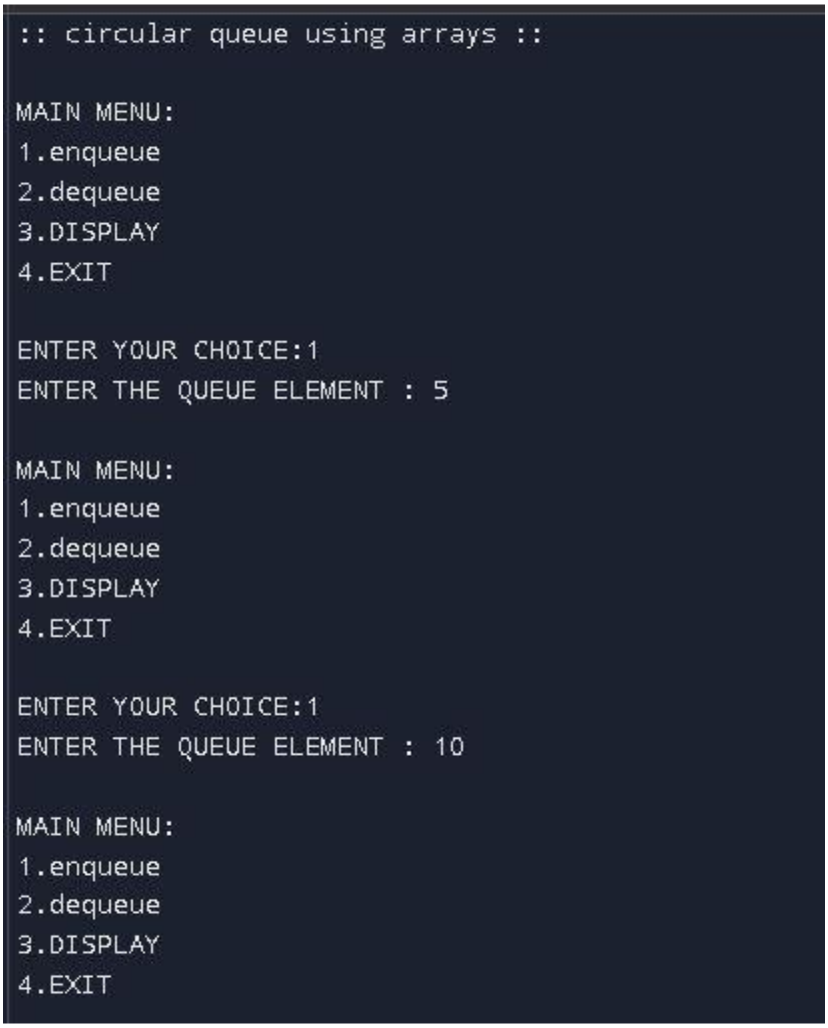
```
            rear=(rear+1)%size;
        }
        Queue[rear]=x;
    }
}
void dequeue()
{
     if(front==-1)
    {
        printf("queue is empty");
    }
    else
    {
        printf("deleted element is %d",Queue[front]);
        if(front==rear)
        {
            front=rear=-1;
        }
        else
        {
            front=(front+1)%size;
        }
    }
}
void display()
{
    int i;
    if(front==-1)
```

```
    {
        printf("queue is empty");
    }
    else
    {
        for(i=front;i!=rear;i=(i+1)%size)
        {
            printf("%d\n",Queue[i]);
        }
        printf("%d\n",Queue[i]);
    }
}

void main()
{
int ch,num;
printf("\n:: circular queue using arrays ::\n");
while(1)
{
printf("\nMAIN MENU:\n1.enqueue\n2.dequeue\n3.DISPLAY\n4.EXIT\n\nENTER YOUR CHOICE:");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("ENTER THE QUEUE ELEMENT : ");
          scanf("%d",&num);
          enqueue(num);
          break;
```

```
case 2: dequeue();
            break;
case 3: display();
             break;
case 4:exit(0);
           break;
default:printf("Invalid Choice : ");
}
}
}
```

Output:



```
:: circular queue using arrays ::

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 5

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 10

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT
```

```
ENTER YOUR CHOICE:1
ENTER THE QUEUE ELEMENT : 15

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:3
5
10
15

MAIN MENU:
1.enqueue
2.dequeue
3.DISPLAY
4.EXIT

ENTER YOUR CHOICE:2
deleted element is 5
MAIN MENU:
1.enqueue
2.dequeue
```

```
ENTER YOUR CHOICE:4


=== Code Execution Successful ===
```

2. Aim: To Write a C program to implement Recursive Binary Tree Traversals(In-Order, Pre-Order, Post-Order).

Program:

```
#include <stdio.h>
#include <stdlib.h>
  struct Node
{
   int data;
   struct Node* left;
   struct Node* right;
 } ;
 Struct node* createNode(int data)
  {
   Struct node* newNode = (struct node*)malloc(sizeof(struct node));
   if (newNode == NULL)
  {
      printf("Memory allocation failed!\n");
      exit(1);
   }
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
 }
 Struct node* insert(struct node* root, int data)
  {
   if (root == NULL)
   {
```

```
        return createNode(data);
    }
    if (data < root->data)
     {
        root->left = insert(root->left, data);
    }
     else if (data > root->data)
      {
        root->right = insert(root->right, data);
    }
else
    return root;
}
void inorderTraversal(struct node* root)
 {
    if (root != NULL)
 {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}
void preorderTraversal(struct node* root)
{
    if (root != NULL) {
        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
```

```
    }
}
void postorderTraversal(struct node* root)
 {
    if (root != NULL)
 {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->data);
    }
}
int main()
{
    Struct node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
    printf("In-order traversal: ");
    inorderTraversal(root);
    printf("\n");
    printf("Pre-order traversal: ");
    preorderTraversal(root);
    printf("\n");
    printf("Post-order traversal: ");
```

```
    postorderTraversal(root);

    printf("\n");

    return 0;

}
```

Output:

```
In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50


=== Code Execution Successful ===
```

Inferences:

- The main advantage of circular queues using arrays is no **wasted space** → unlike simple linear array queue where freed positions at the start cannot be reused and it has efficient memory utilization.
- The **recursive approach** simplifies both the **tree creation** and the **tree traversal** processes, making the code clean and easier to understand.
- The **in-order traversal (Left → Root → Right)** displays nodes in **sorted order** only if the binary tree satisfies the **Binary Search Tree (BST)** property.
- The **pre-order traversal (Root → Left → Right)** is useful for **copying or saving** the tree structure
- The **post-order traversal (Left → Right → Root)** is useful for **deleting** the entire tree or evaluating **expression trees**.
- Recursion ensures that every subtree is processed independently, following the same traversal logic — demonstrating **divide-and-conquer** principle.

WEEK-12

Date:08-10-2025

List of programs:

1. Write a C progam to implement various Operations on Binary Search Tree(Insertion,Search,Display).

1. **Aim:** To Write a C progam to implement various Operations on Binary Search Tree(Insertion,Search,Display).

Program:

```
#include<stdio.h>

#include<stdlib.h>

struct node

{

   int data;

   struct node *left, *right;

};

struct node *newNode(int item)

{

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

   temp->data = item;

   temp->left = temp->right = NULL;

   return temp;

}

void inorder(struct node *root)

{

   if (root != NULL)

   {

      inorder(root->left);
```

```
        printf("%d ", root->data);
        inorder(root->right);
    }
}
struct node* insert(struct node *root, int key)
{
    if (root == NULL)
        return newNode(key);
    if (key <= root->data)
        root->left=insert(root->left, key);
    else
        if (key > root->data)
            root->right=insert(root->right, key);
    return root;
}
struct node *search(struct node *temp, int key)
{
    if(temp==NULL)
    {
        printf("No key found for value - %d\n", key);
        return temp;
    }
   if(temp->data == key)
        printf("Key %d found\n", key);
    else if(temp->data < key)
        search(temp->right, key);
    else
        search(temp->left, key);
```

```
}
struct node *getInSuccessor(struct node *temp)
{
   while(temp->left != NULL)
      temp = temp->left;
   return temp;
}
struct node * deletion(struct node *root, int delKey)
{
   struct node *temp;
   if(root==NULL)
   {
      printf("Unable to delete. No such key exists.\n");
      return root;
   }
   else if(delKey > root->data)
      root->right = deletion(root->right, delKey);
   else if(delKey < root->data)
      root->left = deletion(root->left, delKey);
   else
   {
      if(root->left == NULL)
      {
         temp = root->right;
         free(root);
         return temp;
      }
      else if(root->right == NULL)
```

```
        {
            temp = root->left;
            free(root);
            return temp;
        }
        temp = getInSuccessor(root->right);
        root->data = temp->data;
        root->right = deletion(root->right, temp->data);
    }
    return root;
}
int main()
{
    int ch,n;
    struct node *root = NULL;
    while(1)
    {
        printf("Menu:\nBST
Operations:\n1.Insert\n2.Search\n3.Delete\n4.Display\n5.Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&ch);
       switch(ch)
        {
            case 1:printf("Enter the element to insert a new node: ");
                   scanf("%d",&n);
    if(root==NULL)
                   root=insert(root, n);
                 else
```

```
                insert(root,n);
              break;
        case 2: printf("Enter the element to search: ") ;
               scanf("%d",&n);
               search(root,n);
               break ;
        case 3:printf("Enter node value which you want to delete: ");
              scanf("%d",&n);
              deletion(root, n);
              break ;
        case 4: printf("Inorder Traversal of BST: ");
               inorder(root);
               break;
        case 5: exit(0);
               break;
        default: printf("Wrong Choice\n");
      }
    }
  }
```

Output:

```
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 1
Enter the element to insert a new node: 100
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
```

```
Enter your choice: 1
Enter the element to insert a new node: 200
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 1
Enter the element to insert a new node: 300
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 1
Enter the element to insert a new node: 400
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
```

```
4.Display
5.Exit
Enter your choice: 2
Enter the element to search: 300
Key 300 found
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 4
Inorder Traversal of BST: 100 200 300 400 500 Menu:
```

```
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 3
Enter node value which you want to delete: 200
Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 4
Inorder Traversal of BST: 100 300 400 500 Menu:
BST Operations:
1.Insert
2.Search
3.Delete
4.Display
5.Exit
Enter your choice: 5
```

Inferences:

- In binary search tree, the **insertion operation** ensures that every new element is placed in its correct position according to the BST property, maintaining the order automatically.
- The **search operation** efficiently locates an element by recursively or iteratively comparing it with the root and traversing either the left or right subtree — this reduces the average search time to **O(log n)** for balanced trees.
- The **display (traversal)** operation uses **recursive functions** such as in-order, pre-order, and post-order to visit and print all nodes in different sequences.
- **In-order traversal** displays the elements in **ascending (sorted)** order, verifying that the structure is a valid BST.

WEEK-13

Date:22-10-2025

List of programs:

1. Write a C progam to implement Breadth First search Traversal.
2. Write a C progam to implement Depth First search Traversal.

1. Aim: To Write a C progam to implement Breadth First search Traversal.

Program:

```
#include <stdio.h>
int bfs[20],rear = -1,front = -1,vt[20];
void store(int n)
{
   bfs[++rear] = n;
}
int delet()
{
   return bfs[++front];
}
void bfsearch(int a[][10], int n, int id)
{
   int i;
   for (i=0;i<n;i++)
      vt[i] = 0;
   id=id-1;
   store(id);
   vt[id] = 1;
   printf("\nBFS visited vertices: ");
   while (front!=rear)
   {
```

```
        id = delet();
        printf("%d ", id + 1);
        for (i = 0; i < n; i++)
        {
            if (a[id][i] == 1 && vt[i] == 0)
            {
                store(i);
                vt[i] = 1;
            }
        }
    }
}
int main()
{
    int a[10][10], n, i, j, id;
    printf("Enter number of vertices on graph:");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    printf("Enter starting vertex:");
    scanf("%d", &id);
    bfsearch(a, n, id);
    return 0;
}
```

Output:

```
Enter number of vertices on graph:5
Enter adjacency matrix:
0 1 1 0 0
1 0 0 1 1
1 0 0 1 0
0 1 1 0 1
0 1 0 1 0
Enter starting vertex:1

BFS visited vertices: 1 2 3 4 5

=== Code Execution Successful ===
```

2. Aim: To Write a C progam to implement Depth First Search Graph Traversal.

Program:

```
#include <stdio.h>
int vt[10], dfs[10], top = -1;
void push(int n)
 {
   dfs[++top] = n;
}
void pop()
{
   top--;
}
void dfs_Traversal(int a[][10], int n, int start)
{
   int i, found;
   for (i = 0; i < n; i++)
      vt[i] = 0;
   start = start - 1;
   push(start);
   vt[start] = 1;
   printf("DFS visited nodes are: %d", start + 1);
   while (top != -1)
{
      start = dfs[top];
      found = 0;
      for (i = 0; i < n; i++) {
         if (a[start][i] == 1 && vt[i] == 0) {
            push(i);
```

```
                printf(" -> %d", i + 1);
                vt[i] = 1;
                found = 1;
                break;
            }
        }
        if (found == 0)
            pop();
    }
}
int main()
{
    int a[10][10], n, i, j, id;
    printf("Enter number of vertices in graph: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &a[i][j]);
    printf("Enter starting vertex: ");
    scanf("%d", &id);
    dfs_Traversal(a, n, id);
    return 0;
}
```

Output:

```
Enter number of vertices in graph: 4
Enter adjacency matrix:
1 0 0 1
0 1 1 0
0 1 1 0
1 0 0 1
Enter starting vertex: 1
DFS visited nodes are: 1 -> 4

=== Code Execution Successful ===
```

Inferences:

- **BFS** explores nodes level by level, visiting all neighbors before moving deeper.
- It guarantees the shortest path in unweighted graphs and helps identify levels or layers.
- **DFS** explores as far as possible along each branch before backtracking.
- It’s useful for pathfinding, cycle detection, and topological sorting.
- BFS uses more memory, while DFS is more memory-efficient but may not find the shortest path.