Name - Akshat Shah

Roll No - 21BCP322

Sem – III

Data Structures Lab

Course Code – 20CP201P

Department of Computer Science Engineering
School of Technology,
Pandit Deendayal Energy University

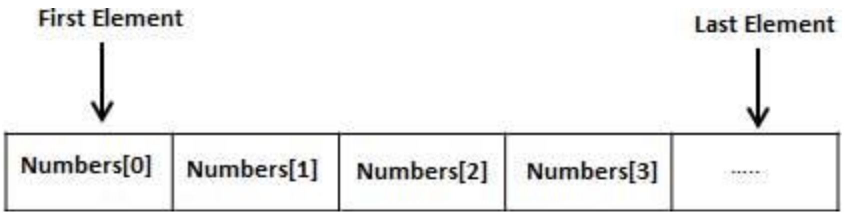
Contents

Practical number 1:[Revision of **Arrays**]

About Arrays:

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type.

General Structure:



Real-time Application:

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables.
- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues, stacks and strings.

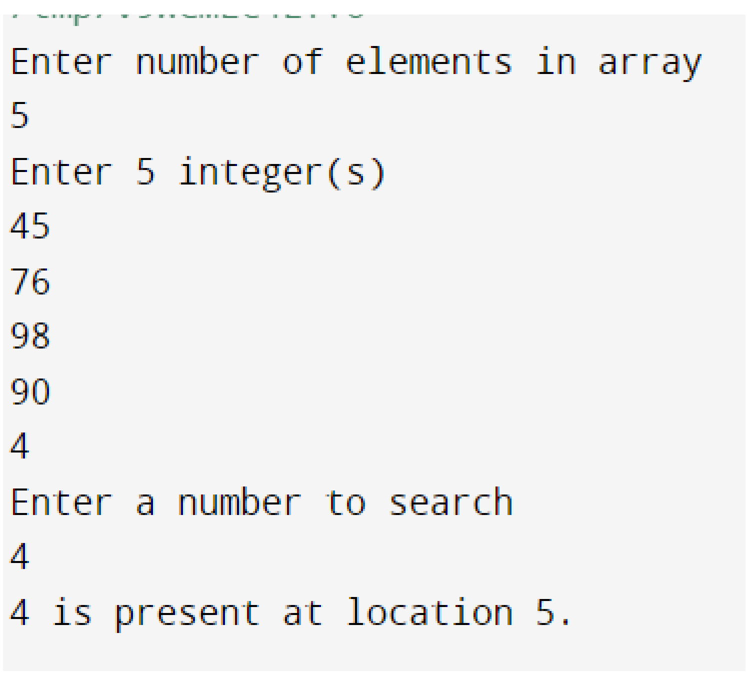
1. C program for performing Linear search.

Code:

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;

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

  printf("Enter %d integer(s)\n", n);

  for (c = 0; c < n; c++)
```

```
    scanf("%d", &array[c]);

  printf("Enter a number to search\n");
  scanf("%d", &search);

  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);
return 0;
}
```

Output:



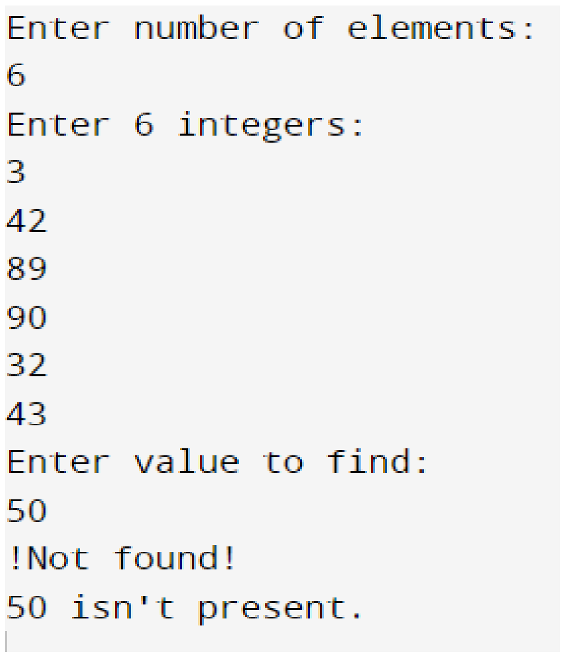
```
Enter number of elements in array
5
Enter 5 integer(s)
45
76
98
90
4
Enter a number to search
4
4 is present at location 5.
```

2. C program for performing Binary Search

Code:

```
#include <stdio.h>
int main()
{
  int c, first, last, middle, n, search, array[100];

  printf("Enter number of elements:\n");
  scanf("%d", &n);

  printf("Enter %d integers:\n", n);

  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);

  printf("Enter value to find:\n");
  scanf("%d", &search);

  first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;
    middle = (first + last)/2;
  }
  if (first > last)
    printf("!Not found!%d isn't present.\n", search);


  return 0;
}
```

Output:



```
Enter number of elements:
6
Enter 6 integers:
3
42
89
90
32
43
Enter value to find:
50
!Not found!
50 isn't present.
```

3. Write a program in C to perform bubble sort, insertion sort and selection sort. Take the array size and array elements from user.

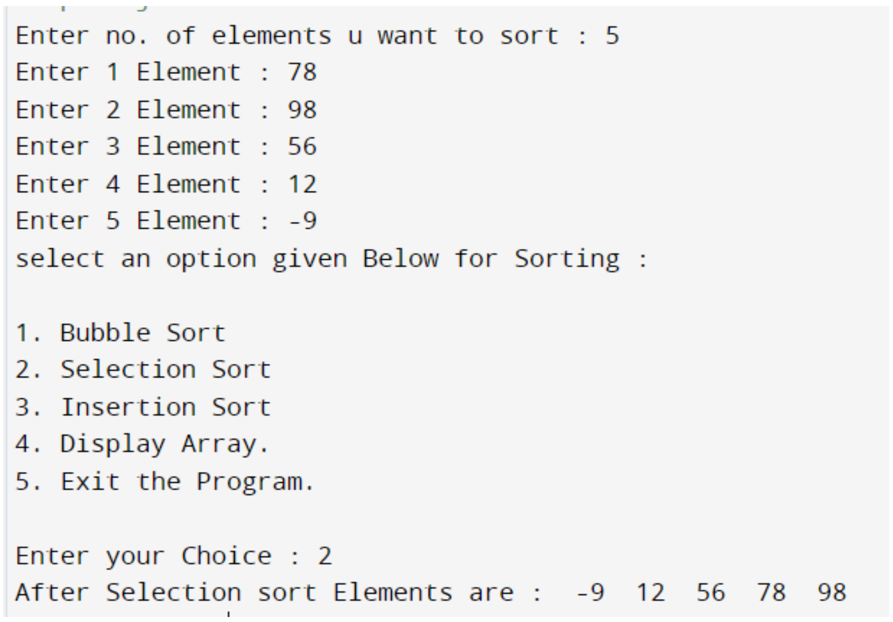
Code:

```
#include<stdio.h>

#include<stdlib.h>


void display(int a[],int n);

void bubble_sort(int a[],int n);

void selection_sort(int a[],int n);

void insertion_sort(int a[],int n);


int main()


{

   int n,choice,i;

   char ch[20];

   printf("Enter no. of elements u want to sort : ");

   scanf("%d",&n);

   int arr[n];


for(i=0;i<n;i++)

   {

        printf("Enter %d Element : ",i+1);

        scanf("%d",&arr[i]);


   }

   printf(" select an option given Below for Sorting : \n");


while(1)


   {
```

```
printf("\n1. Bubble Sort\n2. Selection Sort\n3. Insertion Sort\n4. Display Array.\n5. Exit the Program.\n");


printf("\nEnter your Choice : ");


scanf("%d",&choice);
switch(choice)
{


case 1:


    bubble_sort(arr,n);


    break;


case 2:
    selection_sort(arr,n);
    break;


case 3:
    insertion_sort(arr,n);
    break;


case 4:
    display(arr,n);
    break;
case 5:


    return 0;


default:
    printf("\nPlease Select only 1-5 option ----\n");
    }


}


return 0;
}
 void display(int arr[],int n)
{


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

//Bubble Sort Function
void bubble_sort(int arr[],int n)
{
  int i,j,temp;
  for(i=0;i<n;i++)

  {


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


      {


          if(arr[j]>arr[j+1])


          {
             temp=arr[j];
```

```
            arr[j]=arr[j+1];

            arr[j+1]=temp;

          }

      }

  }

printf("After Bubble sort Elements are : ");

display(arr,n);

}




//Selection Sort Function
void selection_sort(int arr[],int n)

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


        }
    }

printf("After Selection sort Elements are : ");

display(arr,n);
}
//Insertion Sort Function
void insertion_sort(int arr[],int n)
{
    int i,j,min;

    for(i=1;i<n;i++)
    {
        min=arr[i];
        j=i-1;
        while(min<arr[j] && j>=0)
        {
            arr[j+1]=arr[j];
            j=j-1;
        }
        arr[j+1]=min;
}
printf("After Insertion sort Elements are : ");
display(arr,n);
}
```

Output:



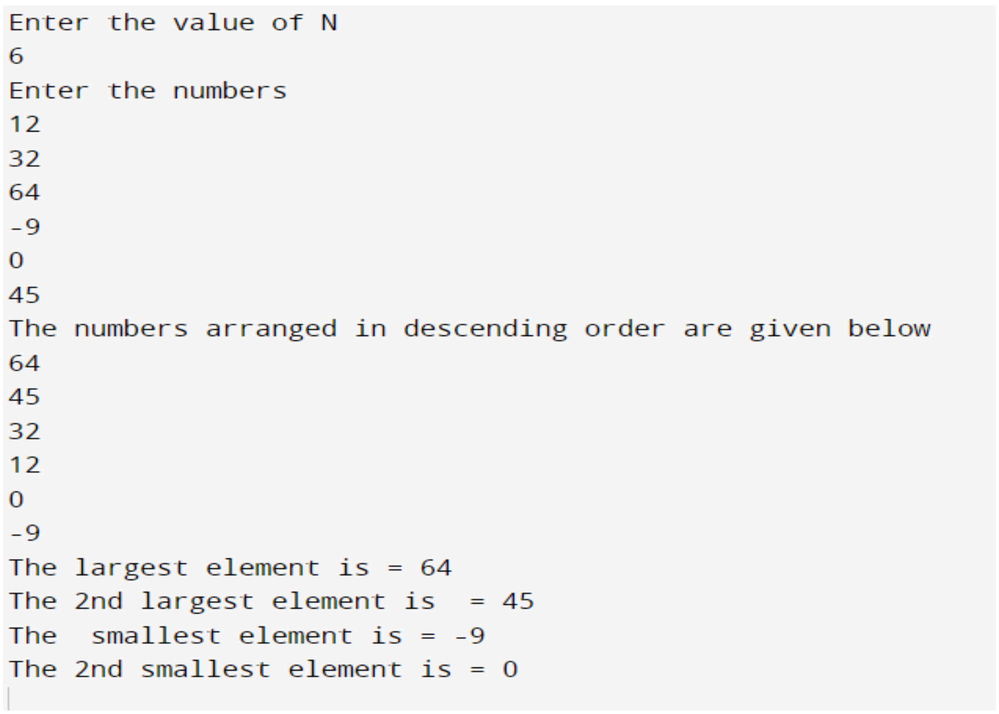
```
Enter no. of elements u want to sort : 5
Enter 1 Element : 78
Enter 2 Element : 98
Enter 3 Element : 56
Enter 4 Element : 12
Enter 5 Element : -9
select an option given Below for Sorting :

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Display Array.
5. Exit the Program.

Enter your Choice : 2
After Selection sort Elements are :  -9  12  56  78  98
```

3. Write a program in C that obtains the minimum and maximum element from the array. Modify this program to give the second largest and second smallest element of the array.

Code:

```
#include <stdio.h>
   void main ()
   {
      int number[30];
      int i, j, a, n, counter, average;

      printf("Enter the value of N\n");
      scanf("%d", &n);
      printf("Enter the numbers \n");
      for (i = 0; i < n; ++i)
         scanf("%d", &number[i]);

      for (i = 0; i < n; ++i)
      {
         for (j = i + 1; j < n; ++j)
         {
            if (number[i] < number[j])
            {
                a = number[i];
                number[i] = number[j];
                number[j] = a;
            }
         }
      }

      printf("The numbers arranged in descending order are given below \n");
      for (i = 0; i < n; ++i)
      {
         printf("%d\n", number[i]);
      }
      printf("The largest element is = %d\n",number[0]);
      printf("The 2nd largest element is  = %d\n", number[1]);
      printf("The  smallest element is = %d\n", number[n - 1]);
       printf("The 2nd smallest element is = %d\n", number[n - 2]);
  }
```

Output:



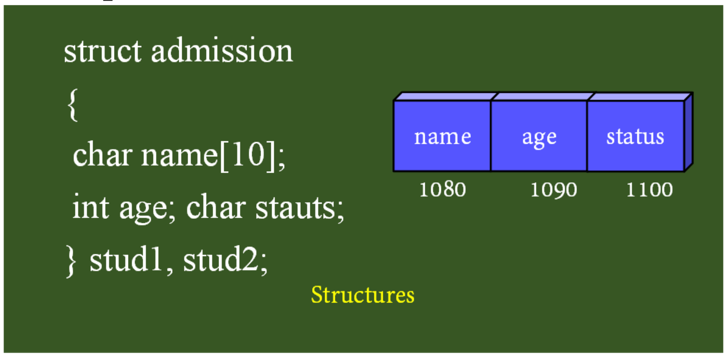
```
Enter the value of N
6
Enter the numbers
12
32
64
-9
0
45
The numbers arranged in descending order are given below
64
45
32
12
0
-9
The largest element is = 64
The 2nd largest element is  = 45
The  smallest element is = -9
The 2nd smallest element is = 0
```

Practical Number 2:[Implementing Structures]

About Structures:

- Structure is a user-defined data type which allows to combine different data types to store a particular record.

  Example:



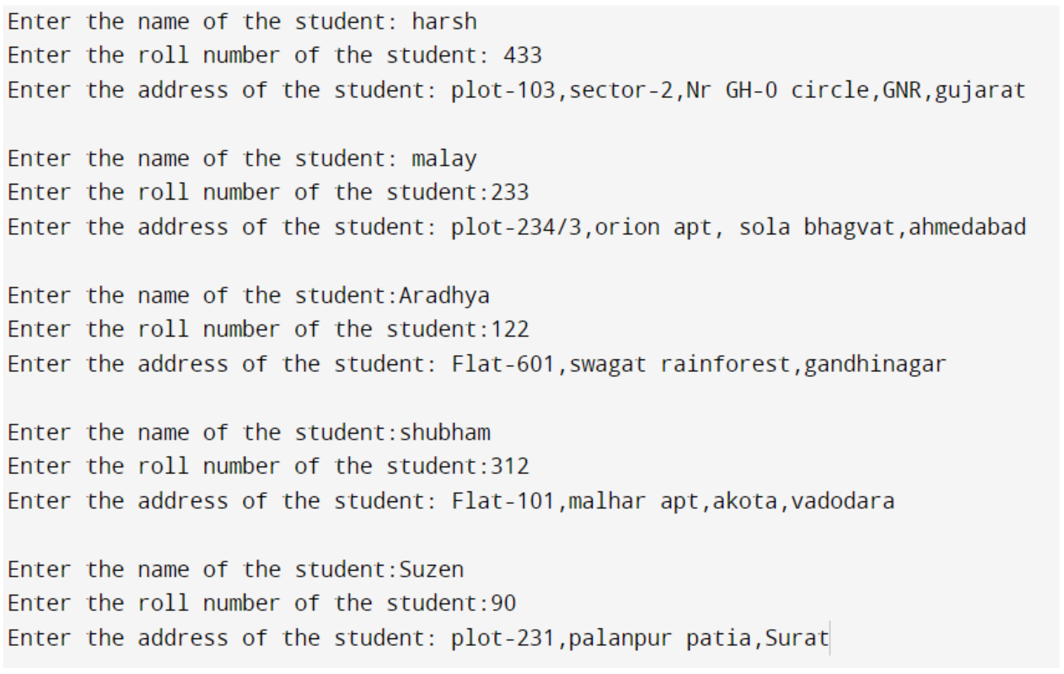
```
struct admission
{
 char name[10];
 int age; char stauts;
} stud1, stud2;
```

1. Create a structure Student in C with student name, student roll number and student address as its data members. Create the variable of type student and print the values.

Code:

```
#include <stdio.h>


struct Student{
    char name[15];
    int roll;
    char address[100];
};
int main(){
    struct Student s2[5];
    for(int i = 0; i < 5; i++){
        printf("Enter the name of the student: ");
        scanf("%s", s2[i].name);
        printf("Enter the roll number of the student: ");
        scanf("%d", &s2[i].roll);
        printf("Enter the address of the student: ");
        scanf("%s", s2[i].address);
}
for(int i = 0; i < 5; i++){
        printf("The name of the student is: %s\n", s2[i].name);
        printf("The roll number of the student is: %d\n", s2[i].roll);
        printf("The address of the student is: %s\n", s2[i].address);
    }
}
```

Output:



```
Enter the name of the student: harsh
Enter the roll number of the student: 433
Enter the address of the student: plot-103,sector-2,Nr GH-0 circle,GNR,gujarat

Enter the name of the student: malay
Enter the roll number of the student:233
Enter the address of the student: plot-234/3,orion apt, sola bhagvat,ahmedabad

Enter the name of the student:Aradhya
Enter the roll number of the student:122
Enter the address of the student: Flat-601,swagat rainforest,gandhinagar

Enter the name of the student:shubham
Enter the roll number of the student:312
Enter the address of the student: Flat-101,malhar apt,akota,vadodara

Enter the name of the student:Suzen
Enter the roll number of the student:90
Enter the address of the student: plot-231,palanpur patia,Surat
```

2. Create a structure Organization with organization name and organization ID as its data members. Next, create another structure Employee that is nested in structure Organization with employee ID, employee salary and employee name as its data members. Write a program in such a way that there are two organizations and each of these contains two employees.

Code:

```
#include <stdio.h>
struct Organization{
    char name[20];
    char id[10];

    struct Employee{
        char id[20];
        int salary;
        char name[20];
    }Employee[2];
};

int main(){
    struct Organization o1;
    printf("Enter the name of the organization: ");
    scanf("%s", o1.name);
    printf("Enter the ID of the organization: ");
    scanf("%d", &o1.id);

    for(int i = 0; i < 2; i++){
        printf("Enter the name of the employee: ");
        scanf("%s", o1.Employee[i].name);
        printf("Enter the ID of the employee: ");
        scanf("%d", &o1.Employee[i].id);
        printf("Enter the salary of the employee: ");
        scanf("%d", &o1.Employee[i].salary);
    }

    struct Organization o2;
    printf("Enter the name of the organization: ");
    scanf("%s", o2.name);
    printf("Enter the ID of the organization: ");
    scanf("%d", &o2.id);

    for(int i = 0; i < 2; i++){
        printf("Enter the name of the employee: ");
        scanf("%s", o2.Employee[i].name);
        printf("Enter the ID of the employee: ");
        scanf("%d", &o2.Employee[i].id);
        printf("Enter the salary of the employee(Rs): ");
        scanf("%d", &o2.Employee[i].salary);
    }

    printf("The name of the organization is: %s\n", o1.name);
    printf("The ID of the organization is: %d\n", o1.id);
    for(int i = 0; i < 2; i++){
        printf("The name of the employee is: %s\n", o1.Employee[i].name);
        printf("The ID of the employee is: %d\n", o1.Employee[i].id);
        printf("The salary of the employee is: %d\n", o1.Employee[i].salary);
    }

    printf("The name of the organization is: %s\n", o2.name);
    printf("The ID of the organization is: %d\n", o2.id);
    for(int i = 0; i < 2; i++){
        printf("The name of the employee is: %s\n", o2.Employee[i].name);
        printf("The ID of the employee is: %d\n", o2.Employee[i].id);
        printf("The salary of the employee is: %d\n", o2.Employee[i].salary);
    }
```

}

Output:

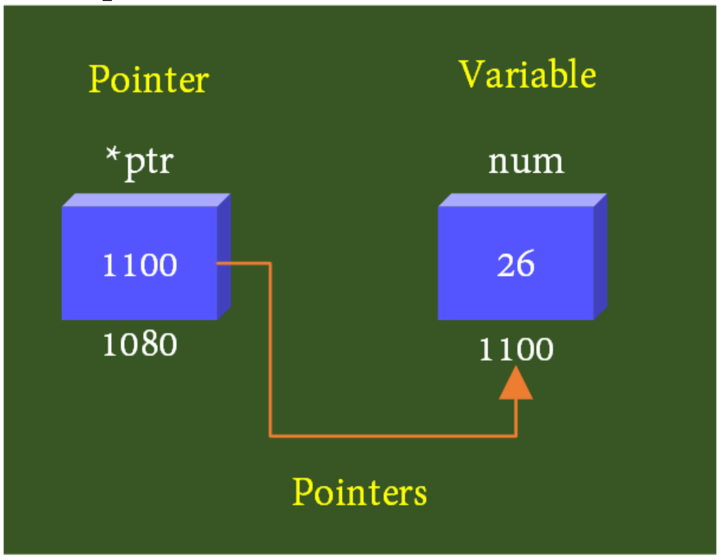
```
Enter the name of the organization: Aramica pvt.ltd
Enter the ID of the organization: UD1091
Enter the name of the employee:Sandhya
Enter the ID of the employee:PIV012
Enter the salary of the employee(Rs):40000

Enter the name of the employee:Sonali
Enter the ID of the employee:PIV010
Enter the salary of the employee(Rs):42000

Enter the name of the organization: Sunday org
Enter the ID of the organization: UD1092

Enter the name of the employee:Aaron
Enter the ID of the employee: 10293
Enter the salary of the employee(Rs): 50000

Enter the name of the employee:Aryan
Enter the ID of the employee: 32109
Enter the salary of the employee(Rs): 30000
```

Practical Number 3:[Revision of pointers]

About pointers:

- The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

  Example:



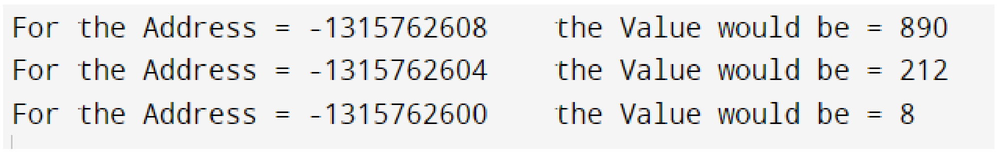
1. Write a program in C to implement arrays of pointers and pointers to arrays.

Code:

// Arrays of pointers

```
#include<stdio.h>

#define SIZE 20

int main()

{

int *arr[3];

int p = 890, q = 212, r = 8, i;

arr[0] = &p;

arr[1] = &q;

arr[2] = &r;
for(i = 0; i < 3; i++)

{

printf("For the Address = %d\t the Value would be = %d\n", arr[i], *arr[i]);

}

return 0;

}
```

Output:

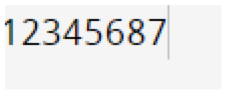


```
For the Address = -1315762608    the Value would be = 890
For the Address = -1315762604    the Value would be = 212
For the Address = -1315762600    the Value would be = 8
```

Code:

```
// Pointers to arrays

#include<stdio.h>


void main()

{

  int a[8] = {1,2,3,4,5,6,8,7};

  int *p = a;

  for (int i = 0; i < 8; i++)

  {

    printf("%d", *p);

    p++;

  }

 return 0;

}
```

Output:



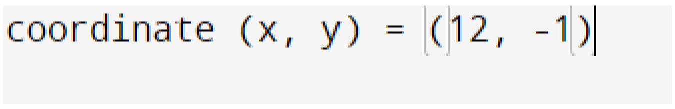
```
12345687
```

2. Write a program in C to implement pointers to structures.

Code:

```
#include<stdio.h>


// create a structure Coordinate
struct Coordinate {
   // declare structure members
   int x,y;
};


int main() {
   struct Coordinate first_point;
   struct Coordinate *cp;
   cp = &first_point;


   (*cp).x = 12;
   (*cp).y = -1;


   printf("First coordinate (x, y) = (%d, %d)", (*cp).x, (*cp).y);
   return 0;
}
```

Output:



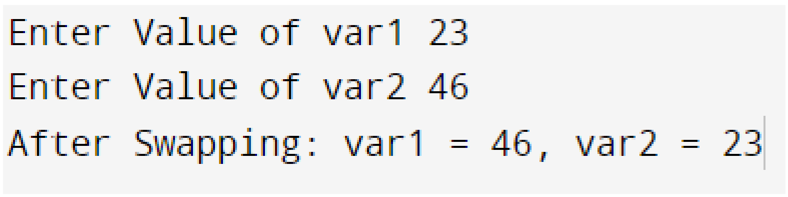
```
coordinate (x, y) = (12, -1)
```

3. Write a program in C to perform swapping of two numbers by passing addresses of the variables to the functions

Code:

```
void swap(int *num1, int *num2)
{
int temp = *num1;
*num1 = *num2;
*num2 = temp;
}
int main()
{
int var1, var2;
printf("Enter Value of var1 ");
scanf("%d", &var1);
printf("Enter Value of var2 ");
scanf("%d", &var2);
swap(&var1, &var2);
printf("After Swapping: var1 = %d, var2 = %d", var1, var2);
return 0;
}
```

Output:



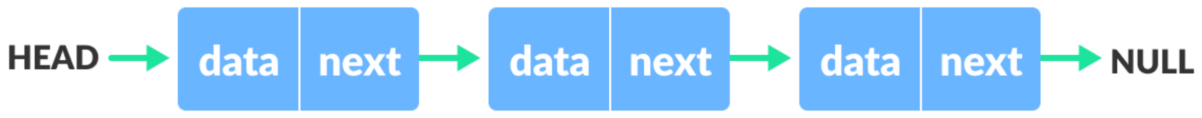
```
Enter Value of var1 23
Enter Value of var2 46
After Swapping: var1 = 46, var2 = 23
```

Practical Number 4:[Implementing **Linked lists**]

About Linked lists:

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the **data** and the **address** of the next node.

General Structure:



Real-time Applications:

- Maintaining a directory of file names
- Cache in a browser which allows us to hit a back button is due a linked list of urls.
- Representing Sparse matrices.

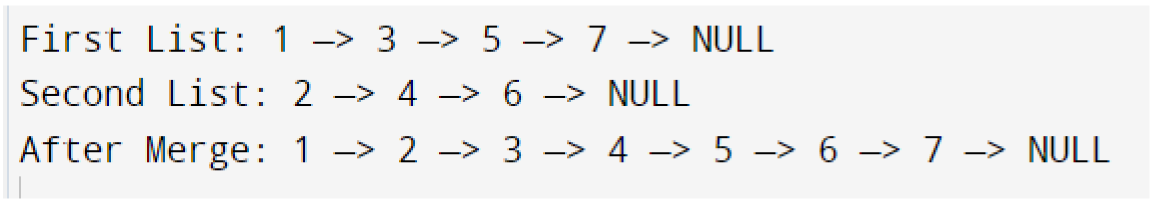
1. C program that takes two sorted lists as inputs and merge them into one sorted list.

Code:

```
#include <stdio.h>

#include <stdlib.h>


// A Linked List Node

struct Node

{

    int data;

    struct Node* next;

};
```

```
// print list
void printList(struct Node* head)
{
    struct Node* ptr = head;
    while (ptr)
    {
        printf("%d —> ", ptr->data);
        ptr = ptr->next;
    }

    printf("NULL\n");
}
//  insert a new node at the beginning of the linked list
void push(struct Node** head, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
// moving node  to the front of the destination
void moveNode(struct Node** destRef, struct Node** sourceRef)
{
    // if the source list empty, do nothing
    if (*sourceRef == NULL) {
        return;
    }
    struct Node* newNode = *sourceRef;  // the front source node
    *sourceRef = (*sourceRef)->next;    // advance the source pointer
    newNode->next = *destRef;           // link the old dest off the new node
    *destRef = newNode;                 // move dest to point to the new node
}

// Takes two lists sorted in increasing order and merge their nodes
// to make one big sorted list, which is returned
struct Node* sortedMerge(struct Node* a, struct Node* b)
{
    // a temp  first node to hang the result on
    struct Node temp;
    temp.next = NULL;

    // points to the last result node — so `tail->next` is the place
    // to add new nodes to the result.
    struct Node* tail = &temp;

    while (1)
    {
        // if either list runs out, use the other list
        if (a == NULL)
        {
            tail->next = b;
            break;
        }
        else if (b == NULL)
        {
            tail->next = a;
            break;
        }
        if (a->data <= b->data) {
            moveNode(&(tail->next), &a);
        }
        else {
            moveNode(&(tail->next), &b);
        }

        tail = tail->next;
    }

    return temp.next;
}
int main(void)
{
    // input key
    int key[] = { 1, 2, 3, 4, 5, 6, 7 };
    int n = sizeof(key)/sizeof(key[0]);

    struct Node *a = NULL, *b = NULL;
    for (int i = n - 1; i >= 0; i = i - 2) {
        push(&a, key[i]);
    }
    for (int i = n - 2; i >= 0; i = i - 2) {
        push(&b, key[i]);
    }

    // print both lists
    printf("First List: ");
    printList(a);

    printf("Second List: ");
    printList(b);

    struct Node* head = sortedMerge(a, b);
    printf("After Merge: ");
    printList(head);

    return 0;
}
```

Output:



```
First List: 1 —> 3 —> 5 —> 7 —> NULL
Second List: 2 —> 4 —> 6 —> NULL
After Merge: 1 —> 2 —> 3 —> 4 —> 5 —> 6 —> 7 —> NULL
```

2. Write a program to insert a new node into the linked list. A node can be added into the linked list using three ways: [Write code for all the three ways.]

a. At the front of the list

b. After a given node

c. At the end of the list

Code:

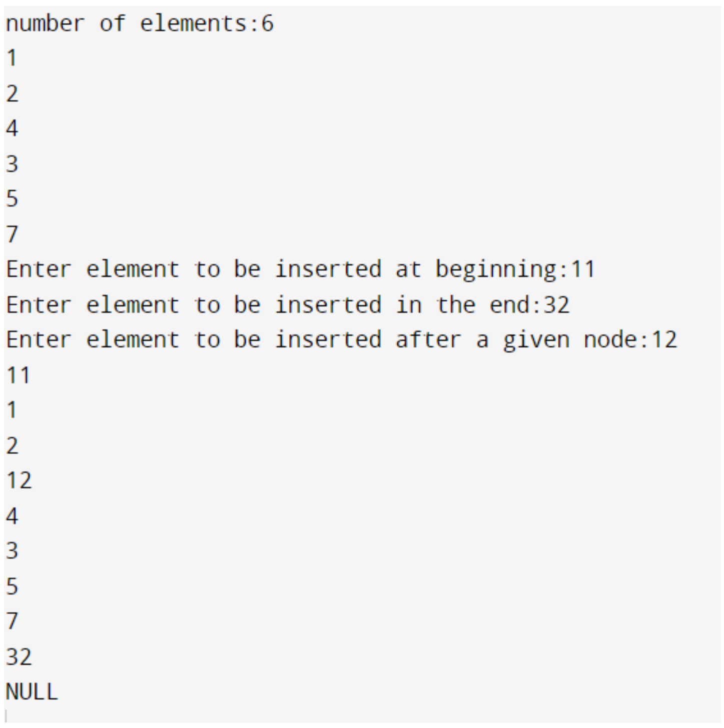
```
#include <stdio.h>
#include <stdlib.h>

struct node
```

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

display(struct node *head)
{
   if(head == NULL)
   {
      printf("NULL\n");
   }
   else
   {
      printf("%d\n", head -> data);
      display(head->next);
   }
}

struct node* front(struct node *head,int value)
{
   struct node *p;
   p=malloc(sizeof(struct node));
   p->data=value;
   p->next=head;
   return (p);
}
```

```
end(struct node *head,int value)
{
    struct node *p,*q;
    p=malloc(sizeof(struct node));
    p->data=value;
    p->next=NULL;
    q=head;
    while(q->next!=NULL)
    {
        q = q->next;
    }
    q->next = p;
}

after(struct node *a, int value)
{
    if (a->next != NULL)
    {
        struct node *p;
        p = malloc(sizeof(struct node));
        p->data = value;
        p->next = a->next;
        a->next = p;
    }
    else
    {
        printf("Use end function to insert at the end\n");
    }
}

int main()
{
    struct node *prev,*head, *p;
    int n,i,f,m,l;
    printf ("number of elements:");
    scanf("%d",&n);
    head=NULL;
    for(i=0;i<n;i++)
    {
        p=malloc(sizeof(struct node));
        scanf("%d",&p->data);
        p->next=NULL;
        if(head==NULL)
            head=p;
        else
            prev->next=p;
        prev=p;
    }
    printf("Enter element to be inserted at beginning:");
    scanf("%d",&f);
    head = front(head,f);
```

```
     printf("Enter element to be inserted in the end:");
    scanf("%d",&l);
    end(head,l);
     printf("Enter element to be inserted after a given node:");
     scanf("%d",&m);
    after(head->next->next,m);
    display(head);
    return 0;
}
```

Output:



```
number of elements:6
1
2
4
3
5
7
Enter element to be inserted at beginning:11
Enter element to be inserted in the end:32
Enter element to be inserted after a given node:12
11
1
2
12
4
3
5
7
32
NULL
```

3. Write a program to delete a node from the linked list. A node can be deleted from the linked list using three ways: [Write code for all the three ways.]

a. Delete from the beginning

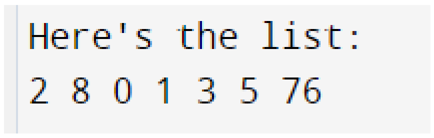
b. Delete from the end

c. Delete from the middle.

Code:

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

struct node{
    int data;
    struct node *next;
};
struct node *head = NULL;
void insert(int data){
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = NULL;
    if(head == NULL){
        head = temp;
    }
    else{
        struct node *temp1 = head;
        while(temp1->next != NULL){
            temp1 = temp1->next;
        }
        temp1->next = temp;
    }
}

void delete_from_beginning(){
    struct node *temp = head;
    head = head->next;
    free(temp);
}

void delete_from_end(){
    struct node *temp = head;
    while(temp->next->next != NULL){
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

void delete_from_middle(int data){
    struct node *temp = head;
    while(temp->next->data != data){
        temp = temp->next;
    }
    struct node *temp1 = temp->next;
    temp->next = temp->next->next;
    free(temp1);
}

void print(){
    struct node *temp = head;
    while(temp != NULL){
        printf("%d ",temp->data);
        temp = temp->next;
    }

}
int main(){
    insert(1);
    insert(2);
    insert(5);
    insert(8);
    insert(0);
    insert(1);
    insert(3);
    insert(5);
    insert(76);
    insert(9);
    delete_from_beginning();
    delete_from_end();
    delete_from_middle(5);
    print();
    return 0;
}
```

Output:



```
Here's the list:
2 8 0 1 3 5 76
```

4. C program to implement all operations on Circular Linked list

   Code:

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

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

struct node *head;

// function prototyping
struct node *create(int);
void insert_at_begin(int);
void insert_at_end(int);
void insert_at_position(int, int);
void delete_at_begin();
void delete_at_end();
void delete_at_position(int);
void search(int);
void update(int, int);
void print_list();
int size_of_list();
int getData();
int getPosition();

int main()
{
   char user_active = 'Y';
   int user_choice;
   int data, position;

   while(user_active == 'Y' || user_active == 'y')
   {
```

```
printf("\n\n* Operations on Circular  Linked List*\n");
printf("\n1. Insert a node at beginning");
printf("\n2. Insert a node at end");
printf("\n3. Insert a node at given position");
printf("\n\n4. Delete a node from beginning");
printf("\n5. Delete a node from end");
printf("\n6. Delete a node from given position");
printf("\n\n7. Traverse the list");
printf("\n8. Search a node data");
printf("\n9. Update a node data");
printf("\n10. Exit");

printf("\nEnter your choice: ");
scanf("%d", &user_choice);


switch(user_choice)
{
   case 1:
      printf("\nInserting a node at beginning");
      data = getData();
      insert_at_begin(data);
      break;

   case 2:
      printf("\nInserting a node at end");
      data = getData();
      insert_at_end(data);
      break;

   case 3:
      printf("\nInserting a node at the given position");
      data = getData();
      position = getPosition();
      insert_at_position(data, position);
      break;
```

```
case 4:
    printf("\nDeleting a node from beginning\n");
    delete_at_begin();
    break;

case 5:
    printf("\nDeleting a node from end\n");
    delete_at_end();
    break;

case 6:
    printf("\nDelete a node from given position\n");
    position = getPosition();
    delete_at_position(position);
    break;

case 7:
    printf("\nTraverse the list\n\n");
    print_list();
    break;

case 8:
    printf("\nSearching the node data");
    data = getData();
    search(data);
    break;

case 9:
    printf("\nUpdating the node data");
    data = getData();
    position = getPosition();
    update(position, data);
    break;

case 10:
    printf("\nProgram was terminated\n\n");
    return 0;
```

```
            default:
                printf("\n\t Invalid Choice\n");
        }


        printf("\nDo you want to continue? (Y/N) : ");
        fflush(stdin);
        scanf(" %c", &user_active);
    }

    return 0;
}

struct node *create(int data)
{
    struct node *new_node = (struct node *)malloc(sizeof(struct node));

    if (new_node == NULL)
    {
        printf("\nMemory can't be allocated.\n");
        return NULL;
    }

    new_node->data = data;
    new_node->next = NULL;

    return new_node;
}

// function to insert a new node at the beginning of the list
void insert_at_begin(int data)
{
    struct node *new_node = create(data);

    if (new_node != NULL)
    {
        struct node *last = head;

        // if the list is empty
        if (head == NULL)
        {
            head = new_node;
            new_node->next = head;
            return;
        }

        // traverse to the end node
        while (last->next != head)
        {
            last = last->next;
        }

        // update  the last node to the new node
        last->next = new_node;

        // update the next pointer of the new node to the head node
        new_node->next = head;

        // update the head of the list to new node
        head = new_node;
    }
}

// function to insert a new node at the end of the list
void insert_at_end(int data)
{
    struct node *new_node = create(data);

    if (new_node != NULL)
    {

        // if the list is empty
        if (head == NULL)
        {
            head = new_node;
            new_node->next = head;
            return;
        }

        struct node *last = head;

        // traverse to the end node
        while (last->next != head)
        {
            last = last->next;
        }

        // update  the last node to the new node
        last->next = new_node;

        // update the next pointer of the new node to the head node
        new_node->next = head;
    }
}

// function to insert a new node at the given position
void insert_at_position(int position, int data)
{
    // checking if the position is valid or not
    if (position <= 0)
    {
        printf("\nInvalid Position\n");
    }
    else if (head == NULL && position > 1)
    {
        printf("\nInvalid Position\n");
    }
    else if (head != NULL && position > size_of_list())
    {
        printf("\nInvalid Position\n");
    }
    else if (position == 1)
    {
        insert_at_begin(data);
    }
    else
    {
        struct node *new_node = create(data);

        if (new_node != NULL)
        {
            struct node *temp = head, *prev = NULL;
            // Since, temp is already pointing to first node
            int i = 1;

            while (++i <= position)
            {
                prev = temp;
                temp = temp->next;
            }

            // update the prev node to the new noe
            prev->next = new_node;

            // update the new node to the temp (position node)
            new_node->next = temp;
        }
    }
}

// function to delete a node from the beginning of the list
void delete_at_begin()
{
    // check where the list is empty or not
    if (head == NULL)
    {
        printf("\n List is Empty! \n");
        return;
    }

    struct node *last = head;
    struct node *temp = head;


    if (last->next == head)
    {
        free(last);
        head = NULL;
        return;
    }

    // traverse to the last node
    while (last->next != head)
    {
        last = last->next;
    }

    head = head->next;
    last->next = head;

    free(temp);
    temp = NULL;
}

// function to delete a node from the end of the list
void delete_at_end()
{
    // check where the list is empty or not
    if (head == NULL)
    {
        printf("\n List is Empty! \n");
        return;
    }
    // traverse to the end of the list
    struct node *prev = head;
    struct node *temp = head->next;

    // if only one node in the list
    if (prev->next == head)
    {
        free(prev);
        head = NULL;
        return;
    }

    while (temp->next != head)
    {
        prev = temp;
        temp = temp->next;
    }

    prev->next = head;

    free(temp);
    temp = NULL;
}

// function to delete a node from the given position
void delete_at_position(int position)
{
    if (position <= 0)
    {
        printf("\n Invalid Position \n");
    }
    else if (position > size_of_list())
    {
        printf("\n Invalid position \n");
    }
    else if (position == 1)
    {
        delete_at_begin();
    }
    else if (position == size_of_list())
    {
        delete_at_end();
    }
    else
    {
        struct node *temp = head;
        struct node *prev = NULL;
        int i = 1;

        while (i < position)
        {
            prev = temp;
            temp = temp->next;
            i += 1;
        }

        prev->next = temp->next;
        free(temp);
        temp = NULL;
    }
}

// print the node values
void print_list()
{
    struct node *temp = head;

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

    printf("\n");
    do
    {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);

    printf("\n");
}

// print the node values recursively
void print_list_reverse(struct node* temp)
{
    if (temp->next == head)
    {
        printf("%d ", temp->data);
        return;
    }
    print_list_reverse(temp->next);
    printf("%d ", temp->data);
}

// search a data into the list
void search(int key)
{
    struct node* temp = head;

    do
    {
        if (temp->data == key)
        {
            printf("\n\t Data Found\n");
            return;
        }
        temp = temp->next;
    }while (temp->next != head);
    printf("\n\tData not Found\n");
}
```

```
// function to update a node
void update(int position, int new_data)
{
    if (position <= 0 || position > size_of_list())
    {
        printf("\n Invalid position\n");
        return;
    }

    struct node* temp = head;
    int i = 0;

    while (i <= position)
    {
        temp = temp->next;
        i += 1;
    }

    temp->data = new_data;
}

// function to calculate the size of the list
int size_of_list()
{
    if (head == NULL)
    {
        return 0;
    }

    struct node *temp = head;
    int count = 1;

    while (temp->next != head)
    {
        count += 1;
        temp = temp->next;
    }
    return count;
}

int getData()
{
    int data;
    printf("\n\nEnter Data: ");
    scanf("%d", &data);

    return data;
}

int getPosition()
{
    int pos;
    printf("\n\nEnter Position: ");
    scanf("%d", &pos);

    return pos;
}
```

Output:

```
* Operations on Circular  Linked List*

1. Insert a node at beginning
2. Insert a node at end
3. Insert a node at given position

4. Delete a node from beginning
5. Delete a node from end
6. Delete a node from given position

7. Traverse the list
8. Search a node data
9. Update a node data
10. Exit
Enter your choice: 1
Inserting a node at beginning
Enter Data: 12
Do you want to continue? (Y/N) : Y
```

5. C program to implement all operations on Doubly Linked list

   Code:

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

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

struct node* head = NULL;

/* functions prototyping */
void insert_at_beginning(int);
void insert_at_end(int);
void insert_at_position(int, int);
void delete_from_beginning();
void delete_from_pos(int);
void delete_from_end();
void print_from_beginning();
void print_from_end(struct node*);
void search_data(int);
void update_node_data(int, int);
void list_sort();

/* helper functions */
struct node* create_node(int);
int size_of_list();
int getData();
int getPosition();
void empty_list_message();
void memory_error_message();
void invalid_pos_message();

int main()
{
   char user_active = 'Y';
   int user_choice;
   int data, position;

   while (user_active == 'Y' || user_active == 'y')
   {

      printf("\n\n* Doubly Linked List*\n");
      printf("\n1. Insert a node at the beginning");
      printf("\n2. Insert a node at the end");
      printf("\n3. Insert a node at the given position");
      printf("\n\n4. Delete a node from the beginning");
      printf("\n5. Delete a node from the end");
      printf("\n6. Delete a node from the given position");
      printf("\n\n7. Print list from the beginning");
      printf("\n8. Print list from the end");
      printf("\n9. Search a node data");
      printf("\n10. Update a node data");
      printf("\n11. Sort the list");
      printf("\n12. Exit");


      printf("\nEnter your choice: ");
      scanf("%d", &user_choice);


      switch(user_choice)
      {
         case 1:
            printf("\nInserting a node at beginning");
            data = getData();
            insert_at_beginning(data);
```

```
        break;

    case 2:
        printf("\nInserting a node at end");
        data = getData();
        insert_at_end(data);
        break;

    case 3:
        printf("\nInserting a node at the given position");
        data = getData();
        position = getPosition();
        insert_at_position(data, position);
        break;

    case 4:
        printf("\nDeleting a node from beginning\n");
        delete_from_beginning();
        break;

    case 5:
        printf("\nDeleting a node from end\n");
        delete_from_end();
        break;

    case 6:
        printf("\nDelete a node from given position\n");
        position = getPosition();
        delete_from_position(position);
        break;

    case 7:
        printf("\nPrinting the list from beginning\n\n");
        print_from_beginning();
        break;

    case 8:
            printf("\nPrinting the list from end\n\n");
            print_from_end(head);
            printf("NULL\n");
            break;

        case 9:
            printf("\nSearching the node data");
            data = getData();
            search_data(data);
            break;

        case 10:
            printf("\nUpdating the node data");
            data = getData();
            position = getPosition();
            update_node_data(data, position);
            break;

        case 11:
            printf("\nSorting the list\n");
            list_sort();
            break;

        case 12:
            printf("\nProgram was terminated\n\n");
            return 0;

        default:
            printf("\n\tInvalid Choice\n");
    }

    printf("\n..............................\n");
    printf("\nDo you want to continue? (Y/N) : ");

    scanf(" %c", &user_active);
}
```

```
    return 0;
}

/* prints the message when the memory was not allocated */
void memory_error_message()
{
    printf("\nMemory was not allocated!\n");
}

// prints the message when the position is not valid for operation
void invalid_pos_message()
{
    printf("\nInvalid position!\n");
}

// prints the message when the linked list is empty
void empty_list_message()
{
    printf("\nList is Empty!\n");
}
 struct node* create_node(int data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    if (new_node == NULL)
    {
        return NULL;
    }
    else
    {
        new_node->prev = NULL;
        new_node->data = data;
        new_node->next = NULL;
    }
}

/* inserts a new node at beginning of the list */
```

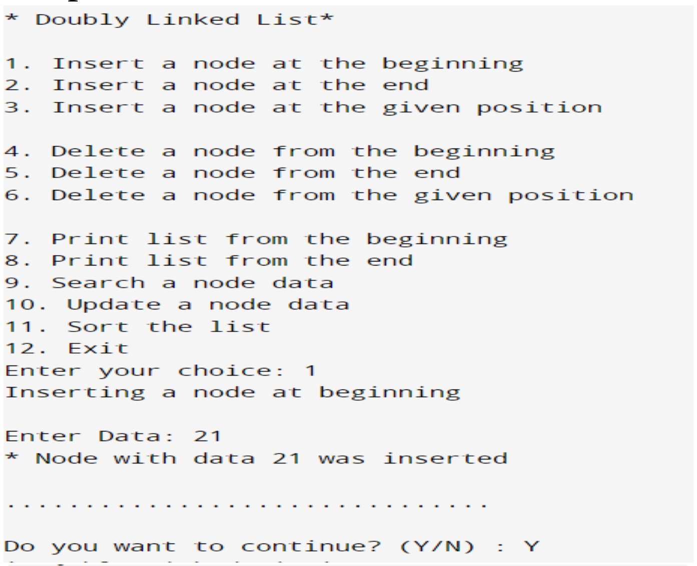
```
void insert_at_beginning(int data)
{
   struct node* new_node = create_node(data);

   if (new_node == NULL)
   {
      memory_error_message();
      return;
   }
   else if(head == NULL)
   {
      head = new_node;
   }
   else
   {
      head->prev = new_node;
      new_node->next = head;
      head = new_node;
   }
   printf("\n* Node with data %d was inserted \n", data);
}

/* inserts a new node at the end of the list */
void insert_at_end(int data)
{
   struct node* new_node = create_node(data);

   if (new_node == NULL)
   {
      memory_error_message();
      return;
   }
   else if (head == NULL)
   {
      head = new_node;
   }
   else
```

```
    {
        struct node* temp = head;
        //traverse to the last node
        while (temp->next != NULL)
        {
            temp = temp = temp->next;
        }
        temp->next = new_node;
        new_node->prev = temp;
    }
    printf("\n* Node with data %d was inserted \n", data);
}

/* inserts a new node at the given position */
void insert_at_position(int data, int pos)
{
    struct node* new_node = create_node(data);
    int size = size_of_list();

    if (new_node == NULL)
    {
        memory_error_message();
        return;
    }
    else if (head != NULL && (pos < 1 || pos > size))
    {
        invalid_position_message();
        return;
    }
    else if (head == NULL && pos == 1)
    {
        head = new_node;
    }
    else if (head != NULL && pos == 1)
    {
        new_node->next = head;
        head->prev = new_node;
        head = new_node;
    }
    else
    {

        struct node* temp = head;
        int count = 1;

        //traverse to the before given position
        while (++count < pos)
        {
            temp = temp->next;
        }

        temp->next->prev = new_node;
        new_node->next = temp->next;

        temp->next = new_node;
        new_node->prev = temp;
    }
    printf("\n* Node with data %d was inserted \n", data);
}

/* deletes a node from the beginning of the list */
void delete_from_beginning()
{
    if (head == NULL)
    {
        empty_list_message();
        return;
    }
    struct node* temp = head;
    head = head->next;
    int data = temp->data;

    //free the memory from the heap
    free(temp);
    printf("\n* Node with data %d was deleted \n", data);
}

/* deletes a node from the end of the list */
void delete_from_end()
{

    if (head == NULL)
    {
        empty_list_message();
        return;
    }
    struct node* temp = head;
    int data = 0;

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

    if (temp->prev == NULL)
    {
        head = NULL;
    }
    else
    {
        temp->prev->next = temp->next;
    }
    data = temp->data;

    free(temp);
    printf("\n* Node with data %d was deleted \n", data);
}

/* deletes a node from the given position */
void delete_from_position(int pos)
```

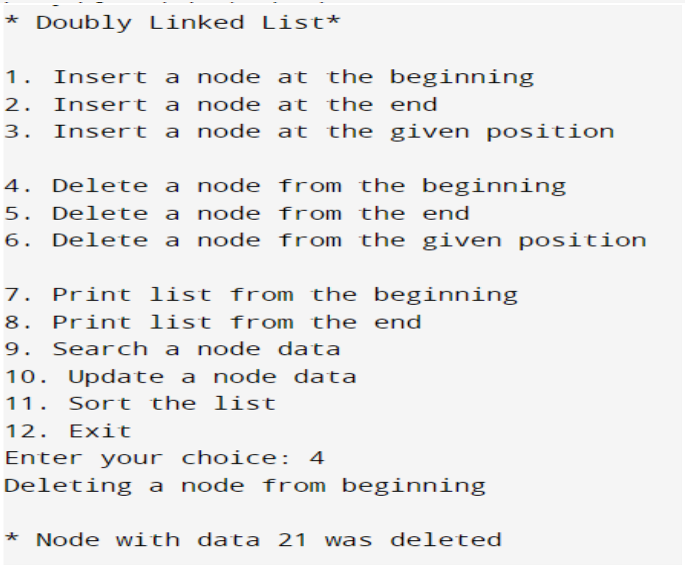
```
{

    if (head == NULL)
    {
        empty_list_message();
        return;
    }
    int size = size_of_list();
    struct node* temp = head;
    int data = 0;

    if (pos < 1 || pos > size)
    {
        invalid_pos_message();
        return;
    }
    else if (pos == 1)
    {
        head = head->next;
        data = head->data;
        free(temp);
        printf("\n* Node with data %d was deleted \n", data);
    }
    else
    {
        int count = 0;

        while (++count < pos)
        {
            temp = temp->next;
        }

        //update previous node
        temp->prev->next = temp->next;

        // if deleting the last node then just update the previous node
        if (pos != size)
        {
            //update next node
            temp->next->prev = temp->prev;
        }
        data = temp->data;

        //free memory
        free(temp);

        printf("\n* Node with data %d was deleted \n", data);
    }
}

/* prints the data of nodes from the beginning of the list */
void print_from_beginning()
{
    struct node* temp = head;

    while (temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

/* prints the data of nodes from the end of the list */
void print_from_end(struct node* head)
{
    if (head == NULL)
    {
        return;
    }

    print_from_end(head->next);
    printf("%d  ", head->data);
}
```

```
/* search a data node with the given value */
void search_data(int data)
{
    struct node* temp = head;
    int position = 0;
    int flag = 0;

    while (temp != NULL)
    {
        position += 1;
        if (temp->data == data)
        {
            flag = 1;
            break;
        }
        temp = temp->next;
    }

    if (flag == 0)
    {
        printf("\nNode with data %d was not found\n", data);
    }
    else
    {
        printf("\nNode found at %d position\n", position);
    }
}

/* updates a node from the given position */
void update_node_data(int data, int pos)
{
    if (head == NULL)
    {
        empty_list_message();
        return;
    }
    int size = size_of_list();

    if (pos < 1 || pos > size)
    {
        invalid_position_message();
        return;
    }

    struct node* temp = head;
    int count  = 1;

    while (++count < pos)
    {
        temp = temp->next;
    }

    temp->data = data;

    printf("\nNode Number %d was Updated!\n", pos);
}

/* sort the linked list data using insertion sort */
void list_sort()
{
    if (head == NULL)
    {
       empty_list_message();
       return;
    }
    struct node* temp1 = head;
    struct node* temp2 = head;
    int key = 0;

    while (temp1 != NULL)
    {
        key = temp1->data;
        temp2 = temp1;

        while (temp2->prev != NULL && temp2->prev->data > key)
        {
            temp2->data = temp2->prev->data;
            temp2 = temp2->prev;
        }
        temp2->data = key;
        temp1 = temp1->next;
    }

    printf("\nList was sorted!\n");
}

/* getting node data from the user */
int getData()
{
    int data;
    printf("\n\nEnter Data: ");
    scanf("%d", &data);

    return data;
}

/* getting node position from the user */
int getPosition()
{
    int position;
    printf("\nEnter Position: ");
    scanf("%d", &position);

    return position;
}

/* finding the size of the list */
int size_of_list()
{
    struct node* temp = head;
    int count = 0;

    while (temp != NULL)
    {
        count += 1;
        temp = temp->next;
    }

    return count;
}
```

Output:



```
* Doubly Linked List*

1. Insert a node at the beginning
2. Insert a node at the end
3. Insert a node at the given position

4. Delete a node from the beginning
5. Delete a node from the end
6. Delete a node from the given position

7. Print list from the beginning
8. Print list from the end
9. Search a node data
10. Update a node data
11. Sort the list
12. Exit
Enter your choice: 1
Inserting a node at beginning

Enter Data: 21
* Node with data 21 was inserted

...............................

Do you want to continue? (Y/N) : Y
```



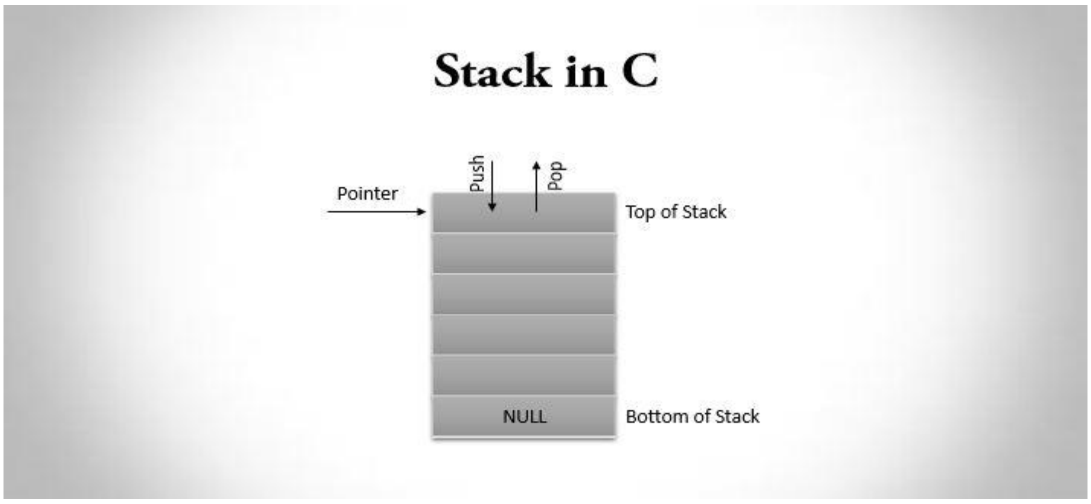
```
* Doubly Linked List*

1. Insert a node at the beginning
2. Insert a node at the end
3. Insert a node at the given position

4. Delete a node from the beginning
5. Delete a node from the end
6. Delete a node from the given position

7. Print list from the beginning
8. Print list from the end
9. Search a node data
10. Update a node data
11. Sort the list
12. Exit
Enter your choice: 4
Deleting a node from beginning

* Node with data 21 was deleted
```

Practical Number 5:[Applications of **Stack**]

<u>About Stack:</u>

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out).

General Structure:



Real-time applications:

- History of a web browser is stored in the form of a stack.
- E-mails, and Google photos in any gallery are also stored in form of a stack.

- For Undo and redo notifications in editors.

1. Implement a stack using an array and using a linked list.

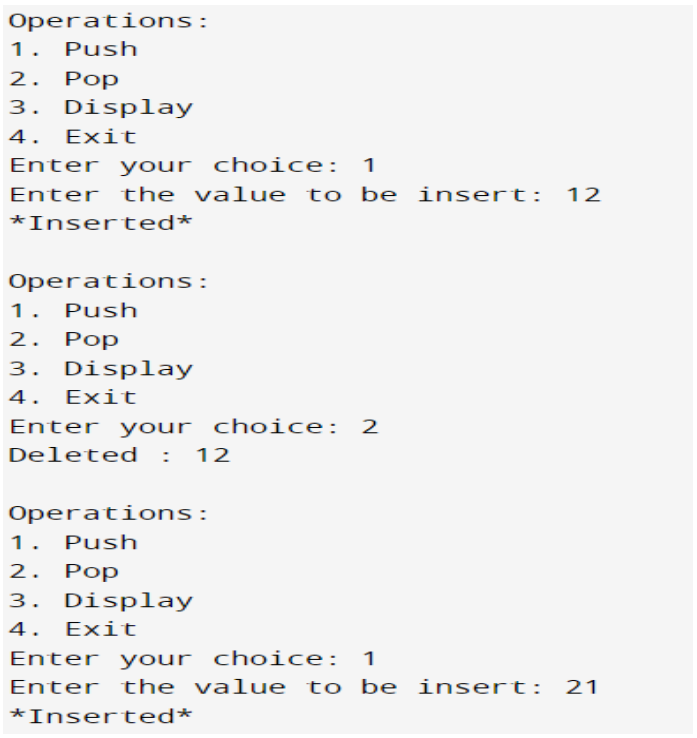
Code:

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

#define SIZE 10

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

int stack[SIZE], top = -1;

void main()
{
   int value, choice;
```

```
  while(1){
    printf("\n\nOperations:\n");
    printf("1. Push\n2. Pop\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
       case 1: printf("Enter the value to be insert: ");
              scanf("%d",&value);
              push(value);
              break;
       case 2: pop();
              break;
       case 3: display();
              break;
       case 4: exit(0);

       default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void push(int value){
  if(top == SIZE-1)
    printf("\nStack is Full!!! Insertion is not possible!!!");
  else{
    top++;
    stack[top] = value;
```

```
      printf("\n*Inserted*");
   }
}
void pop(){
   if(top == -1)
      printf("\nStack is Empty! Deletion is not possible.");
   else{
      printf("\nDeleted : %d", stack[top]);
      top--;
   }
}
void display(){
   if(top == -1)
      printf("\nStack is Empty!!!");
   else{
      int i;
      printf("\nStack elements are:\n");
      for(i=top; i>=0; i--)
          printf("%d\n",stack[i]);
   }
}
```

Output:



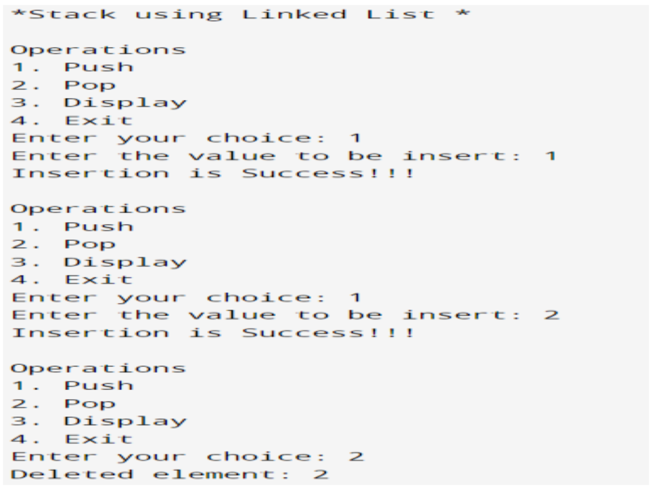
```
Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
*Inserted*

Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted : 12

Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 21
*Inserted*
```

<u>Stack using Linked list:</u>

Code:

```
#include<stdio.h>
struct Node
{
   int data;
   struct Node *next;
}
*top = NULL;
void push(int);
void pop();
void display();
void main()
{
   int choice, value;

   printf("\n *Stack using Linked List *\n");
   while(1){
      printf("\nOperations\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
          case 1: printf("Enter the value to be insert: ");
                   scanf("%d", &value);
                   push(value);
                   break;
          case 2: pop(); break;
          case 3: display(); break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
         printf("%d--->",temp->data);
         temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

Output:



```
*Stack using Linked List *

Operations
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 1
Insertion is Success!!!

Operations
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 2
Insertion is Success!!!

Operations
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Deleted element: 2
```

2. Given a stack, sort it using recursion. Use of any loop constructs like while, for, etc.is not allowed. We can only use the following functions on Stack S:

a. isEmpty(S): Tests whether stack is empty or not.

b. push(S): Adds new element to the stack.

c. pop(S): Removes top element from the stack.

d. top(S): Returns value of the top element.

Code:

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

// Stack is represented using linked list
struct stack
```

```
{
int data;
struct stack *next;
};

// Utility function to initialize stack
void initStack(struct stack **s)

{
*s = NULL;
}
// Utility function to chcek if stack is empty
int isEmpty(struct stack *s)

{

if (s == NULL)

    return 1;

return 0;
}

// Utility function to push an item to stack
void push(struct stack **s, int x)
```

```
{

struct stack *p = (struct stack *)malloc(sizeof(*p));

if (p == NULL)

{
    printf("Memory allocation failed.\n");
    return;

}

p->data = x;
p->next = *s;

*s = p;
}

// Utility function to remove an item from stack

int pop(struct stack **s)

{
int x;
struct stack *temp;
x = (*s)->data;
```

```
temp = *s;

(*s) = (*s)->next;

free(temp);

return x;

}


// Function to find top item

int top(struct stack *s)

{

return (s->data);

}

// Recursive function to insert an item x in sorted way

void sortedInsert(struct stack **s, int x)


{

if (isEmpty(*s) || x > top(*s))


{

    push(s, x);

    return;


}


// If top is greater, remove the top item and recur


int temp = pop(s);
```

```
sortedInsert(s, x);
// Put back the top item removed earlier
push(s, temp);
}

// Function to sort stack
void sortStack(struct stack **s)

{

// If stack is not empty

if (!isEmpty(*s))

{

    // Remove the top item

    int x = pop(s);

    // Sort remaining stack

    sortStack(s);

    // Push the top item back in sorted stack
    sortedInsert(s, x);

}
}

// function to print contents of stack

void printStack(struct stack *s)

{

while (s)

{
    printf("%d ", s->data);
    s = s->next;

}

printf("\n");
}
int main(void)

{

struct stack *top;
```

```
initStack(&top);

push(&top, 30);

push(&top, -5);

push(&top, 18);

push(&top, 14);
printf("Stack elements before sorting:\n");
printStack(top);
sortStack(&top);
printf("\n\n");
printf("Stack elements after sorting:\n");
printStack(top);
return 0;
}
```

Output:

```
Stack elements before sorting:
14 18 -5 30


Stack elements after sorting:
-5 14 18 30
```

Experiment No. 6 [Stack Applications]

About Polish Notations:

- An expression consists of constants, variables, and symbols. Symbols can be operators or parenthesis.
- When the operator is written in between the operands, then it is known as **infix** notation,
- Example: (p + q) * (r + s)
- The postfix expression is an expression in which the operator is written after the operands.
- Example: 23+

Real-time Applications:

1. HP9100A calculator was designed using reverse polish notations,

1. Convert the given infix expression into postfix expression using stack.

Example- Input: $a + b * (c\wedge d - e)\wedge(f + g * h) - i$

Output: $abcd\wedge e - fgh * +\wedge * +i -$

Code:

```
#include<stdio.h>
#include<stdlib.h>
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack= (struct Stack*)malloc(sizeof(struct Stack));

    if (!stack)
        return NULL;

    stack->top = -1;
    stack->capacity = capacity;

    stack->array
        = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}
```

```
// A utility function to check if
// the given character is operand
int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z')
           || (ch >= 'A' && ch <= 'Z');
}

// A utility function to return
// precedence of a given operator
// Higher returned value means
// higher precedence
int Prec(char ch)
{
    switch (ch) {
    case '+':
    case '-':
        return 1;

    case '*':
    case '/':
        return 2;

    case '^':
        return 3;
    }
    return -1;
}
// The main function that
// converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

    // Create a stack of capacity
    // equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    if (!stack) // See if stack was created successfully
        return -1;

    for (i = 0, k = -1; exp[i]; ++i) {

        // If the scanned character is
        // an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        // If the scanned character is an
        // '(', push it to the stack.
        else if (exp[i] == '(')
            push(stack, exp[i]);
        // If the scanned character is an ‘)’,
        // pop and output from the stack
        // until an ‘(‘ is encountered.
        else if (exp[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }

        else // an operator is encountered
        {
            while (!isEmpty(stack)
                   && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }
    }

    // pop all the operators from the stack
    while (!isEmpty(stack))
        exp[++k] = pop(stack);
    exp[++k] = '\0';
    printf("%s", exp);
}
int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";

    // Function call
    infixToPostfix(exp);
    return 0;
}
```

Output:

```
abcd^e-fgh*+^*+i-
```

2. Write a program to evaluate the following given postfix expressions:

a. 2 3 1 ∗ + 9 – Output: -4

b. 2 2 + 2 / 5 ∗ 7 + Output: 17

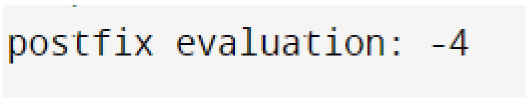
a) Code:

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

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}
```

```
char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

// that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;
    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two
        // elements from stack apply the operator
        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}
```

```
int main()
{
    char exp[] = "231*+9-";
    printf ("postfix evaluation: %d", evaluatePostfix(exp));
    return 0;
}
```

Output:



```
postfix evaluation: -4
```

b) Code:

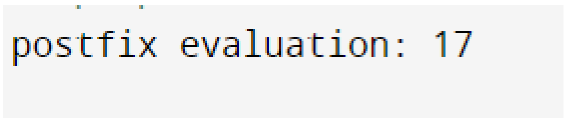
```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

// Stack type
struct Stack
{
    int top;
    unsigned capacity;
    int* array;
```

```
};

// Stack Operations
struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
```

```
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}



// The main function that returns value of a given postfix expression
int evaluatePostfix(char* exp)
{
    // Create a stack of capacity equal to expression size
    struct Stack* stack = createStack(strlen(exp));
    int i;

    // See if stack was created successfully
    if (!stack) return -1;

    // Scan all characters one by one
    for (i = 0; exp[i]; ++i)
    {
        // If the scanned character is an operand (number here),
        // push it to the stack.
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        // If the scanned character is an operator, pop two

        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

int main()
```

```
{
    char exp[] = "22+2/5*7+";
    printf ("postfix evaluation: %d", evaluatePostfix(exp));
    return 0;
}
```

Output:



```
postfix evaluation: 17
```

3. Given an expression, write a program to examine whether the pairs and the orders of

"{", "}", "(", ")", "[", "]" are correct in the expression or not.

Example: a)Input: exp = "[( )]{ }{[( )( )]( )}" Output: Balanced

Input: exp = "[( ])" Output: Not Balance

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define bool int

// Structure of a stack node
struct sNode {
    char data;
    struct sNode* next;
};

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data);
```

```
// Function to pop an item from stack
int pop(struct sNode** top_ref);

// Returns 1 if character1 and character2 are matching left
// and right Brackets
bool isMatchingPair(char character1, char character2)
{
    if (character1 == '(' && character2 == ')')
        return 1;
    else if (character1 == '{' && character2 == '}')
        return 1;
    else if (character1 == '[' && character2 == ']')
        return 1;
    else
        return 0;
}

// Return 1 if expression has balanced Brackets
bool areBracketsBalanced(char exp[])
{
    int i = 0;

    // Declare an empty character stack
    struct sNode* stack = NULL;

    // Traverse the given expression to check matching
    // brackets
    while (exp[i]) {
        // If the exp[i] is a starting bracket then push
        // it
        if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
            push(&stack, exp[i]);

        // If exp[i] is an ending bracket then pop from
        // stack and check if the popped bracket is a
        // matching pair*/
        if (exp[i] == '}' || exp[i] == ')'
            || exp[i] == ']') {

            // If we see an ending bracket without a pair
            // then return false
            if (stack == NULL)
                return 0;

            // Pop the top element from stack, if it is not
            // a pair bracket of character then there is a
            // mismatch.
            // his happens for expressions like {(})
            else if (!isMatchingPair(pop(&stack), exp[i]))
                return 0;
        }
        i++;
    }

    // If there is something left in expression then there
    // is a starting bracket without a closing
    // bracket
    if (stack == NULL)
        return 1; // balanced
    else
        return 0; // not balanced
}

// Driver code
int main()
{
    char exp[100] = "{()}[]";

    // Function call
    if (areBracketsBalanced(exp))
        printf("Balanced \n");
    else
        printf("Not Balanced \n");
    return 0;
}

// Function to push an item to stack
void push(struct sNode** top_ref, int new_data)
```

```
{
    // allocate node
    struct sNode* new_node
        = (struct sNode*)malloc(sizeof(struct sNode));

    if (new_node == NULL) {
        printf("Stack overflow n");
        getchar();
        exit(0);
    }

    // put in the data
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*top_ref);
    // move the head to point to the new node
    (*top_ref) = new_node;
}
// Function to pop an item from stack
int pop(struct sNode** top_ref)
{
    char res;
    struct sNode* top;
    // If stack is empty then error
    if (*top_ref == NULL) {
        printf("Stack overflow n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}
```

Output:

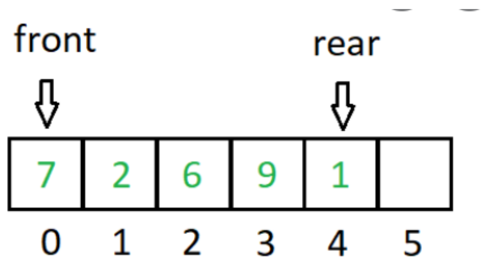
```
Balanced
```

Practical Number 7:[Implementing Queues]

About Queues:

The **queue** is a linear data structure that follows the **FIFO** pattern in which the element inserted first at the queue will be removed first.

General structure:



Real time applications:

1. As Waiting instructions between Bus interface unit and Execution Unit of Microprocessor.
2. Applied to add song/video in a playlist.
3. Useful in Operating Systems for handling interrupts

1. C program to Implement various functionalities of Queue using Arrays. For example: insertion, deletion, front element, rear element etc.

Code:

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

#define MAX 50

void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
void main()
{
   int choice;
   while (1)
   {
       Printf("*Operations on Queue:*);
      printf("1.Insert element to queue \n");
```

```
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(1);
            default:
            printf("Wrong choice \n");
        } /* End of switch */
    } /* End of while */
} /* End of main() */

void insert()
{
    int add_item;
    if (rear == MAX - 1)
    printf("Queue Overflow \n");
    else
    {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Inset the element in queue : ");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /* End of insert() */

void delete()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
    }
} /* End of delete() */

void display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
}
```

Output:

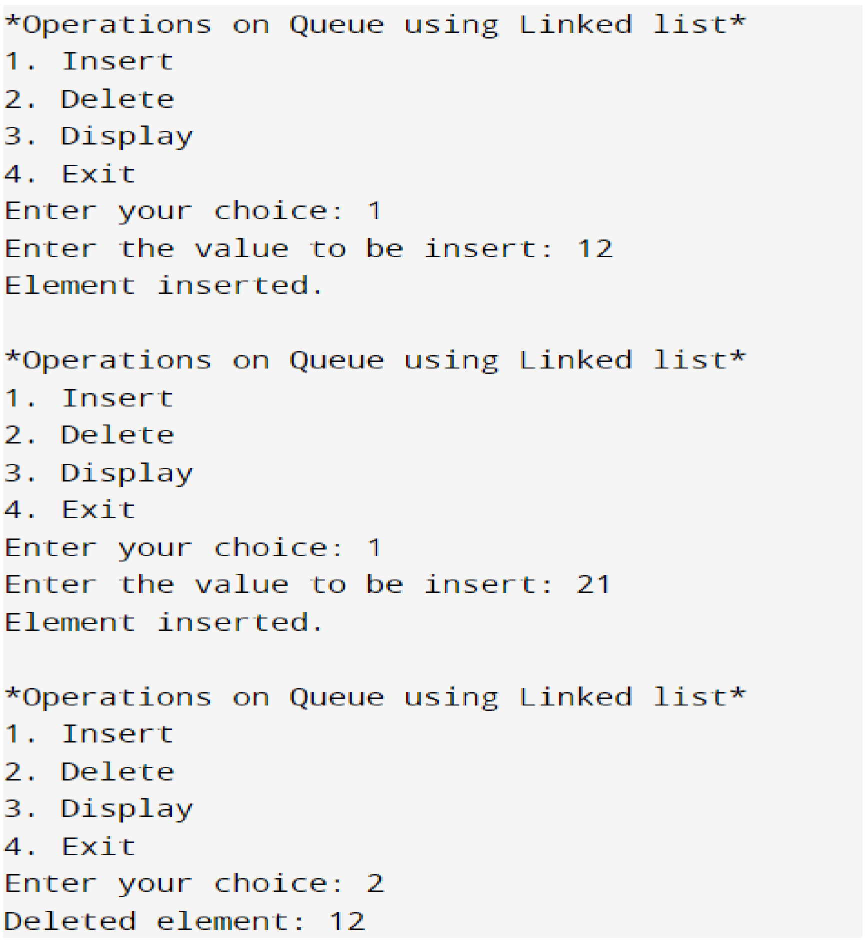
```
*Operations on Queue:*
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 12
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Insert the element in queue : 32
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
12 32
```

2. C program to Implement functionalities of queues using linked list

Code:

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

struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();
void main()
{
   int choice, value;



   while(1){
      printf("\n*Operations on Queue using Linked list*\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
          case 1: printf("Enter the value to be insert: ");
                   scanf("%d", &value);
                   insert(value);
                   break;
          case 2: delete(); break;
          case 3: display(); break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nElement inserted.\n");
}
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
          printf("%d--->",temp->data);
          temp = temp -> next;
      }
      printf("%d--->NULL\n",temp->data);
   }
}
```

Output:



```
*Operations on Queue using Linked list*
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 12
Element inserted.

*Operations on Queue using Linked list*
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 21
Element inserted.

*Operations on Queue using Linked list*
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted element: 12
```

3. C program to Implement Double Ended Queue that supports following operation:

a. insertFront(): Adds an item at the front of Deque.

b. insertLast(): Adds an item at the rear of Deque.

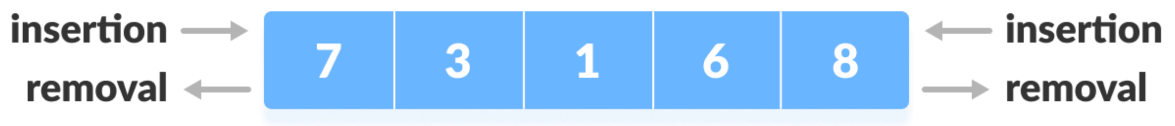
c. deleteFront(): Deletes an item from the front of Deque.

d. deleteLast(): Deletes an item from the rear of Deque

About Double Ended Queue:

Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

General Structure:



Real-time Applications:

- In a web browser's history, recently visited URLs are added to the front of the deque and the URL at the back of the deque is removed after some specified number of operations of insertions at the front.

- Storing a software application’s list of undo operations.

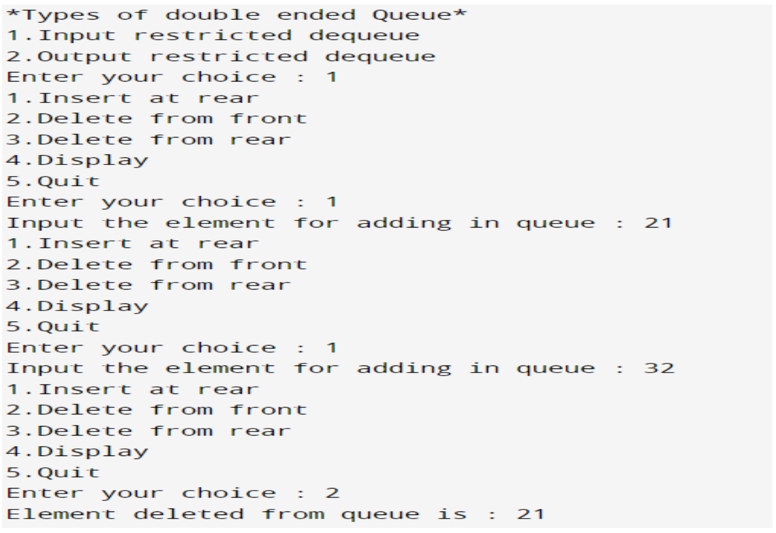
Code:

```
# include<stdio.h>
# define Size 10

int deque_arr[Size];
int front = -1;
int rear = -1;

/*Begin of insert_rear*/
void insert_rear()
{
    int added_item;
    if((front == 0 && rear == Size-1) || (front == rear+1))
    {   printf("Queue Overflow\n");
        return;}
    if (front == -1)  /* if queue is initially empty */
    {   front = 0;
        rear = 0;}
    else
    if(rear == Size-1)  /*rear is at last position of queue */
        rear = 0;
    else
        rear = rear+1;
    printf("Input the element for adding in queue : ");
    scanf("%d", &added_item);
    deque_arr[rear] = added_item ;
}
/*End of insert_rear*/

/*Begin of insert_front*/
void insert_front()
{   int added_item;
    if((front == 0 && rear == Size-1) || (front == rear+1))
    {   printf("Queue Overflow \n");
        return;  }
    if (front == -1)/*If queue is initially empty*/
    {   front = 0;
        rear = 0;    }
    else
    if(front== 0)
        front=Size-1;
    else
        front=front-1;
    printf("Input the element for adding in queue : ");
    scanf("%d", &added_item);
    deque_arr[front] = added_item ;  }
/*End of insert_front*/

/*Begin of delete_front*/
```

```
void delete_front()
{   if (front == -1)
    {   printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",deque_arr[front]);
    if(front == rear) /*Queue has only one element */
    {   front = -1;
        rear=-1;
    }
    else
        if(front == Size-1)
            front = 0;
        else
            front = front+1;
}
/*End of delete_front*/

/*Begin of delete_rear*/
void delete_rear()
{
    if (front == -1)
    {
        printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",deque_arr[rear]);
    if(front == rear) /*queue has only one element*/
    {
        front = -1;
        rear=-1;
    }
    else
        if(rear == 0)
            rear=Size-1;
        else
            rear=rear-1;    }
/*End of delete_rear*/

/*Begin of input_que*/
void display_queue()
{
    int front_pos = front,rear_pos = rear;

    if(front == -1)
    {   printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
    if( front_pos <= rear_pos )
    {
        while(front_pos <= rear_pos)
        {
            printf("%d ",deque_arr[front_pos]);
            front_pos++;
        }
    }
    else
    {
        while(front_pos <= Size-1)
        {   printf("%d ",deque_arr[front_pos]);
            front_pos++;
        }
        front_pos = 0;
        while(front_pos <= rear_pos)
        {
            printf("%d ",deque_arr[front_pos]);
            front_pos++;
        }
    }
    printf("\n");
}
/*End of display_queue*/

/*Begin of input_que*/
void input_que()
{   int choice;
    do
```

```
{   printf("1.Insert at rear\n");
    printf("2.Delete from front\n");
    printf("3.Delete from rear\n");
    printf("4.Display\n");
    printf("5.Quit\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);

    switch(choice)
    {   case 1:
        insert_rear();
        break;
     case 2:
        delete_front();
        break;
     case 3:
        delete_rear();
        break;
     case 4:
        display_queue();
        break;
     case 5:
        break;
     default:
        printf("Wrong choice\n");
    }
    }while(choice!=5);
}
/*End of input_que*/

/*Begin of output_que*/
void output_que()
{   int choice;
    do
    {   printf("1.Insert at rear\n");
        printf("2.Insert at front\n");
        printf("3.Delete from front\n");
        printf("4.Display\n");
        printf("5.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
         case 1:
            insert_rear();
            break;
         case 2:
            insert_front();
            break;
         case 3:
            delete_front();
            break;
         case 4:
            display_queue();
            break;
         case 5:
            break;
         default:
            printf("Wrong choice\n");
        }
    }while(choice!=5);
}
/*End of output_que*/

/*Begin of main*/
main()
{   int choice;
printf("*Types of double ended Queue*");
    printf("1.Input restricted dequeue\n");
    printf("2.Output restricted dequeue\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
     case 1 :
        input_que();
        break;
     case 2:
        output_que();
        break;
     default:
        printf("Wrong choice\n");
    }
}
```

Output:



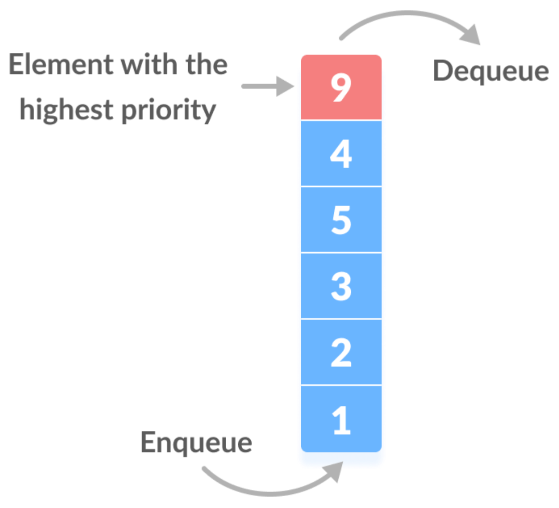
```
*Types of double ended Queue*
1.Input restricted dequeue
2.Output restricted dequeue
Enter your choice : 1
1.Insert at rear
2.Delete from front
3.Delete from rear
4.Display
5.Quit
Enter your choice : 1
Input the element for adding in queue : 21
1.Insert at rear
2.Delete from front
3.Delete from rear
4.Display
5.Quit
Enter your choice : 1
Input the element for adding in queue : 32
1.Insert at rear
2.Delete from front
3.Delete from rear
4.Display
5.Quit
Enter your choice : 2
Element deleted from queue is : 21
```

4. C program for implementing Priority Queue:

About Priority Queues:

A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a “priority” associated with it. In a priority queue, an element with high priority is served before an element with low priority.

General structure:



Real-time applications:

1. In many operating Systems, processes with shortest execution time are executed first which is done on basis of priority.
2. Giving priority in Recruiting candidates based on resume and CV.

Code:

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

#define MAX 10

void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();

int pri_que[MAX];
int front, rear;

void main()
{
    int n, ch;

    printf("\n1 - Insert an element into queue");
    printf("\n2 - Delete an element from queue");
    printf("\n3 - Display queue elements");
    printf("\n4 - Exit");

    create();
```

```
while (1)
{
    printf("\nEnter your choice : ");
    scanf("%d", &ch);

    switch (ch)
    {
    case 1:
        printf("\nEnter value to be inserted : ");
        scanf("%d",&n);
        insert_by_priority(n);
        break;
    case 2:
        printf("\nEnter value to delete : ");
        scanf("%d",&n);
        delete_by_priority(n);
        break;
    case 3:
        display_pqueue();
        break;
    case 4:
        exit(0);
    default:
        printf("\nChoice is incorrect, Enter a correct choice");
    }
}
```

```
}

/* Function to create an empty priority queue */
void create()
{
    front = rear = -1;
}

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
    if (rear >= MAX - 1)
    {
        printf("\nQueue overflow no more elements can be inserted");
        return;
    }
    if ((front == -1) && (rear == -1))
    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;
    }
    else
        check(data);
    rear++;
```

```
}

/* Function to check priority and place element */
void check(int data)
{
    int i,j;

    for (i = 0; i <= rear; i++)
    {
        if (data >= pri_que[i])
        {
            for (j = rear + 1; j > i; j--)
            {
                pri_que[j] = pri_que[j - 1];
            }
            pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}

/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;
    if ((front==-1) && (rear==-1))
    {
        printf("\nQueue is empty no elements to delete");
        return;
    }

    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }

        pri_que[i] = -99;
        rear--;

        if (rear == -1)
            front = -1;
        return;
        }
    }
    printf("\n%d not found in queue to delete", data);
}
```

```
/* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }

    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }

    front = 0;
}
```

Output:

```
*Operations on Priority Queue*
1 - Insert an element into queue
2 - Delete an element from queue
3 - Display queue elements
4 - Exit
Enter your choice : 1
Enter value to be inserted : 12
Enter your choice : 1
Enter value to be inserted : 21
Enter your choice : 2
Enter value to delete : 21
Enter your choice : 3
12
```

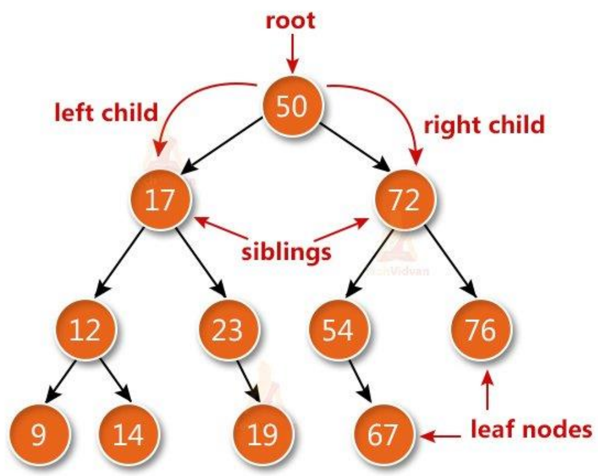
Practical Number-8:[Implementing Trees]

About trees:

- A tree is a non-linear data structure in which the data is stored non-linearly. A tree is used to represent the data in hierarchical form.

- A binary tree is a kind of tree in which every node contains at most two children. A node contains the address of the left and right child in the linked list representation of the binary tree.

- General Structure:



- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways:

  1. Pre-order Traversal

  2. In-order Traversal

  3.Post-order Traversal

1. C program to implement binary tree, perform operation of insertion and deletion and perform the three traversal methods.

   Code:

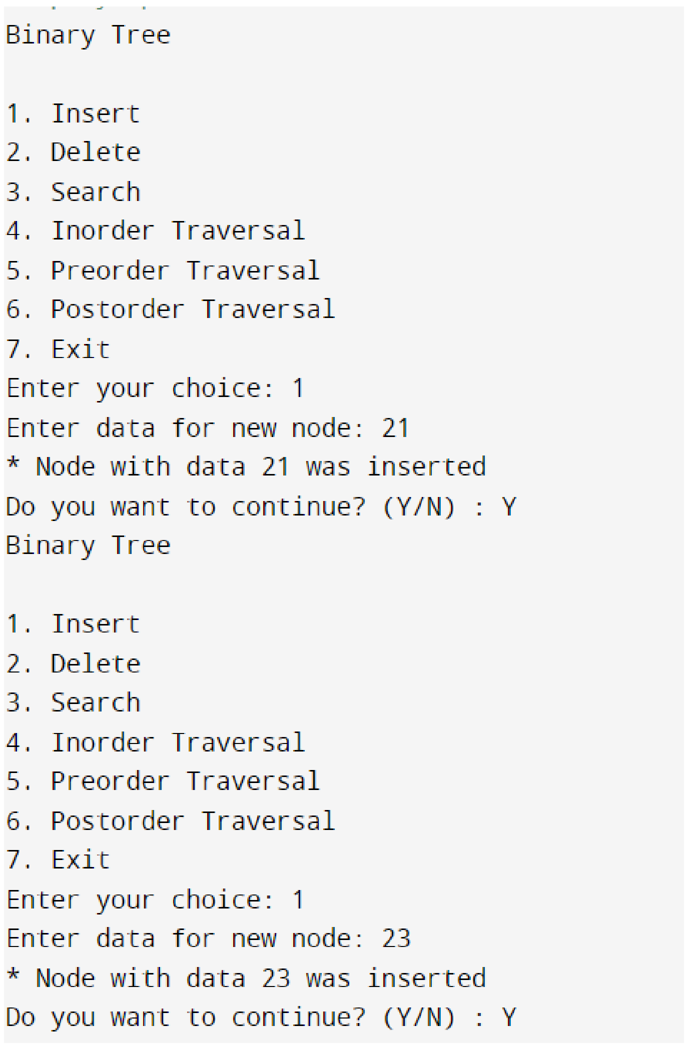
```
#include<stdio.h>
#include<stdlib.h>
#define MAX_SIZE 25
#define max(a, b) a>b ? a : b

struct TreeNode
{
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};

// global root declaration
struct TreeNode* root = NULL;

// function prototyping
void insert(int);
void delete(int);
int search(int);
void inorder(struct TreeNode*);
void preorder(struct TreeNode*);
void postorder(struct TreeNode*);
void levelorder();

int main()
{
   char user_active = 'Y';
    int user_choice;
    int node_data, position;


    while(user_active == 'Y' || user_active == 'y')
    {
        printf("\n Binary Tree\n");
        printf("\n1. Insert ");
        printf("\n2. Delete");
        printf("\n3. Search");
        printf("\n4. Inorder Traversal");
        printf("\n5. Preorder Traversal");
        printf("\n6. Postorder Traversal");
        printf("\n7. Exit");
        printf("\nEnter your choice: ");
        scanf("%d", &user_choice);

        switch(user_choice)
        {
           case 1:
              printf("Enter data for new node: ");
              scanf("%d", &node_data);
              insert(node_data);
              break;

           case 2:
              printf("Enter node data: ");
              scanf("%d", &node_data);
              delete(node_data);
              break;

           case 3:
              printf("Enter node data: ");
              scanf("%d", &node_data);
              int has_found = search(node_data);

              if(has_found == -1) {
                 printf("\nNode was not found!");
              } else {
                 printf("\nNode was found");
              }
              break;



           case 4:
              printf("Inorder Traversal\n\n");
              inorder(root);
              break;

           case 5:
              printf("Preorder Traversal\n\n");
              preorder(root);
              break;
```

```
            case 6:
               printf("Postorder Traversal\n\n");
               postorder(root);
               break;

            case 8:
               printf("Level order Traversal\n\n");
               levelorder();
               break;

            case 7:
               printf("Program is terminating...\n\n");
               exit(0);

            default:
               printf("Invalid choice");
         }

            printf("\nDo you want to continue? (Y/N) : ");

         scanf(" %c", &user_active);
      }

      return 0;

      }

   struct TreeNode* create(int data)
   {
      struct TreeNode* new_node = (struct TreeNode*) malloc (sizeof(struct TreeNode));
      if(new_node == NULL)
      {
         printf("\nMemory can't be allocated for new node\n");
         return NULL;
      }

      new_node->left = NULL;
      new_node->right = NULL;
      new_node->val = data;

      return new_node;
```

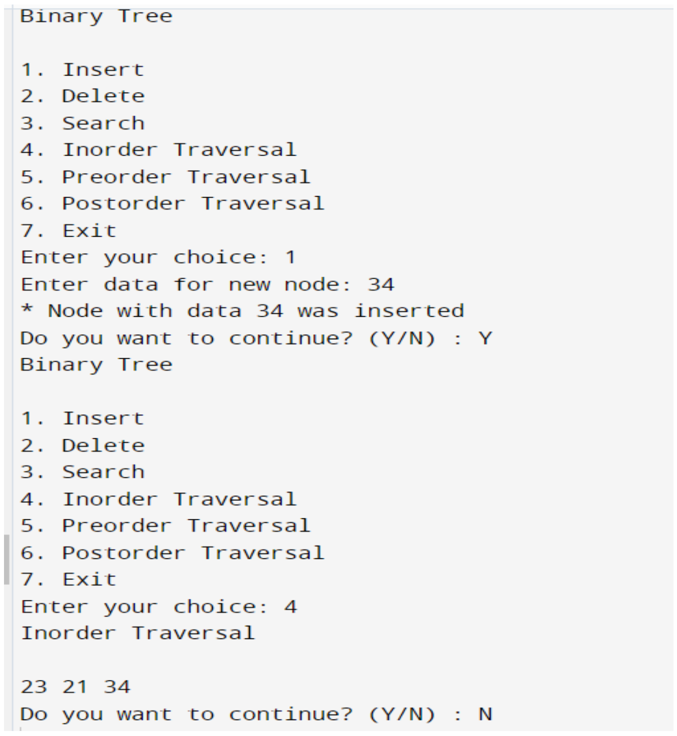
```
}

// inserts a new node in the tree
void insert(int data)
{
   if(root == NULL)
   {
      struct TreeNode* new_node = create(data);
      if(new_node)
      {
         root = new_node;
         printf("\n * Node with data %d was inserted", data);
      }
      return;
   }

   struct TreeNode* queue[MAX_SIZE];
   struct TreeNode* new_node = NULL;
   int front = -1;
   int rear = -1;
   queue[front+1] = root;
   front = rear = 0;

   while(front <= rear)
   {
      struct TreeNode* temp = queue[front];
      front++;

      if(temp->left != NULL)
      {
         queue[++rear] = temp->left;
      }
      else
      {
         new_node = create(data);
         if(new_node)
         {
            temp->left = new_node;
            printf("\n* Node with data %d was inserted", data);
         }
         return;
      }
```

```
        if(temp->right != NULL)
        {
            queue[++rear] = temp->right;
        }
        else
        {
            new_node = create(data);
            if(new_node)
            {
                temp->right = new_node;
                printf("\n* Node with data %d was inserted", data);
            }
            return;
        }
    }
}

void delete(int key)
{
    // Tree is empty
    if(root == NULL)
    {
        return;
    }

    // Tree having only one node
    if(root->left == NULL && root->right == NULL)
    {
        if(root->val == key)
        {
            root = NULL;
            printf("\n* Node with data %d was deleted", key);
            return;
        }
        else
        {
            return;
        }
    }

    struct TreeNode* temp = NULL, *last_node = NULL, *key_node = NULL;
    struct TreeNode* queue[MAX_SIZE];
    int front = -1;
    int rear = -1;

    queue[front + 1] = root;
    front = rear = 0;

    // do level order traversal to find the deepest node
    while (front <= rear)
    {
        temp = queue[front];
        front++;

        if (temp->val == key)
        {
            key_node = temp;
        }

        if (temp->left != NULL)
        {
            last_node = temp;
            queue[++rear] = temp->left;
        }

        if (temp->right != NULL)
        {
            last_node = temp;
            queue[++rear] = temp->right;
        }
    }

    // if key is found in the binary tree
    if (key_node != NULL)
    {
        // replace the keynode data with the deepest node
        key_node->val = temp->val;

        // free the last node after updating its parent
        if (last_node->right == temp)
        {
            last_node->right = NULL;
        }
        else
        {
            last_node->left = NULL;
        }
        printf("\n* Node with data %d was deleted", key);
        free(temp);
        return;
    }
    printf("\n* Node with data %d was not found", key);
}

int search(int key)
{
    if (root == NULL)
    {
        return -1;
    }
    struct TreeNode* queue[MAX_SIZE];
    int front = -1;
    int rear = -1;

    queue[front + 1] = root;
    front = rear = 0;

    /
    while (front <= rear)
    {
        struct TreeNode* temp = queue[front];
        front++;

        if (temp->val == key)
        {
            return 1;
        }

        if (temp->left != NULL)
        {
            queue[++rear] = temp->left;
        }

        if (temp->right != NULL)
        {
            queue[++rear] = temp->right;
        }
    }
    return -1;
}



// inorder traversal
void inorder(struct TreeNode* root)
{
    if(root == NULL) return;

    inorder(root->left);
    printf("%d ", root->val);
    inorder(root->right);
}

// preorder traversal
void preorder(struct TreeNode* root)
{
    if(root == NULL) return;

    printf("%d ", root->val);
    preorder(root->left);
    preorder(root->right);
}

// postorder traversal
void postorder(struct TreeNode* root)
{
    if(root == NULL) return;

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->val);
}

// level order traversal
void levelorder()
{
    if (root == NULL)
    {
        printf("Tree is Empty!");
        return;
    }

    struct TreeNode* queue[MAX_SIZE];
    int front = -1;
    int rear = -1;

    queue[front + 1] = root;
    front = rear = 0;

    // do level order traversal to find the deepest node
    while (front <= rear)
    {
        struct TreeNode* temp = queue[front];
        front++;

        printf("%d ", temp->val);

        if (temp->left != NULL)
        {
            queue[++rear] = temp->left;
        }

        if (temp->right != NULL)
        {
            queue[++rear] = temp->right;
        }
    }
}
```

Output:



```
Binary Tree

1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Preorder Traversal
6. Postorder Traversal
7. Exit
Enter your choice: 1
Enter data for new node: 21
* Node with data 21 was inserted
Do you want to continue? (Y/N) : Y
Binary Tree

1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Preorder Traversal
6. Postorder Traversal
7. Exit
Enter your choice: 1
Enter data for new node: 23
* Node with data 23 was inserted
Do you want to continue? (Y/N) : Y
```



```
Binary Tree

1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Preorder Traversal
6. Postorder Traversal
7. Exit
Enter your choice: 1
Enter data for new node: 34
* Node with data 34 was inserted
Do you want to continue? (Y/N) : Y
Binary Tree

1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Preorder Traversal
6. Postorder Traversal
7. Exit
Enter your choice: 4
Inorder Traversal

23 21 34
Do you want to continue? (Y/N) : N
```

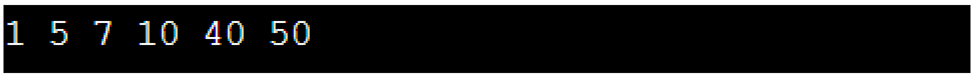
Q3] Given a preorder traversal sequence of a Binary Search Tree, construct the corresponding Binary Search Tree.

Code:

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

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

struct node* newNode(int data)
{
        struct node* temp
                = (struct node*)malloc(sizeof(struct node));
        temp->data = data;
        temp->left = temp->right = NULL;

        return temp;
}
struct node* constructTreeUtil(int pre[], int* preIndex,
                                                int low, int high, int size)
{
        // Base case
        if (*preIndex >= size || low > high)
                return NULL;
        struct node* root = newNode(pre[*preIndex]);
        *preIndex = *preIndex + 1;
        if (low == high)
                return root;
        int i;
        for (i = low; i <= high; ++i)
                if (pre[i] > root->data)
                        break;
        root->left = constructTreeUtil(pre, preIndex, *preIndex,
                                                        i - 1, size);
        root->right
                = constructTreeUtil(pre, preIndex, i, high, size);

        return root;
}
struct node* constructTree(int pre[], int size)
{
        int preIndex = 0;
        return constructTreeUtil(pre, &preIndex, 0, size - 1,
                                                size);
}
void printInorder(struct node* node)
{
        if (node == NULL)
                return;
        printInorder(node->left);
        printf("%d ", node->data);
        printInorder(node->right);
```

```
}
int main()
{
        int pre[] = { 10, 5, 1, 7, 40, 50 };
        int size = sizeof(pre) / sizeof(pre[0]);

        struct node* root = constructTree(pre, size);

        printf("Inorder traversal of the constructed tree: \n");
        printInorder(root);
        return 0;
}
```

Output:



```
1 5 7 10 40 50
```

Practical no-9 Graphs

Q1 & Q2] Implement breadth first search and depth first search for a given graph G=(V,E).

Code:

```
#include<stdio.h>


int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];

int delete();

void add(int item);

void bfs(int s,int n);

void dfs(int s,int n);

void push(int item);

int pop();


void main()

{

int n,i,s,ch,j;

char c,dummy;

printf("ENTER THE NUMBER VERTICES ");

scanf("%d",&n);

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

{

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

{

printf("ENTER b IF %d HAS A NODE WITH %d ELSE 0 ",i,j);

scanf("%c",&a[i][j]);

}

}

printf("THE ADJACENCY MATRIX IS\n");

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

{
```

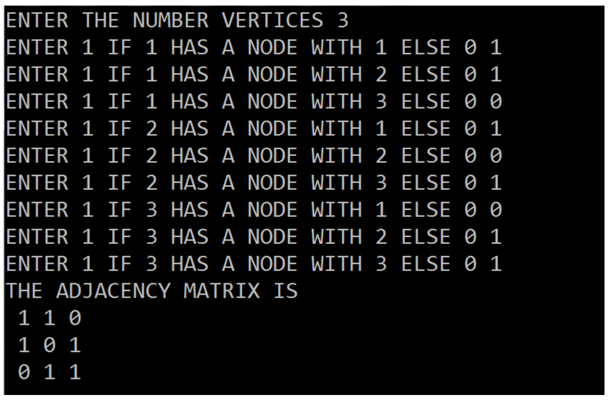
```
for(j=1;j<=n;j++)
{
printf(" %d",a[i][j]);
}
printf("\n");
}

do
{
for(i=1;i<=n;i++)
vis[i]=0;
printf("\nMENU");
printf("\n1.B.F.S");
printf("\n2.D.F.S");
printf("\nENTER YOUR CHOICE");
scanf("%d",&ch);
printf("ENTER THE SOURCE VERTEX :");
scanf("%d",&s);

switch(ch)
{
case 1:bfs(s,n);
break;
case 2:
dfs(s,n);
break;
}
printf("DO U WANT TO CONTINUE(Y/N) ? ");
scanf("%c",&dummy);
scanf("%c",&c);
}while((c=='y')||(c=='Y'));
}
```

```
//BFS(breadth-first search) code
void bfs(int s,int n)
{
int p,i;
add(s);
vis[s]=1;
p=delete();
if(p!=0)
printf(" %d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
if((a[p][i]!=0)&&(vis[i]==0))
{
add(i);
vis[i]=1;
}
p=delete();
if(p!=0)
printf(" %d ",p);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
bfs(i,n);
}
void add(int item)
{
if(rear==19)
printf("QUEUE FULL");
else
{
if(rear==-1)
```

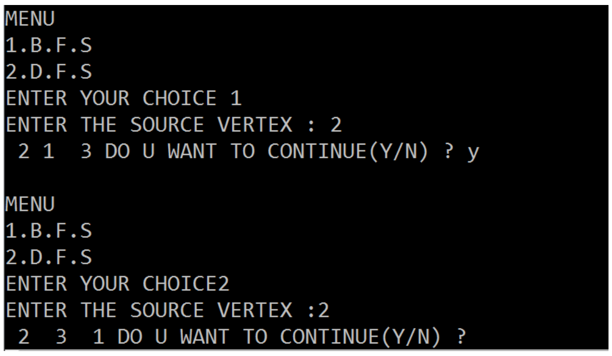
```
{
q[++rear]=item;
front++;
}
else
q[++rear]=item;
}
}
int delete()
{
int k;
if((front>rear)||(front==-1))
return(0);
else
{
k=q[front++];
return(k);
}
}


//DFS(depth-first search) code
void dfs(int s,int n)
{
int i,k;
push(s);
vis[s]=1;
k=pop();
if(k!=0)
printf(" %d ",k);
while(k!=0)
{
```

```
for(i=1;i<=n;i++)
if((a[k][i]!=0)&&(vis[i]==0))
{
push(i);
vis[i]=1;
}
k=pop();
if(k!=0)
printf(" %d ",k);
}
for(i=1;i<=n;i++)
if(vis[i]==0)
dfs(i,n);
}
void push(int item)
{
if(top==19)
printf("Stack overflow ");
else
stack[++top]=item;
}
int pop()
{
int k;
if(top==-1)
return(0);
else
{
k=stack[top--];
return(k);
}
}
```

Output:



```
ENTER THE NUMBER VERTICES 3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 1
THE ADJACENCY MATRIX IS
 1 1 0
 1 0 1
 0 1 1
```



```
MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE 1
ENTER THE SOURCE VERTEX : 2
 2 1  3 DO U WANT TO CONTINUE(Y/N) ? y

MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE2
ENTER THE SOURCE VERTEX :2
 2  3  1 DO U WANT TO CONTINUE(Y/N) ?
```

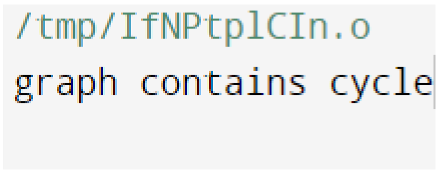
Q3] Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. Perform same task for undirected graph as well.

Code:

```
#include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
#include <stdbool.h>
 struct Edge{
 int src, dest;
 };
 struct Graph{
 int V, E;
 struct Edge* edge;
};
struct Graph* createGraph(int V, int E){
 struct Graph* graph =
 (struct Graph*) malloc( sizeof(struct Graph) );
 graph->V = V;
 graph->E = E;
 graph->edge =
 (struct Edge*) malloc(graph->E*sizeof(struct Edge));
 return graph;
}
int find(int parent[], int i){
 if (parent[i] == -1)
 return i;
 return find(parent, parent[i]);
}
void Union(int parent[], int x, int y){
 int xset = find(parent, x);
 int yset = find(parent, y);
 parent[xset] = yset;
}
int isCycle( struct Graph* graph ){
 int *parent = (int*) malloc( graph->V * sizeof(int) );
 memset(parent, -1, sizeof(int) * graph->V);

 for(int i = 0; i < graph->E; ++i){
 int x = find(parent, graph->edge[i].src);
 int y = find(parent, graph->edge[i].dest);
 if (x == y)
 return true;
 Union(parent, x, y);
 }
 return 0;
}
int main(){
 int V = 3, E = 3;
```

```
struct Graph* graph = createGraph(V, E);
 graph->edge[0].src = 0;
graph->edge[0].dest = 1;
graph->edge[1].src = 1;
graph->edge[1].dest = 2;
graph->edge[2].src = 0;
graph->edge[2].dest = 2;
if (isCycle(graph))
printf( "graph contains cycle" );
else
printf( "graph doesn't contain cycle" );
return 0;
}
```

Output:



```
/tmp/IfNPtplCIn.o
graph contains cycle
```

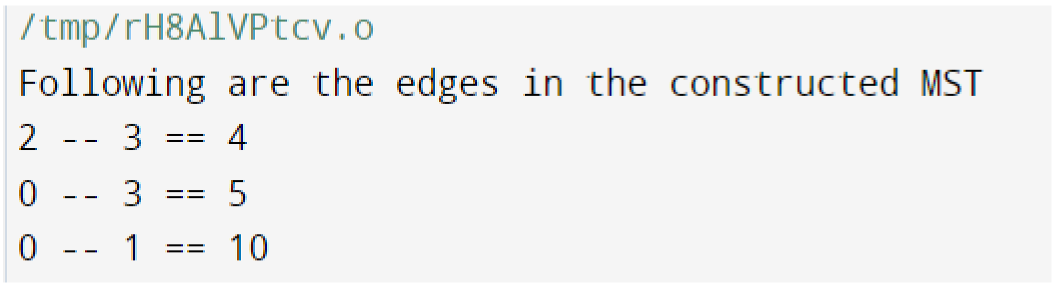
Q4] Implement Minimum Spanning Tree (MST) using the greedy Kruskal’s algorithm.

Code:

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

// a structure to rep a weighted edge in graph
struct Edge
{
	int src, dest, weight;
};
```

```
struct Graph
{
        // V-> Number of vertices, E-> Number of edges
        int V, E;

        struct Edge* edge;
};


struct Graph* createGraph(int V, int E)
{
        struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
        graph->V = V;
        graph->E = E;

        graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

        return graph;
}

struct subset
{
        int parent;
        int rank;
};

int find(struct subset subsets[], int i)
{

        if (subsets[i].parent != i)
                subsets[i].parent = find(subsets, subsets[i].parent);
        return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
        int xroot = find(subsets, x);
        int yroot = find(subsets, y);

        if (subsets[xroot].rank < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
        else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;

        else
        {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++;
        }
}

int myComp(const void* a, const void* b)
{
        struct Edge* a1 = (struct Edge*)a;
        struct Edge* b1 = (struct Edge*)b;
        return a1->weight > b1->weight;
}

void KruskalMST(struct Graph* graph)
{
        int V = graph->V;
        struct Edge result[V]; // This will store the resultant MST
        int e = 0; // An index variable, used for result[]
        int i = 0; // An index variable, used for sorted edges

        qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

        struct subset *subsets =
                (struct subset*) malloc( V * sizeof(struct subset) );

        for (int v = 0; v < V; ++v)
        {
                subsets[v].parent = v;
                subsets[v].rank = 0;
        }

        while (e < V - 1)
        {
                struct Edge next_edge = graph->edge[i++];

                int x = find(subsets, next_edge.src);
                int y = find(subsets, next_edge.dest);

                if (x != y)
                {
                        result[e++] = next_edge;
                        Union(subsets, x, y);
                }

        }

        printf("Following are the edges in the constructed MST\n");
        for (i = 0; i < e; ++i)
                printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                                                        result[i].weight);
        return;
}

// Driver program to test above functions
int main()
{

        int V = 4; // Number of vertices in graph
        int E = 5; // Number of edges in graph
        struct Graph* graph = createGraph(V, E);


        // add edge 0-1
        graph->edge[0].src = 0;
        graph->edge[0].dest = 1;
        graph->edge[0].weight = 10;

        // add edge 0-2
        graph->edge[1].src = 0;
        graph->edge[1].dest = 2;
        graph->edge[1].weight = 6;

        // add edge 0-3
        graph->edge[2].src = 0;
        graph->edge[2].dest = 3;
        graph->edge[2].weight = 5;

        // add edge 1-3
        graph->edge[3].src = 1;
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 15;
        // add edge 2-3
        graph->edge[4].src = 2;
        graph->edge[4].dest = 3;
        graph->edge[4].weight = 4;

        KruskalMST(graph);

        return 0;
}
```

Output:



```
/tmp/rH8AlVPtcv.o
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
```

Practical No-10 Hashing

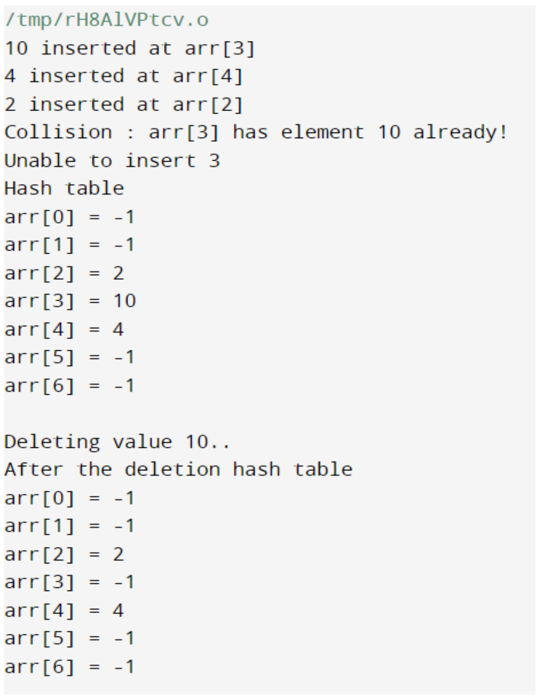
Q1] Given a limited range array containing both positive and non-positive numbers, i.e., elements are in the range from -MAX to +MAX. Our task is to search if some number is present in the array or not in O(1) time.

Code:

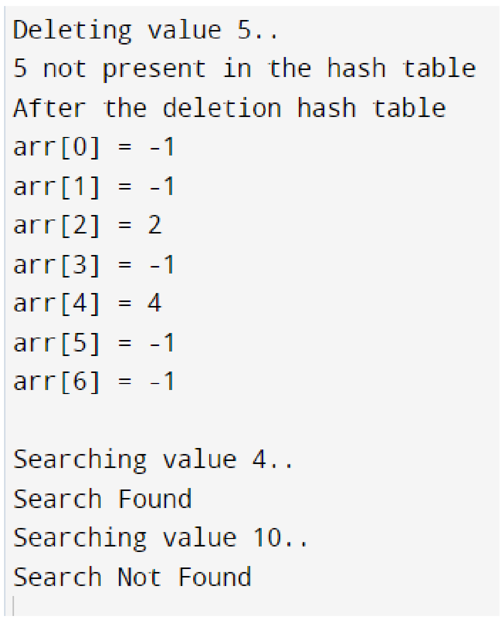
```
#include<stdio.h>
#define size 7
int arr[size];
void init()
{
    int i;
    for(i = 0; i < size; i++)
        arr[i] = -1;
}
void insert(int value)
{
    int key = value % size;
    if(arr[key] == -1)
    {
        arr[key] = value;
        printf("%d inserted at arr[%d]\n", value,key);
    }
    else
    {
        printf("Collision : arr[%d] has element %d already!\n",key,arr[key]);
        printf("Unable to insert %d\n",value);
    }
}
```

```
void del(int value)
{
    int key = value % size;
    if(arr[key] == value)
        arr[key] = -1;
    else
        printf("%d not present in the hash table\n",value);
}

void search(int value)
{
    int key = value % size;
    if(arr[key] == value)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
}

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

int main()
{
    init();
    insert(10); //key = 10 % 7 ==> 3
    insert(4);  //key = 4 % 7  ==> 4
    insert(2);  //key = 2 % 7  ==> 2
    insert(3);  //key = 3 % 7  ==> 3 (collision)

    printf("Hash table\n");
    print();
    printf("\n");

    printf("Deleting value 10..\n");
    del(10);
    printf("After the deletion hash table\n");
    print();
    printf("\n");

    printf("Deleting value 5..\n");
    del(5);
    printf("After the deletion hash table\n");
    print();
    printf("\n");

    printf("Searching value 4..\n");
    search(4);
    printf("Searching value 10..\n");
    search(10);

    return 0;
}
```

Output:



```
/tmp/rH8AlVPtcv.o
10 inserted at arr[3]
4 inserted at arr[4]
2 inserted at arr[2]
Collision : arr[3] has element 10 already!
Unable to insert 3
Hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = 10
arr[4] = 4
arr[5] = -1
arr[6] = -1

Deleting value 10..
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1
```



```
Deleting value 5..
5 not present in the hash table
After the deletion hash table
arr[0] = -1
arr[1] = -1
arr[2] = 2
arr[3] = -1
arr[4] = 4
arr[5] = -1
arr[6] = -1

Searching value 4..
Search Found
Searching value 10..
Search Not Found
```

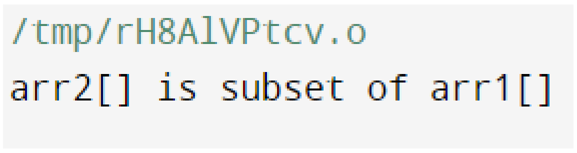
Q2] Given two arrays: A and B. Find whether B is a subset of A or not using Hashing. Both the arrays are not in sorted order. It may be assumed that elements in both arrays are distinct.

Code:

```
#include <stdio.h>


int isSubset(int arr1[], int arr2[], int m, int n)
```

```
{
    int i = 0;
    int j = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++) {
            if (arr2[i] == arr1[j])
                break;
        }

        if (j == m)
            return 0;
    }

    return 1;
}

int main()
{
    int arr1[] = { 11, 1, 13, 21, 3, 7 };
    int arr2[] = { 11, 3, 7, 1 };

    int m = sizeof(arr1) / sizeof(arr1[0]);
    int n = sizeof(arr2) / sizeof(arr2[0]);

    if (isSubset(arr1, arr2, m, n))
        printf("arr2[] is subset of arr1[] ");
    else
        printf("arr2[] is not a subset of arr1[]");
    getchar();

    return 0;

}
```

Output:



```
/tmp/rH8AlVPtcv.o
arr2[] is subset of arr1[]
```