Q.1. Implementation of structures

```
#include <stdio.h>
struct student{
   char name[100];
   int rollNo;
   int marks[5];
   int avg;
};
int main(){
   struct student stu;
   printf("Enter the student name: ");
   scanf("%s",stu.name);
   printf("Enter the student roll no.: ");
   scanf("%d",&stu.rollNo);
   stu.avg=0;
   printf("Enter the student marks: ");
   for(int i=0; i<5;i++){
      scanf("%d",&stu.marks[i]);
      stu.avg+=stu.marks[i];
   }
   stu.avg/=5;
   printf("Student name: %s\n",stu.name);
   printf("Roll No.: %d\n", stu.rollNo);
   printf("Avg marks: %d\n", stu.avg);
   if(stu.avg>50)
      printf("Pass\n");
   else
      printf("Fail\n");
}
```

Q.2. Implementation of structures using pointers

Code for employee:-

```
#include <stdio.h>
#include <stdlib.h>
struct employee{
        char name[50];
        int empID;
        char dept[50];
        float salary;
};
int main(){
        struct employee* emp = (struct employee*) malloc(2 * sizeof(struct employee));
        for(int i=0; i<2;i++){
                printf("Enter the name: ");
```

```
            fgets(emp[i].name, 50, stdin);
            emp[i].empID=i+1;
            printf("Enter the dept: ");
            fgets(emp[i].dept, 50, stdin);
            printf("Enter the salary: ");
            scanf("%f",&emp[i].salary);
            getchar();
        }
        for(int i=0; i<2;i++){
            printf("Employee %d-\n", emp[i].empID);
            printf("Name: %s", emp[i].name);
            printf("Dept: %s", emp[i].dept);
            printf("Salary: %.2f\n", emp[i].salary);
        }
        free(emp);
        return 0;
}
```

Code for student:-

```
#include <stdio.h>
struct student{
   char name[100];
   int rollNo;
   struct student *link;
};
int main(){
   struct student stu1;
   printf("Enter the student name: ");
   scanf("%s",stu1.name);
   printf("Enter the student roll no.: ");
   scanf("%d",&stu1.rollNo);
   stu1.link=&stu1;
   printf("Student name: %s\n", stu1.link->name);
   printf("Roll No.: %d\n", stu1.link->rollNo);
}
```

Q.3. Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
int main(){
   int **p,**q,**A,r1,c1,r2,c2,i,j;
   printf("Enter the row and column of the first matrix: ");
   scanf("%d%d",&r1,&c1);
   p=(int**) malloc(sizeof(int*)*r1);
```

```
    for(i=0;i<r1;i++){
        *(p+i)=(int*) malloc(sizeof(int)*c1);
    }
    printf("enter the values: ");
    for(i=0;i<r1;i++){
        for(j=0;j<c1;j++){
            scanf("%d",*(p+i)+j);
        }
    }
    printf("Enter the row and column of the second matrix: ");
    scanf("%d%d",&r2,&c2);
    q=(int**) malloc(sizeof(int*)*r2);
    for(i=0;i<r2;i++){
        *(q+i)=(int*)malloc(sizeof(int)*c2);
    }
    printf("enter the values: ");
    for(i=0;i<r2;i++){
        for(j=0;j<c2;j++){
            scanf("%d",*(q+i)+j);
        }
    }
    A=(int**) malloc(sizeof(int*)*r1);
    for(i=0;i<r1;i++){
        *(A+i)=(int*) malloc(sizeof(int)*c2);
    }
    for(i=0;i<r1;i++){
        for(j=0;j<c2;j++){
            *(*(A+i)+j)=0;
            for(int k=0;k<c2;k++){
                *(*(A+i)+j)+= *(*(p+i)+k) * (*(*(q+k)+j));
            }
        }
    }
    printf("Resultant matrix: \n");
    for(i=0;i<r1;i++){
        for(j=0;j<c2;j++){
            printf("%d\t",*(*(A+i)+j));
        }
        putchar('\n');
    }
    return 0;
}
```

Q.4 Array implementation of list

```
#include <stdio.h>
int main(){
    int arr[100], n, i, pos, ele;
    printf("Enter the size of the array: ");
    scanf("%d", &n);
    printf("Enter the elements of the array: ");
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);
    printf("Enter the position where you want to insert: ");
    scanf("%d", &pos);
    printf("Enter the element you want to insert: ");
    scanf("%d", &ele);
    for(i=n-1; i>=pos-1; i--)
        arr[i+1] = arr[i];
    arr[pos-1] = ele;
    printf("The array after insertion is: ");
    for(i=0; i<=n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Q.5. Implementation of linked list

```
#include <stdio.h>
#include <stdlib.h>
//linked list node
struct node{
    int data;
    struct node *next;
};
//inserting at the beginning
void insertAtBeginning(struct node **head, int data){
    struct node *newNode = (struct node*) malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
//inserting at the end
void insertAtEnd(struct node **head, int data){
    struct node *newNode = (struct node*) malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = NULL;
    if(*head==NULL){
        *head = newNode;
```

```
        return;
    }
    struct node *last = *head;
    while(last->next!=NULL){
        last = last->next;
    }
    last->next = newNode;
}
//inserting at a given position
void insertAtPosition(struct node **head, int data, int position){
    struct node *newNode = (struct node*) malloc(sizeof(struct node));
    newNode->data = data;
    if(position==1){
        newNode->next = *head;
        *head = newNode;
        return;
    }
    struct node *temp = *head;
    for(int i=1;i<position-1;i++){
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
//deleting a node
void deleteNode(struct node **head, int position){
    struct node *temp = *head, *del;
    if(position==1){
        *head = temp->next;
        free(temp);
        return;
    }
    for(int i=1;i<position-1;i++){
        temp = temp->next;
    }
    del = temp->next;
    temp->next = del->next;
    free(del);
}
//printing the list
void printList(struct node *head){
    while(head!=NULL){
        printf("%d ",head->data);
        head = head->next;
```

```
    }
    printf("\n");
}
int main(){
    struct node *head = NULL;
    int choice, data, position;
    while(1){
        printf("1. Insert at beginning\n2. Insert at end\n3. Insert at position\n4. Delete node\n5.
Print list\n6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
                printf("Enter data: ");
                scanf("%d",&data);
                insertAtBeginning(&head, data);
                break;
            case 2:
                printf("Enter data: ");
                scanf("%d",&data);
                insertAtEnd(&head, data);
                break;
            case 3:
                printf("Enter data: ");
                scanf("%d",&data);
                printf("Enter position: ");
                scanf("%d",&position);
                insertAtPosition(&head, data, position);
                break;
            case 4:
                printf("Enter position: ");
                scanf("%d",&position);
                deleteNode(&head, position);
                break;
            case 5:
                printList(head);
                break;
            case 6:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
```

```
}
```

Q.6. Implementation of doubly linked list

```
#include <stdio.h>
#include <stdlib.h>
struct node{
   int data;
   struct node *next;
   struct node *prev;
};
void insertAtBeginning(struct node **head, int data){
   struct node *newNode = (struct node*) malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = *head;
   newNode->prev = NULL;
   if((*head)!=NULL){
      (*head)->prev = newNode;
   }
   (*head) = newNode;
}
void insertAtEnd(struct node **head, int data){
   struct node *newNode=(struct node*) malloc(sizeof(struct node));
   newNode->data=data;
   newNode->next=NULL;
   if((*head)==NULL){
      newNode->prev=NULL;
      (*head)=newNode;
   }
   struct node *last = *head;
   while(last->next!=NULL){
      last=last->next;
   }
   newNode->prev=last;
   last->next=newNode;
}
void insertAtPosition(struct node **head, int data, int position){
   struct node *newNode=(struct node*) malloc(sizeof(struct node));
   newNode->data=data;
   if(position==1){
      (*head)->prev=newNode;
      newNode->prev=NULL;
      newNode->next=*head;
      *head=newNode;
      return;
```

```
    }
    struct node *temp=*head;
    for(int i=1;i<position-1;i++){
        temp=temp->next;
    }
    newNode->next=temp->next;
    newNode->prev=temp;
    temp->next->prev=newNode;
    temp->next=newNode;
}
void deleteNode(struct node **head, int position){
    if(position==1){
        (*head)=(*head)->next;
        free((*head)->prev);
        (*head)->prev=NULL;
        return;
    }
    struct node *previous=NULL, *temp=*head;
    for(int i=1;i<position;i++){
        previous=temp;
        temp=temp->next;
    }
    previous->next=temp->next;
    temp->next->prev=previous;
    free(temp);
}
void printList(struct node *head){
    struct node* temp=head;
    while(temp!=NULL){
        printf("%d ",temp->data);
        temp=temp->next;
    }
    printf("\n");
}
int main(){
    struct node *head = NULL;
    int choice, data, position;
    while(1){
        printf("1. Insert at beginning\n2. Insert at end\n3. Insert at position\n4. Delete node\n5.
Print list\n6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1:
```

```
            printf("Enter data: ");
            scanf("%d",&data);
            insertAtBeginning(&head, data);
            break;
        case 2:
            printf("Enter data: ");
            scanf("%d",&data);
            insertAtEnd(&head, data);
            break;
        case 3:
            printf("Enter data: ");
            scanf("%d",&data);
            printf("Enter position: ");
            scanf("%d",&position);
            insertAtPosition(&head, data, position);
            break;
        case 4:
            printf("Enter position: ");
            scanf("%d",&position);
            deleteNode(&head, position);
            break;
        case 5:
            printList(head);
            break;
        case 6:
            exit(0);
        default:
            printf("Invalid choice\n");
      }
   }
   return 0;
}
```

Q.7. Implementation of stack

Stack through array:-

```
#include <stdio.h>
int top=-1,size=5;
void push(int arr[], int x){
   if(top==size-1){
      printf("Stack overflow");
   }
   else{
      top++;
      arr[top]=x;
```

```
    }
}
void display(int arr[]){
    printf("Stack: ");
    for(int i=0;i<=top;i++){
        printf("%d ",arr[i]);
    }
    putchar('\n');
}
void pop(int arr[]){
    if(top==-1){
        printf("Stack underflow");
        return;
    }
    arr[top]=0;
    top--;
}
void peek(int arr[]){
    if(top!=-1)
        printf("Top element: %d",arr[top]);
}
int main(){
    int choice;
    int arr[size];
    while(1){
        printf("\n1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: {
                int x;
                printf("Enter element to push: ");
                scanf("%d",&x);
                push(arr,x);
                break;
            }
            case 2: {
                pop(arr);
                break;
            }
            case 3: {
                peek(arr);
                break;
            }
            case 4: {
```

```
            display(arr);
            break;
        }
        case 5: {
            return 0;
        }
        default: {
            printf("Invalid choice");
        }
    }
  }
}
```

Stack through linked list

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

struct node{
   int data;
   struct node *next;
}*top=NULL;

void push(int x){
   struct node* newNode=(struct node*) malloc(sizeof(struct node));
   newNode->data=x;
   newNode->next=top;
   top=newNode;
}
void pop(){
   if(top==NULL){
      printf("Stack underflow");
   }
   else{ printf("Removing %d\n",top->data);
      struct node *temp=top;
      top=top->next;
      free(temp);
   }
}
void display(){
   if(top==NULL)
      printf("Stack empty");
   else{printf("Stack: ");
      struct node* temp=top;
      while(temp!=NULL){
```

```
            printf("%d ",temp->data);
            temp=temp->next;
        }
        putchar('\n');
    }
}
void peek(){
    if(top!=NULL){printf("Top element: ");
        printf("%d\n",top->data);
    }
}
int main(){
    int choice,x;
    printf("1. Push\n2. Pop\n3. Display\n4. Peek\n0. Exit\n");
    while(choice!=0){
        printf("Enter your choice: ");
        scanf("%d",&choice);
        switch(choice){
            case 1: printf("Enter data: ");
                    scanf("%d",&x);
                    push(x);
                    break;
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: peek();
                    break;
            case 0: exit(1);
                    break;
        }
    }
}
```

Q.8. Implementation of queue

Using array

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

int queue[N];
int front=-1, rear=-1;
```

```
void enqueue(int x){
    if(rear==N-1){
        printf("Queue full");
    }
    else if(front==-1){ // Or, (front == -1 && rear == -1)
        front += 1;
        rear += 1;
        queue[rear]=x;
    }
    else{
        rear+=1;
        queue[rear]=x;
    }
}

void dequeue(){
    if(front==-1){
        printf("queue empty");
    }
    else if(front==rear || front > rear){
        front=rear=-1;
    }
    else{
        front+=1;
    }
}

void display(){
    if(front==-1){
        printf("queue empty");
    }
    else{
        for(int i=front; i<=rear; i++){
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}
int main(){
    int choice, x;
    printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
    while(1){
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch(choice){
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice");
        }
    }
}
```

Using linked list

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

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

void enqueue(int x){
    struct node* newNode= (struct node*) malloc(sizeof(struct node));
    newNode->data=x;
    newNode->next=NULL;
    if(front==NULL){
        rear=newNode;
        front=newNode;
    }
    else{
        rear->next=newNode;
        rear=newNode;
    }
}
void dequeue(){
    if(front==NULL){
```

```
        printf("Queue empty");
    }
    else if(front==rear){
        free(front);
        front=rear=NULL;
    }
    else{
        struct node* temp=front;
        front=front->next;
        free(temp);
    }
}
void display(){
    if(front==NULL){
        printf("Queue empty");
        return;
    }
    struct node* temp= front;
    while(temp!=NULL){
        printf("%d ", temp->data);
        temp=temp->next;
    }
    printf("\n");
}
int main(){
    int choice, x;
    printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
    while(1){
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice){
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
```

```
            default:
                printf("Invalid choice");
        }
    }
    return 0;
}
```

Q.9. Application of stack and queue
Circular queue

```
//circular queue using array
#include<stdio.h>
#define MAX 5
int queue[5], front=-1, rear=-1;

void enqueue(int x)
{
    if((front==0 && rear==MAX-1) || (front==rear+1))
    {
        printf("Queue Overflow\n");
        return;
    }
    if(front==-1)
    {
        front=0;
        rear=0;
    }
    else
    {
        if(rear==MAX-1)
            rear=0;
        else
            rear=rear+1;
    }
    queue[rear]=x;
}

void dequeue()
{
    if(front==-1)
    {
        printf("Queue Underflow\n");
        return;
    }
    printf("Element deleted from queue is : %d\n", queue[front]);
```

```
    if(front==rear)
    {
        front=-1;
        rear=-1;
    }
    else
    {
        if(front==MAX-1)
            front=0;
        else
            front=front+1;
    }
}

void display()
{
    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 ", queue[front_pos]);
            front_pos++;
        }
    }
    else
    {
        while(front_pos<=MAX-1)
        {
            printf("%d ", queue[front_pos]);
            front_pos++;
        }
        front_pos=0;
        while(front_pos<=rear_pos)
        {
            printf("%d ", queue[front_pos]);
            front_pos++;
        }
```

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

int main()
{
    int choice, x;
    printf("1.Enqueue\n");
    printf("2.Dequeue\n");
    printf("3.Display\n");
    printf("4.Exit\n");
    do
    {
        printf("Enter choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter value: ");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                break;
            default:
                printf("Wrong choice\n");
        }
    }while(choice!=4);
    return 0;
}

/*
PSEUDOCODE:

# Circular queue using array

# Define the queue size and queue
MAX = 5
```

```
queue[MAX]

# Define the front and rear pointers
front = -1
rear = -1

# Function to enqueue an element into the queue
def enqueue(x):
  # Check if the queue is full
  if (front == 0 and rear == MAX - 1) or (front == rear + 1):
    print("Queue Overflow")
    return

  # If the queue is empty, initialize the front and rear pointers
  if front == -1:
    front = 0
    rear = 0

  # Otherwise, increment the rear pointer
  else:
    if rear == MAX - 1:
      rear = 0
    else:
      rear += 1

  # Add the element to the queue
  queue[rear] = x

# Function to dequeue an element from the queue
def dequeue():
  # Check if the queue is empty
  if front == -1:
    print("Queue Underflow")
    return

  # Print the element that is being deleted
  print("Element deleted from queue is: ", queue[front])

  # If the front and rear pointers are equal, the queue is empty
  if front == rear:
    front = -1
    rear = -1

  # Otherwise, increment the front pointer
```

```
  else:
    if front == MAX - 1:
      front = 0
    else:
      front += 1

# Function to display the elements of the queue
def display():
  # If the queue is empty, print a message
  if front == -1:
    print("Queue is empty")
    return

  # Define the front and rear positions
  front_pos = front
  rear_pos = rear

  # Print the elements of the queue
  if front_pos <= rear_pos:
    while front_pos <= rear_pos:
      print(queue[front_pos], " ")
      front_pos += 1
  else:
    while front_pos <= MAX - 1:
      print(queue[front_pos], " ")
      front_pos += 1
    front_pos = 0
    while front_pos <= rear_pos:
      print(queue[front_pos], " ")
      front_pos += 1

*/
```

INFIX POSTFIX

```
//infix to postfix
#include<stdio.h>
#include<ctype.h>
#define MAX 20

char stack[MAX];
char ans[MAX];
int top = -1;

void push(char x)
{
```

```
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else{
        return stack[top--];
    }
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    if(x == '^')
        return 3;
}

int main()
{
    char exp[MAX];
    char *e, x;
    printf("Enter the expression: ");
    scanf("%s", exp);
    e = exp;
    int i = 0;
    while(*e != '\0')
    {
        if(isalnum(*e))
            ans[i++] = *e;
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                ans[i++] = x;
        }
        else{
```

```
            while(priority(stack[top]) >= priority(*e))
                ans[i++] = pop();
            push(*e);
        }
        e++;
    }
    while(top != -1)
    {
        ans[i++] = pop();
    }
    printf("Postfix expression: %s\n", ans);
    return 0;
}

/*
OUTPUT:
Enter the expression: a+b*(c-d)
Postfix expression: abcd-*+

PSEUDOCODE:

define max 20 and the stack and answer arrays of max size
define top as -1

define push function
    increment top
    add x to stack[top]

define pop function
    if top is -1
        return -1
    else
        return stack[top--]

define priority function
    if x is '('
        return 0
    if x is '+' or '-'
        return 1
    if x is '*' or '/'
        return 2
    if x is '^'
        return 3
```

```
define main function
    define exp array of max size
    define e and x as char pointer and char
    take input in exp
    set pointer e to exp
    define i as 0
    while e is not null
        if e is alphanumeric
            add e to ans[i]
            increment i
        else if e is '('
            push e
        else if e is ')'
            while pop() is not '('
                add pop() to ans[i]
                increment i
        else
            while priority(stack[top]) >= priority(e)
                add pop() to ans[i]
                increment i
            push e
        increment e

    while top is not -1
        add pop() to ans[i]
        increment i
    print ans

*/
```

TOWERS OF HANOI

```
#include <stdio.h>

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
  if (n == 1) {
    printf("Move disk 1 from rod %c to rod %c\n", from_rod, to_rod);
    return;
  }

  towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
  printf("Move disk %d from rod %c to rod %c\n", n, from_rod, to_rod);
  towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
```

```
  int n;
  scanf("%d",&n);
  char from_rod = 'A', to_rod = 'C', aux_rod = 'B';

  towerOfHanoi(n, from_rod, to_rod, aux_rod);

  return 0;
}
```

Q.10. Implementation of tree ussing array

```
#include <stdio.h>

int tree[15];

void insert(int x){
    for(int i=0;i<15;i++){
        if(tree[i]=='\0'){
            tree[i]=x;
            return;
        }
    }
    printf("\nTree is full");
}

void print_tree() {
    printf("\n");
    for (int i = 0; i < 15; i++) {
        if (tree[i] != '\0')
            printf("%d ", tree[i]);
        else
            printf("0 ");
    }
}

void delete(int pos){
    if(tree[pos]=='\0'){
        printf("\nEmpty node");
        return;
    }
    tree[pos]='\0';
    if(tree[2*pos+1]!='\0'){
        delete(2*pos+1);
    }
```

```
    if(tree[2*pos+2]!='\0'){
        delete(2*pos+2);
    }
}

void search(int x){
    for(int i=0;i<15;i++){
        if(tree[i]==x){
            printf("\nFound");
            return;
        }
    }
    printf("\nNot found");
}

int main() {
    int choice;
    int x,parent;
    printf("\n1. Insert\n2. Search Element\n3. Delete\n4. Print tree\n0. Exit\n");
    do {
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter value to be inserted: ");
                scanf(" %d", &x);
                insert(x);
                break;
            case 2:
                printf("\nEnter value to be searched: ");
                scanf(" %d", &x);
                search(x);
                break;
            case 3:
                printf("Enter the position to delete: ");
                scanf("%d", &parent);
                delete(parent);
                break;
            case 4:
                print_tree();
                break;
            case 0:
                printf("\nExiting...");
                break;
```

```
            default:
                printf("\nInvalid choice");
        }
    } while (choice != 0);
    return 0;
}
```

BINARY TREE

```
#include <stdio.h>

int tree[15]; //tree of height 3

void root(int x) {
    tree[0] = x;
}

void left_set(int x, int parent) {
    if (tree[parent] == '\0')
        printf("\nCan't set child at %d, no parent found", (parent * 2) + 1);
    else
        tree[(parent * 2) + 1] = x;
}

void right_set(int x, int parent) {
    if (tree[parent] == '\0')
        printf("\nCan't set child at %d, no parent found", (parent * 2) + 2);
    else
        tree[(parent * 2) + 2] = x;
}

void print_tree() {
    printf("\n");
    for (int i = 0; i < 15; i++) {
        if (tree[i] != '\0')
            printf("%d ", tree[i]);
        else
            printf("0 ");
    }
}

void search(int x){
    for(int i=0;i<15;i++){
        if(tree[i]==x){
```

```
            printf("\nFound");
            return;
        }
    }
    printf("\nNot found");
}

int main() {
    int choice;
    int x,parent;
    printf("\nEnter root value: ");
    scanf("%d", &x);
    root(x);
    printf("\n1. Insert left child\n2. Insert right child\n3. Search Element\n4. Print tree\n0. Exit\n");
    do {
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("\nEnter value to be inserted: ");
                scanf(" %d", &x);
                printf("\nEnter the parent: ");
                scanf("%d", &parent);
                left_set(x, parent);
                break;
            case 2:
                printf("\nEnter value to be inserted: ");
                scanf(" %d", &x);
                printf("\nEnter the parent: ");
                scanf("%d", &parent);
                right_set(x, parent);
                break;
            case 3:
                printf("\nEnter value to be searched: ");
                scanf(" %d", &x);
                search(x);
                break;
            case 4:
                print_tree();
                break;
            case 0:
                break;
            default:
                printf("\nInvalid Choice");
```

```
        }
    } while (choice != 0);
}
```

Q.11. Binary Search Tree (BST) using linked list

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

struct node {
  int data;
  struct node *left;
  struct node *right;
}*root=NULL;

void insert(int data){
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    if(root == NULL){
        root = newNode;
        return;
    }
    struct node *temp = root;
    while(1){
        if(data < temp->data){
            if(temp->left == NULL){
                temp->left = newNode;
                return;
            }
            temp = temp->left;
        }
        else{
            if(temp->right == NULL){
                temp->right = newNode;
                return;
            }
            temp = temp->right;
        }
    }
}

void search(int data){
```

```
    struct node *temp = root;
    while(temp != NULL){
        if(data == temp->data){
            printf("Found\n");
            return;
        }
        else if(data < temp->data){
            temp = temp->left;
        }
        else{
            temp = temp->right;
        }
    }
    printf("Not found\n");
}

void delete(int data){
    struct node *temp = root, *parent = NULL;
    while(temp != NULL){
        if(data == temp->data){
            break;
        }
        else if(data < temp->data){
            parent = temp;
            temp = temp->left;
        }
        else{
            parent = temp;
            temp = temp->right;
        }
    }
    if(temp == NULL){
        printf("Not found\n");
        return;
    }
    if(temp->left == NULL && temp->right == NULL){
        //if the data is root node
        if(parent == NULL){
            root = NULL;
        }
        else if(parent->left == temp){
            parent->left = NULL;
        }
        else{
```

```
            parent->right = NULL;
        }
        free(temp);
    }
    else if(temp->left == NULL){
        //if the node is root node, set right child as root
        if(parent == NULL){
            root = temp->right;
        }
        //if the data is left child of parent, set right child of data as left child of parent
        else if(parent->left == temp){
            parent->left = temp->right;
        }
        //if the data is right child of parent, set right child of data as right child of parent
        else{
            parent->right = temp->right;
        }
        free(temp);
    }
    else if(temp->right == NULL){
        //if the node is root node, set left child as root
        if(parent == NULL){
            root = temp->left;
        }
        //if the data is left child of parent, set left child of data as left child of parent
        else if(parent->left == temp){
            parent->left = temp->left;
        }
        //if the data is right child of parent, set left child of data as right child of parent
        else{
            parent->right = temp->left;
        }
        free(temp);
    }
    else{
        struct node *successor = temp->right, *parentSuccessor = temp;
        //search for lowest value in right subtree
        while(successor->left != NULL){
            parentSuccessor = successor;
            successor = successor->left;
        }
        //copy the data of successor to temp and delete successor
        temp->data = successor->data;
        if(parentSuccessor->left == successor){
```

```
            parentSuccessor->left = successor->right;
        }
        else{
            parentSuccessor->right = successor->right;
        }
        free(successor);
    }
}

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

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

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

void display(){
    //display the tree with spacings like a tree
}

int main(){
    int choice, data;
    printf("\n1. Insert\n2. Search\n3. Delete\n4. Display\n5. Exit\n");
    while(1){
```

```
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice){
            case 1:
                printf("Enter data: ");
                scanf("%d", &data);
                insert(data);
                break;
            case 2:
                printf("Enter data: ");
                scanf("%d", &data);
                search(data);
                break;
            case 3:
                printf("Enter data: ");
                scanf("%d", &data);
                delete(data);
                break;
            case 4:
                display();
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice\n");
        }
    }
    return 0;
}
```

Q.12. Btrees

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

struct node {
    int n;
    int* keys;
    struct node** p;
};

int M; // Order of the B-tree

enum KeyStatus { Duplicate, SearchFailure, Success, InsertIt, LessKeys };
```

```
struct node* root = NULL;

void insert(int key);
void display(struct node* root, int);
void DelNode(int x);
void search(int x);
enum KeyStatus ins(struct node* r, int x, int* y, struct node** u);
int searchPos(int x, int* key_arr, int n);
enum KeyStatus del(struct node* r, int x);

int main() {
   int key;
   int choice;

   // Prompt the user for the order of the B-tree
   printf("Enter the order of the B-tree: ");
   scanf("%d", &M);
   printf("Creation of B tree for node %d\n", M);

   // Initialize the root node
   root = (struct node*)malloc(sizeof(struct node));
   root->n = 0;
   root->keys = (int*)malloc((M - 1) * sizeof(int));
   root->p = (struct node*)malloc(M * sizeof(struct node));

   for (int i = 0; i < M - 1; i++) {
      root->keys[i] = 0;
      root->p[i] = NULL;
   }
   root->p[M - 1] = NULL;

   while (1) {
      printf("1.Insert\n");
      printf("2.Delete\n");
      printf("3.Search\n");
      printf("4.Display\n");
      printf("5.Quit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter the key: ");
            scanf("%d", &key);
```

```
            insert(key);
            break;
        case 2:
            printf("Enter the key: ");
            scanf("%d", &key);
            DelNode(key);
            break;
        case 3:
            printf("Enter the key: ");
            scanf("%d", &key);
            search(key);
            break;
        case 4:
            printf("Btree is:\n");
            display(root, 0);
            break;
        case 5:
        exit(1);
        default:
            printf("Wrong choice\n");
            break;
      }
   }

   return 0;
}


void insert(int key) {
   struct node* newnode;
   int upKey;
   enum KeyStatus value;
   value = ins(root, key, &upKey, &newnode);
   if (value == Duplicate)
      printf("Key already available\n");
   if (value == InsertIt) {
      struct node* uproot = root;
      root = malloc(sizeof(struct node));
      root->n = 1;
      root->keys[0] = upKey;
      root->p[0] = uproot;
      root->p[1] = newnode;
   }
}
```

```
enum KeyStatus ins(struct node* ptr, int key, int* upKey, struct node** newnode) {
    int pos, i, n, splitPos;
    int newKey, lastKey;
    enum KeyStatus value;

    if (ptr == NULL) {
        *newnode = NULL;
        *upKey = key;
        return InsertIt;
    }

    n = ptr->n;
    pos = searchPos(key, ptr->keys, n);

    if (pos < n && key == ptr->keys[pos])
        return Duplicate;

    value = ins(ptr->p[pos], key, &newKey, newnode);
    if (value != InsertIt)
        return value;

    if (n < M - 1) {
        pos = searchPos(newKey, ptr->keys, n);

        for (i = n; i > pos; i--) {
            ptr->keys[i] = ptr->keys[i - 1];
            ptr->p[i + 1] = ptr->p[i];
        }

        ptr->keys[pos] = newKey;
        ptr->p[pos + 1] = *newnode;
        ++ptr->n;

        return Success;
    }

    if (pos == M - 1) {
        lastKey = newKey;
    } else {
        lastKey = ptr->keys[M - 2];
        for (i = M - 2; i > pos; i--) {
            ptr->keys[i] = ptr->keys[i - 1];
            ptr->p[i + 1] = ptr->p[i];
```

```
        }

        ptr->keys[pos] = newKey;
        ptr->p[pos + 1] = *newnode;
    }

    splitPos = (M - 1) / 2;
    (*upKey) = ptr->keys[splitPos];

    (*newnode) = malloc(sizeof(struct node));
    ptr->n = splitPos;
    (*newnode)->n = M - 1 - splitPos;

    for (i = 0; i < (*newnode)->n; i++) {
        (*newnode)->p[i] = ptr->p[i + splitPos + 1];
        if (i < (*newnode)->n - 1)
            (*newnode)->keys[i] = ptr->keys[i + splitPos + 1];
        else
            (*newnode)->keys[i] = lastKey;
    }

    (*newnode)->p[(*newnode)->n] = ptr->p[ptr->n + 1];

    return InsertIt;
}

void display(struct node* ptr, int blanks) {
    int i;

    if (ptr) {
        for (i = 1; i <= blanks; i++)
            printf(" ");

        for (i = 0; i < ptr->n; i++)
            printf("%d ", ptr->keys[i]);

        printf("\n");

        for (i = 0; i <= ptr->n; i++)
            display(ptr->p[i], blanks + 10);
    }
}

void search(int key) {
```

```
    int pos, i, n;
    struct node* ptr = root;
    printf("Search path:\n");

    while (ptr) {
        n = ptr->n;
        for (i = 0; i < ptr->n; i++)
            printf(" %d", ptr->keys[i]);

        printf("\n");
        pos = searchPos(key, ptr->keys, n);

        if (pos < n && key == ptr->keys[pos]) {
            printf("Key %d found in position %d of the last displayed node\n", key, i);
            return;
        }

        ptr = ptr->p[pos];
    }

    printf("Key %d is not available\n", key);
}

int searchPos(int key, int* key_arr, int n) {
    int pos = 0;
    while (pos < n && key > key_arr[pos])
        pos++;
    return pos;
}

void DelNode(int key) {
    struct node* uproot;
    enum KeyStatus value;
    value = del(root, key);

    switch (value) {
        case SearchFailure:
            printf("Key %d is not available\n", key);
            break;
        case LessKeys:
            uproot = root;
            root = root->p[0];
            free(uproot);
            break;
```

```
    }
}

enum KeyStatus del(struct node* ptr, int key) {
    int pos, i, pivot, n, min;
    int* key_arr;
    enum KeyStatus value;
    struct node** p, * lptr, * rptr;

    if (ptr == NULL)
        return SearchFailure;

    n = ptr->n;
    key_arr = ptr->keys;
    p = ptr->p;
    min = (M - 1) / 2;

    pos = searchPos(key, key_arr, n);

    if (p[0] == NULL) {
        if (pos == n || key < key_arr[pos]) {
            return SearchFailure;
        }

        for (i = pos + 1; i < n; i++) {
            key_arr[i - 1] = key_arr[i];
            p[i] = p[i + 1];
        }

        return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
    }

    if (pos < n && key == key_arr[pos]) {
        struct node* qp = p[pos], * qp1;
        int nkey;
        while (1) {
            nkey = qp->n;
            qp1 = qp->p[nkey];
            if (qp1 == NULL)
                break;
            qp = qp1;
        }
        key_arr[pos] = qp->keys[nkey - 1];
        qp->keys[nkey - 1] = key;
```

```
}

value = del(p[pos], key);
if (value != LessKeys)
    return value;

if (pos > 0 && p[pos - 1]->n > min) {
    pivot = pos - 1;
    lptr = p[pivot];
    rptr = p[pos];
    rptr->p[rptr->n + 1] = rptr->p[rptr->n];

    for (i = rptr->n; i > 0; i--) {
        rptr->keys[i] = rptr->keys[i - 1];
        rptr->p[i] = rptr->p[i - 1];
    }

    rptr->keys[0] = key_arr[pivot];
    rptr->p[0] = lptr->p[lptr->n];
    key_arr[pivot] = lptr->keys[--lptr->n];

    return Success;
}

if (pos < n && p[pos + 1]->n > min) {
    pivot = pos;
    lptr = p[pivot];
    rptr = p[pivot + 1];

    lptr->keys[lptr->n] = key_arr[pivot];
    lptr->p[lptr->n + 1] = rptr->p[0];
    key_arr[pivot] = rptr->keys[0];
    lptr->n++;
    rptr->n--;

    for (i = 0; i < rptr->n; i++) {
        rptr->keys[i] = rptr->keys[i + 1];
        rptr->p[i] = rptr->p[i + 1];
    }

    rptr->p[rptr->n] = rptr->p[rptr->n + 1];

    return Success;
}
```

```
    if (pos == n)
        pivot = pos - 1;
    else
        pivot = pos;
    lptr = p[pivot];
    rptr = p[pivot + 1];
    lptr->keys[lptr->n] = key_arr[pivot];
    lptr->p[lptr->n + 1] = rptr->p[0];

    for (i = 0; i < rptr->n; i++) {
        lptr->keys[lptr->n + 1 + i] = rptr->keys[i];
        lptr->p[lptr->n + 2 + i] = rptr->p[i + 1];
    }

    lptr->n = lptr->n + rptr->n + 1;
    free(rptr);

    for (i = pos + 1; i < n; i++) {
        key_arr[i - 1] = key_arr[i];
        p[i] = p[i + 1];
    }

    return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;
}
```

Q.13. Graphs ussing array

```
#include <stdio.h>

#define MAX_VERTICES 100

int adj_matrix[MAX_VERTICES][MAX_VERTICES];
int num_vertices = 0;

void add_vertex() {
    if (num_vertices < MAX_VERTICES) {
        num_vertices++;
    } else {
        printf("Max number of vertices reached\n");
    }
}

void add_edge(int i, int j) {
```

```
    if (i < num_vertices && j < num_vertices) {
        adj_matrix[i][j] = 1;
        adj_matrix[j][i] = 1;
    } else {
        printf("Invalid vertex index\n");
    }
}

void print_graph() {
    printf("Adjacency Matrix:\n");
    for (int i = 0; i < num_vertices; i++) {
        for (int j = 0; j < num_vertices; j++) {
            printf("%d ", adj_matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    //switch case
    int choice;
    int i, j;
    do {
        printf("1. Add vertex\n");
        printf("2. Add edge\n");
        printf("3. Print graph\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                add_vertex();
                break;
            case 2:
                printf("Enter vertex indices: ");
                scanf("%d %d", &i, &j);
                add_edge(i, j);
                break;
            case 3:
                print_graph();
                break;
            case 4:
                break;
            default:
```

```
            printf("Invalid choice\n");
        }
    } while (choice != 4);
    return 0;
}
```