1) a. Write a program to implement Linear or sequential Search on the elements of a given array.

Program Statement:
Write a program to implement Linear or sequential Search on the elements of a given array.

Algorithm: Linear Search
Input: An array A with 'n' elements and K is the key to be searched
Output: if key is found then search is said to be successful and the location of the array element where K matches, otherwise an unsuccessful
Remarks: Assume that array is not empty and elements are in random order
Data Structure: An array data structure
Steps:

```
1. start
2. flag = 0
3. for i = 0 to n do
4.       if (a[i] = = key) then
5.            flag = 1
6.            break
7.       endif
8. endfor
9. if (flag = = 1) then
10.      print " Search is successful"
11.      print " Element is found at position in the list", key, i+1
12. else
13.      print " Search is unsuccessful"
14.      print " Element is not found in the list ", key
15. endif
16. stop
```

Program:**//** Linear Search using Function

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],i,n,key,flag = 0,loction=0;
printf("enter how many elements to search(n<10) :");
scanf("%d",&n);
printf("enter the %d element : ",n);
for (i=0;i<n; i++)
{
printf("\nenter a[%d] element : ",i);
```

```
scanf("%d",&a[i]);
}
printf("\nenter key element : ");
scanf("%d",&key);
for (i=0;i<n;i++)
{
if(a[i] ==key)
{
flag=1;
location=i;
}
}
if(flag==1)
{
printf(" enter %d found",location);
}
else
{
printf("element not found");
}
}
```

Output:

Test case 1:

enter how many elements to search(n<10) :5
enter the 5 element :
enter a[0] element : 5
enter a[1] element : 4
enter a[2] element : 3
enter a[3] element : 2
enter a[4] element : 1
enter key element : 3
enter 2 found

Test case 2:

enter how many elements to search(n<10) :4
enter the 4 element :
enter a[0] element : 8
enter a[1] element : 4
enter a[2] element : 6
enter a[3] element : 3
enter key element : 9
element not found

1) b. Write a program to implement Binary Search on the elements of a given array.

Program Statement:

Write a program to implement Binary Search on the elements of a given array.

Algorithm: Binary Search

Input: An array A with 'n' elements and K is the key to be searched

Output: if key is found then search is said to be successful and the location of the array element where K matches, otherwise an unsuccessful

Remarks: All elements in the array A are stored in ascending order

Data Structure: An array data structure

Steps:

```
1. start
2. l = 0, u = n-1, flag = 0
3. while ( l < u ) do
4.      if ( key = = a[mid] ) then
5.              flag = 1
6.              break
7.      endif
8.      if ( key > a[mid] ) then
9.              l = mid + 1
10.     else
11.             u = mid - 1
12.     endif
13.     mid = ( l + u ) / 2
14. endwhile
15. if (flag = = 1) then
16.     print " Search is successful"
17.     print " Element is found at position in the list", key, i+1
18. else
19.     print " Search is unsuccessful"
20.     print " Element is not found in the list ", key
21. endif
22. stop
```

Program: **//** Binary Search using Function

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i, arr[10],search,first,last,middle;
printf("enter 10 elements (in ascending order) : ");
```

```
for(i=0;i<10;i++)
{
scanf("%d",&arr[i]);
}
printf("\nenter element to be search : ");
scanf("%d",&search);
first = 0;
last=9;
middle=(first+last)/2;
while(first <= last)
{
if(arr[middle]<search)
{
first = middle+1;
}
else if (arr[middle]==search)
{
printf("\nthe number,%d found at position %d",search,middle+1);
break;
}
else
{
last = middle-1;
}
middle = (first + last)/2;
}
if (first>last)
{
printf("\nthe number %d is not found in given array",search);
}
getch();
return 0;
}
```

Output:

Test case 1:

enter 10 elements (in ascending order) : 1 2 3 4 5 6 7 8 9 10
enter element to be search : 6
the number,6 found at position 6

Test case 2:

enter 10 elements (in ascending order) : 4 6 8 9 10 16 20 21 23 30
enter element to be search : 2
the number 2 is not found in given array

2) Write a program to implement operations on Stack using arrays.

Program Statement :
Write a program to implement operations on Stack using arrays.

Algorithm: Push Array
Input: the new ITEM to be pushed onto it.
Output: a stack with a newly pushed ITEM at the TOP position.
Data structure: An array representation of a stack with TOP as the pointer to the top-most element.
Steps:
1.if (TOP = = N-1) then /* check for stack overflow */
2. print "stack is overflow i.e., full" and return
3.else
4. Read ITEM
5. TOP = TOP+1
6. stack[TOP] = ITEM
7.endif
8.stop

Algorithm: Pop array
Input: A stack with elements.
Output: Removes an ITEM from the top of the stack if it is not empty.
Data structure: An array representation of a stack with TOP as the pointer to the top-most element.
Steps:
1. if (TOP = = -1) then /* check for stack underflow */
2. print "stack is underflow i.e., empty " and return
3. else
4. ITEM = STACK[TOP]
5. TOP = TOP-1
6. print " The deleted element is", ITEM
6. endif
7. stop

Algorithm: Show array
Input: A stack with elements.
Output: Displays all the elements of the stack exactly once
Data structure: An array representation of the stack with TOP as the pointer to the top-most element.
Steps:
1. if (TOP = = -1) then
2. print " stack is empty" and return
3. else
4. print " the stack elements are"
5. for i=0 to TOP do

6. print ‘STACK[i]’
7. endfor
8. endif
9. stop

Program: **//** operations on stack using arrays

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t
                                   3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
```

```
            printf("\n\t EXIT POINT ");
            break;
          }
          default:
          {
            printf ("\n\t Please Enter
                        a Valid Choice(1/2/3/4)");
          }

        }
      }
      while(choice!=4);
      return 0;
    }
    void push()
    {
      if(top>=n-1)
      {
        printf("\n\tSTACK is over flow");
      }
      else
      {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
      }
    }
    void pop()
    {
      if(top<=-1)
      {
        printf("\n\t Stack is under flow");
      }
      else
      {
        printf("\n\t The popped elements is %d",
                                      stack[top]);
        top--;
      }
    }
    void display()
    {
      if(top>=0)
```

```
    {
       printf("\n The elements in STACK \n");
       for(i=top; i>=0; i--)
          printf("\n%d",stack[i]);
       printf("\n Press Next Choice");
    }
    else
    {
       printf("\n The STACK is empty");
    }

}
```

Output:

Test case 1:

```
Enter the size of STACK[MAX=100]:5
STACK OPERATIONS USING ARRAY
            --------------------------------
             1.PUSH
             2.POP
            3.DISPLAY
             4.EXIT
 Enter the Choice:3
 The STACK is empty
 Enter the Choice:1
 Enter a value to be pushed:4
 Enter the Choice:2
 The popped elements is 4
 Enter the Choice:5
 Please Enter a Valid Choice(1/2/3/4)
 Enter the Choice:4
 EXIT POINT
```

Test case 2:

```
Enter the size of STACK[MAX=100]:5
STACK OPERATIONS USING ARRAY
            --------------------------------
             1.PUSH
             2.POP
            3.DISPLAY
             4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:5
 Enter the Choice:1
```

```
 Enter a value to be pushed:6
 Enter the Choice:1
 Enter a value to be pushed:7
 Enter the Choice:3
 The elements in STACK

7
6
5
 Press Next Choice
 Enter the Choice:2
 The popped elements is 7
 Enter the Choice:3
 The elements in STACK

6
5
 Press Next Choice
 Enter the Choice:4
 EXIT POINT
```

3) a. Write a Program to convert a given infix expression into its Postfix using stack.

Program Statement:
Write a Program to convert a given infix expression into its Postfix using stack.

Algorithm: InfixToPostfix
Input: E, simple arithmetic expression in infix notation
Output: An arithmetic expression in postfix notation
Data structure: Array representation of a stack with TOP as pointer to the top-most element.
Steps:
1. Scan the infix string or expression from left to right.
2. Create an empty stack
3. If the scanned character is an operand, add it to the postfix string. If the scanned character is an operator and if the stack is empty, push the character to stack.
4. If the scanned character is an operator and the stack is not empty, compare the precedence of the scanned character with the element on top of the stack. If top stack has higher precedence over the scanned character then pop the stack, else push the scanned character into the stack. Repeat this step as long as the stack is not empty and top stack has precedence over the scanned character. Continue this step till all the characters are scanned.
5. Left parenthesis : push into the stack
6. Right parenthesis : pop elements from the stack till left parenthesis is encountered.
7. Return the postfix string

Program: // Infix to Postfix Conversion

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

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

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

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

```
int main()
{
   char exp[100];
   char *e, x;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");
   e = exp;

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

   while(top != -1)
   {
      printf("%c ",pop());
   }return 0;
}
```

Output:

Test case 1:

Enter the expression : a+b-c

a b + c -

Test case 2:

Enter the expression : (a+b)-(a*b)

a b + a b * -

3) b. Write a Program to evaluate the Postfix Expression using stack.

Program Statement:
Write a Program to evaluate the Postfix Expression using stack

Algorithm: EvaluatePostfix
Input: E, an expression in postfix notation, with values of the operands appearing in the expression
Output: value of the expression
Data structure: Array representation of a stack with TOP as the pointer to the top-most element.
Steps:
1. Append a special delimiter '#' at the end of the expression
2. item = E.ReadSymbol( )
3. while(item !='#') do
4. if (item = = operand) then
5. push(item)
6. else
7. op = item
8. y = POP( )
9. x = POP( )
10. t = x op y
11. PUSH(t)
12. endif
13. item = E.ReadSymbol( )
14.endwhile
15.value = POP( )
16.return(value)
17.stop

Program: // evaluation of the Postfix Expression using stack

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

#define SIZE 40

int pop();
void push(int);

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

int main()
```

```
{
        int i, a, b, result, pEval;
        char ch;

        for(i=0; i<SIZE; i++)
        {
                stack[i] = -1;
        }
        printf("\nEnter a postfix expression: ");
        scanf("%s",postfix);

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

                if(isdigit(ch))
                {
                        push(ch-'0');
                }
                else if(ch == '+' || ch == '-' || ch == '*' || ch == '/')
                {
                        b = pop();
                        a = pop();

                        switch(ch)
                        {
                                case '+': result = a+b;
                                         break;
                                case '-': result = a-b;
                                         break;
                                case '*': result = a*b;
                                         break;
                                case '/': result = a/b;
                                         break;
                                case '%':result = a%b;
                                         break;
                        }

                        push(result);
                }

        }

        pEval = pop();
```

```
        printf("\nThe postfix evaluation is: %d\n",pEval);

        return 0;
}

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

Output:

Test case 1:

Enter a postfix expression: 456*+
The postfix evaluation is: 34

Test case 2:

Enter a postfix expression: 756+*
The postfix evaluation is: 77

4) Write a program to implement operations on Queue using array.

Program Statement :
Write a program to implement operations on Queue using array

Algorithm : Enqueue_array
Input: An element ITEM that has to be inserted.
Output: The ITEM is at the REAR of the queue.
Data structure: The array representation of a queue structure; two pointers FRONT and REAR of the queue.
Steps:
1. if(REAR = = N-1) then /* check for queue overflow */
2. print "queue is full" and return
3. else
4. if(FRONT = = -1) then
5. FRONT = 0, REAR = 0
6. else
7. REAR = REAR+1
8. endif
9. QUE[REAR] = ITEM
10. endif
11. stop

Algorithm: Dequeue_array
Input: A Queue with elements
Output: Removes an ITEM from the FRONT of the queue if it is not empty.
Data structure: The array representation of a queue structure; two pointers FRONT and REAR of the queue.
Steps:
1. if(FRONT = = -1) then /* check for queue underflow*/
2. print " Queue is empty " and return
3. else
4. ITEM = QUE[FRONT]
5. if(REAR = = FRONT)
6. REAR= -1, FRONT= -1
7. else
8. FRONT = FRONT+1
9. endif
10. endif
11. stop

Algorithm: size_array
Input: A queue with elements.
Output: No. of elements in the queue.
Data structure: The array representation of a queue structure; two pointers FRONT and REAR of the queue.
Steps:
1. if (FRONT = = -1) then
2. print " queue is empty" and return
3. else
4. print " the no. of elements in the queue is", REAR + 1 - FRONT
5. endif
6. stop

Algorithm: show_array
Input: A queue with elements.
Output: Displays all the elements of the queue exactly once
Data structure: The array representation of a queue structure; two pointers FRONT and REAR of the queue.
Steps:
1. if (FRONT = = -1) then
2. print “ queue is empty” and return
3. else
4. print “ the queue elements are”
5. for i = FRONT to REAR do
6. print ‘QUE[i]’
7. endfor
8. endif
9. stop

Program: // operations on Queue using array

```
#include<stdio.h>
#define n 5
int main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
   printf("Queue using Array");
   printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
   while(ch)
   {
      printf("\nEnter the Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:
         if(rear==x)
            printf("\n Queue is Full");
         else
         {
            printf("\n Enter no %d:",j++);
            scanf("%d",&queue[rear++]);
         }
         break;
      case 2:
         if(front==rear)
         {
            printf("\n Queue is empty");
         }
         else
         {
            printf("\n Deleted Element is %d",queue[front++]);
            x++;
         }
         break;
      case 3:
         printf("\nQueue Elements are:\n ");
         if(front==rear)
            printf("\n Queue is Empty");
         else
```

```
        {
          for(i=front; i<rear; i++)
          {
            printf("%d",queue[i]);
            printf("\n");
          }
          break;
        case 4:
          printf("\n\t EXIT POINT");
          break;
        default:
          printf("Wrong Choice: please see the options");
        }
      }
    }
  return 0;
}
```

Output:

Test case 1:

Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:1
Enter no 1:9
Enter the Choice:1
Enter no 2:8
Enter the Choice:1
Enter no 3:7
Enter the Choice:2
Deleted Element is 9
Enter the Choice:3
Queue Elements are:
8
7

Enter the Choice:4
EXIT POINT

Test case 2:

Queue using Array
1.Insertion
2.Deletion
3.Display
4.Exit
Enter the Choice:2

```
Queue is empty
Enter the Choice:1
Enter no 1:4
Enter the Choice:1
Enter no 2:5
Enter the Choice:2
Deleted Element is 4
Enter the Choice:3
Queue Elements are:
 5

Enter the Choice:6
Wrong Choice: please see the options
Enter the Choice:4
EXIT POINT
```

5) Write a Program to Implement Circular Queue Operations using Arrays.

Program Statement:

Write a Program to Implement Circular Queue Operations using Arrays.

Algorithm: ENQUEUE_CQ_array

Input: An element ITEM to be inserted into the circular queue.

Output: The ITEM is at the REAR of the circular queue.

Data structure: The array representation of a circular queue structure; two pointers FRONT and REAR of the queue.

Steps:

1. if(FRONT = = (REAR+1)%N) then /* check for circular queue overflow*/
2. print "circular queue is full" and return
3. else
4. if(FRONT = = -1) then
5. FRONT = 0, REAR = 0
6. else
7. REAR = (REAR+1) % N
8. endif
9. CIR_QUE[REAR] = ITEM
10. endif
11. stop

Algorithm: DEQUEUE_CQ_array

Input: A circular queue with elements

Output: Removes an ITEM from the FRONT of the circular queue if it is not empty.

Data structure: The array representation of a circular queue structure; two pointers FRONT and REAR of the queue.

Steps:

1. if(FRONT = = -1) then /* check for circular queue underflow*/
2. print " Circular Queue is empty " and return
3. else
4. ITEM = CIR_QUE[FRONT]
5. if(REAR = = FRONT)
6. REAR= -1, FRONT= -1
7. else
8. FRONT = (FRONT+1)%N
9. endif
10. endif
11. stop

Algorithm: show_CQ_array

Input: A circular queue with elements.

Output: Displays all the elements of the circular queue exactly once

Data structure: The array representation of a circular queue structure; two pointers FRONT and REAR of the queue.

Steps:

1. start
2. if(rear = = -1) then
3. print "circular queue is underflow“ and return;
4. endif
5. print "the elements are:"
6. if(front<= rear) then
7. for i=front to rear do
8. print ‘a[i]’
9. endfor
10. else
11. for i=front to MAX do
12. print ‘a[i]’
13. endfor
14. for i=0 to rear do
15. print ‘ a[i]’
16. endfor
17. endif
18. stop

Program: // Circular Queue Operations using Arrays

```
#include <stdio.h>

#define SIZE 5

int items[SIZE];
int front = -1, rear = -1;

// Check if the queue is full
int isFull() {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}

// Check if the queue is empty
int isEmpty() {
```

```
  if (front == -1) return 1;
  return 0;
}

// Adding an element
void enQueue(int element) {
  if (isFull())
    printf("\n Queue is full!! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }
}

// Removing an element
int deQueue() {
  int element;
  if (isEmpty()) {
    printf("\n Queue is empty !! \n");
    return (-1);
  } else {
    element = items[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
    else {
      front = (front + 1) % SIZE;
    }
    printf("\n Deleted element -> %d \n", element);
    return (element);
  }
}

// Display the queue
void display() {
  int i;
  if (isEmpty())
    printf(" \n Empty Queue\n");
  else {
```

```
    printf("\n Front -> %d ", front);
    printf("\n Items -> ");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d ", items[i]);
    }
    printf("%d ", items[i]);
    printf("\n Rear -> %d \n", rear);
  }
}

int main() {
  // Fails because front = -1
  deQueue();

  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);

  display();
  deQueue();

  display();

  enQueue(7);
  display();

  // Fails to enqueue because front == rear + 1
  enQueue(8);

  return 0;
}
```

Output:

```
Queue is empty !!

 Inserted -> 1
 Inserted -> 2
 Inserted -> 3
 Inserted -> 4
```

```
Inserted -> 5
Queue is full!!

Front -> 0
Items -> 1 2 3 4 5
Rear -> 4

Deleted element -> 1

Front -> 1
Items -> 2 3 4 5
Rear -> 4

Inserted -> 7
Front -> 1
Items -> 2 3 4 5 7
Rear -> 0

Queue is full!!
```

6) Write a program to implement operations on Singly linked list.

Program Statement:

Write a program to implement operations on Singly linked list.

Algorithm: Insert Front_Singly linked list

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: A singly linked list with a newly inserted node at the front of the list.

Data structure: a singly linked list whose address of the starting node is known from the HEADER.

Steps:

1. Read ele
2. temp = (struct node*)malloc(sizeof(struct node))
3. if(temp = = NULL) then
4. print “ memory allocation error"
5. return
6. endif
7. if(head->link = = NULL) then
8. temp->data = ele
9. head->link = temp
10. temp->link = NULL
11. else
12. temp->link = head->link
13. head->link = temp
14. temp->data = ele
15. endif
16. stop

Algorithm: Insert Last_Singly linked list

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted.

Output: a singly linked list with a newly inserted node at the end of the list.

Data structure: a singly linked list whose address of the starting node is known from the HEADER.

Steps:

1. Read ele
2. temp = (struct node*)malloc(sizeof(struct node))
3. if(temp = = NULL) then
4. print " memory allocation error"
5. return
6. endif
7. if(head->link = = NULL) then
8. temp->data = ele
9. head->link = temp
10. temp->link = NULL
11. else
12. temp1 = head->link
13. while(temp1->link != NULL) do
14. temp1 = temp1->link

15. endwhile
16. temp1->link = temp
17. temp->data = ele
18. temp->link = NULL
19. endif
20. stop

Algorithm: Insert Specified Position_Singly linked list

Input: HEADER is the pointer to the header node and X is the data of the node to be inserted, and KEY being the data of the key node after which the node has to be inserted.

Output: A singly linked list enriched with newly inserted node having data X after the node with data KEY.

Data structure: a singly linked list whose address of the starting node is known from the HEADER.

Steps:

1. Read ele, pos
2. temp = (struct node*)malloc(sizeof(struct node))
3. if(temp = = NULL) then
4. print " memory allocation error"
5. return;
6. endif
7. if(head->link = = NULL) then
8. temp->data = ele
9. head->link = temp
10. temp->link = NULL
11. else
12. temp1 = head->link
13. i = 1
14. while(i < pos) do
15. temp1 = temp1->link
16. i++
17. endwhile
18. temp->link = temp1->link
19. temp1->link = temp
20. temp->data = ele
21. endif
22. stop

Program: // operations on Singly linked list

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
struct node {
  int value;
  struct node *next;
};
```

```
void insert();
void display();
void delete();
int count();

typedef struct node DATA_NODE;

DATA_NODE *head_node, *first_node, *temp_node = 0, *prev_node, next_node;
int data;

int main() {
  int option = 0;

  printf("Singly Linked List Example - All Operations\n");

  while (option < 5) {

    printf("\nOptions\n");
    printf("1 : Insert into Linked List \n");
    printf("2 : Delete from Linked List \n");
    printf("3 : Display Linked List\n");
    printf("4 : Count Linked List\n");
    printf("Others : Exit()\n");
    printf("Enter your option:");
    scanf("%d", &option);
    switch (option) {
      case 1:
        insert();
        break;
      case 2:
        delete();
        break;
      case 3:
        display();
        break;
      case 4:
        count();
        break;
      default:
        break;
    }
  }

  return 0;
}

void insert() {
```

```
  printf("\nEnter Element for Insert Linked List : \n");
  scanf("%d", &data);

  temp_node = (DATA_NODE *) malloc(sizeof (DATA_NODE));

  temp_node->value = data;

  if (first_node == 0) {
    first_node = temp_node;
  } else {
    head_node->next = temp_node;
  }
  temp_node->next = 0;
  head_node = temp_node;
  fflush(stdin);
}

void delete() {
  int countvalue, pos, i = 0;
  countvalue = count();
  temp_node = first_node;
  printf("\nDisplay Linked List : \n");

  printf("\nEnter Position for Delete Element : \n");
  scanf("%d", &pos);

  if (pos > 0 && pos <= countvalue) {
    if (pos == 1) {
      temp_node = temp_node -> next;
      first_node = temp_node;
      printf("\nDeleted Successfully \n\n");
    } else {
      while (temp_node != 0) {
        if (i == (pos - 1)) {
          prev_node->next = temp_node->next;
          if(i == (countvalue - 1))
          {
                                head_node = prev_node;
                        }
          printf("\nDeleted Successfully \n\n");
          break;
        } else {
          i++;
          prev_node = temp_node;
          temp_node = temp_node -> next;
        }
      }
```

```
    }
  } else
    printf("\nInvalid Position \n\n");
}

void display() {
  int count = 0;
  temp_node = first_node;
  printf("\nDisplay Linked List : \n");
  while (temp_node != 0) {
    printf("# %d # ", temp_node->value);
    count++;
    temp_node = temp_node -> next;
  }
  printf("\nNo Of Items In Linked List : %d\n", count);
}

int count() {
  int count = 0;
  temp_node = first_node;
  while (temp_node != 0) {
    count++;
    temp_node = temp_node -> next;
  }
  printf("\nNo Of Items In Linked List : %d\n", count);
  return count;
}
```

Output:

Test case 1:

Singly Linked List Example - All Operations

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1
Enter Element for Insert Linked List :
2
Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List

```
4 : Count Linked List
Others : Exit()
Enter your option:1
Enter Element for Insert Linked List :
3
Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1
Enter Element for Insert Linked List :
4
Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:1
Enter Element for Insert Linked List :
5
Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:2
No Of Items In Linked List : 4

Display Linked List :

Enter Position for Delete Element :
2
Deleted Successfully


Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
```

Others : Exit()
Enter your option:3
Display Linked List :
2 # # 4 # # 5
No Of Items In Linked List : 3

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:4
No Of Items In Linked List : 3

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:5

<u>Test case 2:</u>

Singly Linked List Example - All Operations

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:3
Display Linked List :

No Of Items In Linked List : 0

Options
1 : Insert into Linked List
2 : Delete from Linked List
3 : Display Linked List
4 : Count Linked List
Others : Exit()
Enter your option:7

7) a. Write a program to sort the elements of an array using bubble sort (i.e., sorting by exchange).

Program Statement:

Write a program to sort the elements of an array using bubble sort (i.e., sorting by exchange).

Algorithm: Bubble sort

Input: An array A with 'n' elements

Output: An array A with 'n' elements in sorted order

Remarks: Sort the elements in ascending order

Data Structure: An array data structure

Steps:

1. start
2. for i = 0 to n - 1 do
3. for j = 0 to n - 1 - i do
4. if (A[j] > A[j+1]) then
5. temp = A[j]
6. A[j] = A[j+1]
7. A[j+1] = temp
8. endif
9. endfor
10. endfor
11. stop

Program: // to sort the elements of an array using bubble sort

```
#include <stdio.h>

int main()
{
 int array[100], n, c, d, swap;

 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]);

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

```
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (array[d] > array[d+1]) /* For decreasing order use '<' instead of '>' */
      {
        swap       = array[d];
        array[d]   = array[d+1];
        array[d+1] = swap;
      }
    }
  }

  printf("Sorted list in ascending order:\n");

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

  return 0;
}
```

Output:

Test case 1:

Enter number of elements
5
Enter 5 integers
26
84
33
65
29
Sorted list in ascending order:
26
29
33
65
84

Test case 2:

Enter number of elements
7
Enter 7 integers
11
66
44
88

22
99
55
Sorted list in ascending order:
11
22
44
55
66
88
99

7) b. Write a program to sort the elements of an array using Selection Sort.

Program Statement:
Write a program to sort the elements of an array using Selection Sort.

Algorithm: Selection sort
Input: An array A with 'n' elements
Output: An array A with 'n' elements in sorted order
Remarks: Sort the elements in ascending order
Data Structure: An array data structure
Steps:
1. start
2. for i = 0 to n - 1 do
3. min = i
4. for j = i+1 to n do
5. if (a[j] < a[min]) then
6. min = j
7. endif
8. endfor
9. temp = a[min]
10. a[min] = a[i]
11. a[i] = temp
12. endfor
13. stop

Program: // to sort the elements of an array using Selection Sort

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

 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]);

 for (c = 0; c < (n - 1); c++) // finding minimum element (n-1) times
 {
   position = c;
```

```
  for (d = c + 1; d < n; d++)
  {
    if (array[position] > array[d])
      position = d;
  }
  if (position != c)
  {
    t = array[c];
    array[c] = array[position];
    array[position] = t;
  }
 }

 printf("Sorted list in ascending order:\n");

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

 return 0;
}
```

Output:

Test case 1:

Enter number of elements
5
Enter 5 integers
34
54
43
45
62
Sorted list in ascending order:
34
43
45
54
62

Test case 2:

Enter number of elements
10
Enter 10 integers
23

```
54
36
52
47
86
74
69
45
12
Sorted list in ascending order:
12
23
36
45
47
52
54
69
74
86
```

7) c. Write a program to sort elements using insertion sort.

Program Statement:

Write a program to sort elements using insertion sort.

Algorithm: Insertion sort

Input: An array A with 'n' elements

Output: An array A with 'n' elements in sorted order

Remarks: Sort the elements in ascending order

Data Structure: An array data structure

Steps:

1. for i = 1 to n do
2. index = a[i]
3. j = i
4. while ((j > 0) && (a[j-1] > index)) do
5. a[j] = a[j-1]
6. j = j - 1
7. endwhile
8. a[j] = index
9. endfor
10. stop

Program: // to sort elements using insertion sort

```
#include <stdio.h>

int main()
{
  int n, array[1000], c, d, t, flag = 0;

  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]);

  for (c = 1 ; c <= n - 1; c++) {
    t = array[c];

    for (d = c - 1 ; d >= 0; d--) {
      if (array[d] > t) {
```

```
        array[d+1] = array[d];
        flag = 1;
      }
      else
        break;
    }
    if (flag)
      array[d+1] = t;
  }

  printf("Sorted list in ascending order:\n");

  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }

  return 0;
}
```

Output:

Test case 1:

Enter number of elements
5
Enter 5 integers
22
55
33
11
10
Sorted list in ascending order:
10
11
22
33
55

Test case 2:

Enter number of elements
10
Enter 10 integers
16
15
18
17

```
19
13
14
12
11
10
Sorted list in ascending order:
10
11
12
13
14
15
16
17
18
19
```

8) a. Write a program to sort elements using quick sort.

Program Statement:

Write a program to sort elements using quick sort.

Algorithm: Quick_sort

Input: An array A[l,……,u] where l and u are lower and upper indexes of A

Output: Array A[l,…….,u] with all elements arranged in ascending order

Data Structure: An array data structure

Steps:

1. Start
2. Pivot = lb, i = lb, j = ub
3. While(i<j) do
4. while(A[i]<=A[pivot]) do
5. i = i+1
6. endwhile
7. while(A[j]>A[pivot]) do
8. j = j-1
9. endwhile
10. if(i<j) then
11. A[j] = (A[i]+A[j]) – (A[i] = A[j])
12. else
13. A[pivot] = (A[pivot] + A[j]) – (A[j] = A[pivot])
14. endif
15. Endwhile
16. Quicksort(A,lb,j-1)
17. Quicksort(A,j+1,ub)
18. stop

Program: // to sort elements using quick sort.

```
#include<stdio.h>
void quicksort(int number[25],int first,int last){
  int i, j, pivot, temp;

  if(first<last){
    pivot=first;
    i=first;
    j=last;

    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
        i++;
```

```
      while(number[j]>number[pivot])
        j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }

    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);

  }
}

int main(){
  int i, count, number[25];

  printf("Enter the no of elements: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  quicksort(number,0,count-1);

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  return 0;
}
```

Output:

Test case 1:

Enter the no of elements: 5
Enter 5 elements: 52 43 12 25 42
Order of Sorted elements: 12 25 42 43 52

Test case 2:

Enter the no of elements: 3
Enter 3 elements: 23 51 12
Order of Sorted elements: 12 23 51

8) b. Write a program to sort the elements using merge sort.

Program Statement:
Write a program to sort the elements using merge sort.

Algorithm: Merge_sort
Input: An array A[l,……,u] where l and u are lower and upper indexes of A
Output: Array A[l,…….,u] with all elements arranged in ascending order
Data Structure: An array data structure
Steps:
1. Start
2. if(l<u) then
3. mid=(l+u)/2
4. mergesort(a, l, mid)
5. mergesort(a,mid+1,u)
6. merge(a, l, mid, u)
7. endif
8. Stop

Algorithm: merge(int a[ ], int l, int mid, int u)
Input: An array A with two sub_arrays where indices range from l to mid and mid+1 to u
Output: The two sub_arrays are merged and sorted in the array A
Steps:
1. i=l, j=mid+1,k=0
2. while(i <= mid && j <= u) do
3. if(a[i] <= a[j]) then
4. b[k] = a[i]
5. k=k+1,i=i+1
6. else
7. b[k] = a[j]
8. k=k+1,j=j+1
9. endif
10. endwhile
11. while(i <= mid) do
12. b[k] = a[i]
13. k=k+1,i=i+1
14. endwhile
15. while(j <= u) do
16. b[k]=a[j]
17. k=k+1,j=j+1
18. endwhile
19. for k=0 to k <= u-l do

20. a[k+l] = b[k]
21. endfor

Program: // to sort the elements using merge sort

```
#include<stdio.h>
void merge(int arr[],int min,int mid,int max)
{
    int tmp[30];
    int i,j,k,m;
    j=min;
    m=mid+1;
    for(i=min; j<=mid && m<=max ; i++)
    {
        if(arr[j]<=arr[m])
        {
            tmp[i]=arr[j];
            j++;
        }
        else
        {
            tmp[i]=arr[m];
            m++;
        }
    }
    if(j>mid)
    {
        for(k=m; k<=max; k++)
        {
            tmp[i]=arr[k];
            i++;
        }
    }
    else
    {
        for(k=j; k<=mid; k++)
        {
            tmp[i]=arr[k];
            i++;
        }
    }
    for(k=min; k<=max; k++)
    arr[k]=tmp[k];
}
```

```
void sortm(int arr[],int min,int max)
{
    int mid;
    if(min<max)
    {
        mid=(min+max)/2;
        sortm(arr,min,mid);
        sortm(arr,mid+1,max);
        merge(arr,min,mid,max);
    }
}
int main()
{
    int arr[30];
    int i,size;
    printf("\tMerge sort\n");
    printf("-----------------------------------\n");
    printf(" Enter no of elements for sorting ");
    scanf("%d",&size);
    printf("\n Enter %d elements :\n ",size);
    for(i=0; i<size; i++)
    {
        scanf("%d",&arr[i]);
    }
    sortm(arr,0,size-1);
    printf("\n Sorted elements after using merge sort:\n\n");
    for(i=0; i<size; i++)
        printf(" %d ",arr[i]);
    return 0;
}
```

Output:

Test case 1:

```
Merge sort
-----------------------------------
 Enter no of elements for sorting 5
 Enter 5 elements :
 76
 54
 37
 62
 18
 Sorted elements after using merge sort:
```

18 37 54 62 76

Test case 2:

Merge sort

Enter no of elements for sorting 10

Enter 10 elements :

67

64

69

63

68

66

62

61

65

60

Sorted elements after using merge sort:

9 60 61 62 63 64 65 66 67 68

9) Write a Program to implement operations on trees.

Program Statement:

Write a Program to implement operations on trees

Algorithm: InsertBinaryTree_LINK

Input: KEY is the data content of the key node after which a new node is to be inserted and ITEM is the data content of the new node that has to be inserted.

Output: a node with data component ITEM inserted an external node after the node having data KEY if such a node exists with empty link(s), that is, either child or both children is / are absent

Data structure: Linked structure of a binary tree. ROOT is the pointer to the root node.

Steps:

```
1.  ptr = search_LINK(ROOT,KEY)
2.  if (ptr = NULL )then
3.       Print "Search is unsuccessful : No insertion"
4.       Exit
5.  endif
6.  if (ptr→LC = NULL) or (ptr→RC = NULL)
7.       Read option to insert as left (L) or right (R) child (give option = L/R)
8.       if(option = L) then
9.           if(ptr→LC = NULL) then
10.              new = GetNode(NODE)
11.              new→DATA = ITEM
12.              new→LC = new→RC = NULL
13.              ptr→LC = new
14.          else
15.              print "Insertion is not possible as left child"
16.              exit
17.          endif
18.      else
19.              if (ptr→RC = NULL)
20.                  new = GetNode (NODE)
21.                  new →DATA = ITEM
22.                  new→LC = new→RC = NULL
23.                  ptr→RC = new
24.             else
25.                  print "Insertion is not possible as right child"
26.                  exit
27.              endif
28. else
29.      print "The key node already has child"
30.  endif
31. endif
32.  stop
```

Algorithm: DeleteBinary Tree_LINK

Input: A binary tree whose address of the root pointer is known from ROOT and ITEM is the data of the node identified for deletion.

Output: A binary tree without having a node with data ITEM.

Data Structure: Linked structure of a binary tree having ROOT as the pointer to the root node.

Steps:

```
1.  ptr = ROOT
2.  if (ptr = NULL) then
3.      print "Tree is empty"
4.      Exit
5.  endif
6.  parent = SearchParent(ROOT, ITEM)
7.  if(parent != NULL) then
8.      ptr1 = parent→LC,ptr2 = parent→RC
9.      if(ptr1→DATA = ITEM) then
10.         if(ptr1→LC = NULL) and (ptr1→RC = NULL) then
11.             parent→LC = NULL
12.         else
13.             Print "Node is not a leaf node: No deletion"
14.         endif
15.     else
16.         if(ptr2→LC = NULL) and (ptr2→RC = NULL) then
17.             parent→RC = NULL
18.         else
19.             print "Node is not a leave node: No deletion"
20.         endif
21.     endif
22. else
23.     print "Node with data ITEM does not exist: Deletion fails"
24. endif
25. Stop
```

Algorithm: Preorder_Binary Tree_Traversal_Recursively

Input: ROOT is the pointer to the root node of the binary tree

Output: Visiting all the nodes in preorder fashion.

Data Structure: Linked structure of binary tree.

Steps:

```
1. ptr = ROOT
2. if (ptr!=NULL) then
3. visit(ptr)
4. preorder(ptr→LC)
5. preorder(ptr→RC)
6. endif
7. stop
```

Algorithm: Inorder_Binary Tree_Traversal_Recursively
Input: ROOT is the pointer to the root node of the binary tree
Output: Visiting all the nodes in inorder fashion.
Data Structure: Linked structure of binary tree.
Steps:

```
1.      ptr = ROOT
2.      if (ptr!=NULL) then
3.          inorder(ptr->LC)
4.          visit(ptr)
5.          inorder(ptr->RC)
6.      endif
7.      Stop
```

Algorithm: Postorder_Binary Tree_Traversal_Recursively
Input: ROOT is the pointer to the root node of the binary tree
Output: Visiting all the nodes in postorder fashion.
Data Structure: Linked structure of binary tree.
Steps:

```
1.      ptr = ROOT
2.      if (ptr!=NULL) then
3.          postorder(ptr->LC)
4.          postorder(ptr->RC)
5.          visit(ptr)
6.      endif
7.      Stop
```

Program: // to implement operations on trees.

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

struct node {
  int data;

  struct node *leftChild;
  struct node *rightChild;
};

struct node *root = NULL;

void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
```

```
    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if(root == NULL) {
      root = tempNode;
    } else {
      current = root;
      parent = NULL;

      while(1) {
        parent = current;

        //go to left of the tree
        if(data < parent->data) {
          current = current->leftChild;

          //insert to the left
          if(current == NULL) {
            parent->leftChild = tempNode;
            return;
          }
        }  //go to right of the tree
        else {
          current = current->rightChild;

          //insert to the right
          if(current == NULL) {
            parent->rightChild = tempNode;
            return;
          }
        }
      }
    }
  }

  struct node* search(int data) {
    struct node *current = root;
    printf("Visiting elements: ");

    while(current->data != data) {
      if(current != NULL)
        printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
```

```
            current = current->leftChild;
         }
         //else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL) {
            return NULL;
         }
      }

   return current;
}

void pre_order_traversal(struct node* root) {
   if(root != NULL) {
      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);
   }
}

void inorder_traversal(struct node* root) {
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
   }
}

void post_order_traversal(struct node* root) {
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}

int main() {
   int i;
   int array[7] = { 17, 14, 25, 12, 19, 52, 41 };

   for(i = 0; i < 7; i++)
      insert(array[i]);
```

```
   i = 31;
   struct node * temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   i = 15;
   temp = search(i);

   if(temp != NULL) {
      printf("[%d] Element found.", temp->data);
      printf("\n");
   }else {
      printf("[ x ] Element not found (%d).\n", i);
   }

   printf("\nPreorder traversal: ");
   pre_order_traversal(root);

   printf("\nInorder traversal: ");
   inorder_traversal(root);

   printf("\nPost order traversal: ");
   post_order_traversal(root);

   return 0;
}
```

Output:

Visiting elements: 17 25 52 41 [x] Element not found (31).
Visiting elements: 17 14 [x] Element not found (15).

Preorder traversal: 17 14 12 25 19 52 41
Inorder traversal: 12 14 17 19 25 41 52
Post order traversal: 12 14 19 41 52 25 17

10) write a program to perform operations creation, insertion, deletion and traversing on a BST.

Program Statement:

Write a program to perform operations creation, insertion, deletion and traversing on a binary search tree.

Algorithm: Search_BST

Input: ITEM is the data that has to be searched.

Output: If found then pointer to the node containing data Item else a message.

Data Structure: Linked structure of the binary tree. Pointer to the root node is ROOT.

Steps:

```
1.    ptr = ROOT, flag = FLASE                          //Start from the root
2.    while (ptr != NULL) and (flag = FLASE) do
3.          case: ITEM < ptr → DATA                     //Go to the left sub tree
4.                ptr = ptr→ LCHILD
5.          case: ptr → DATA = ITEM                     //Search is successful
6.                flag = TRUE
7.          case: ITEM > ptr → DATA                     //Go to the right sub-Tree
8.                ptr = ptr → RCHILD
9.          endcase
10.   endwhile
11.   if (flag = TRUE) then
12.         print "ITEM has found at the node",ptr      //Search is successful
13.   else
14.         print "ITEM does not exist: Search is unsuccessful"
15.   endif
16.   stop
```

Algorithm: Insert_BST

Input: ITEM is the data component of a node that has to be inserted.

Output: If there is no node having data ITEM. It is inserted into the tree else a message.

Data Structure: Linked structure of binary tree. Pointer to the root node is ROOT.

Steps:

```
1.    ptr = ROOT , flag = FLASE                   //Start from the root node
2.    while (ptr != NULL) and (flag = FLASE) do
3.          case: ITEM < ptr → DATA               // Go to the left sub-Tree
4.              ptr1 = ptr
5.              ptr = ptr → LCHILD
6.          case: ITEM > ptr → DATA               // Go to the right sub-tree
7.                ptr1 = ptr
8.                ptr = ptr → RCHID
9.          case: ptr→ DATA = ITEM                //Node exits
10.               flag = TRUE
```

```
                    print "ITEM already exits"
                    EXIT                                    //Quit the execution
            endcase
    endwhile
    if(ptr = NULL) then                     //Insert when the search halts at the dead end
            new = GetNode(NODE)
            new → DATA = ITEM
            new → LCHID = NULL
            new →RCHILD = NULL
            if (ptr1 → DATA < ITEM) then            //insert as right child
                    ptr1 → RCHILD = new
            else
                    ptr1 → LCHILD = new             //insert as left chid
            endif
    endif
    stop
```

Algorithm: Delete_BST

Input: ITEM is the data of the node to be deleted.

Output: If the node with data ITEM exits it is deleted else a message.

Data Structure: Linked structure of binary tree. Pointer to the root node is ROOT.

Steps:

```
    ptr = ROOT , flag = FLASE
    while (ptr != NULL) and (Flag = FLASE) then
            case:  ITEM <  ptr → DATA                       // Go to the left sub-Tree
                   ptr1 = ptr
                   ptr = ptr → LCHILD
            case: ITEM > ptr → DATA                         // Go to the right sub-tree
                    ptr1 = ptr
                    ptr = ptr → RCHID
            case: ptr → DATA = ITEM                         //Node exits
                    flag = TRUE
                    print "ITEM already exits"
                    EXIT                                    //Quit the execution
    endwhile
    if (flag = FLASE) then
            print "ITEM does not exit: NO deletion"
            Exit

    /*DECIDE THE CASE OF DELETION */
            if (ptr → LCHILD = NULL) the(ptr → RCHILD = NULL) then
                    case = 1
    else
```

```
        if(ptr→ LCHILD != NULL) and (ptr → RCHILD != NULL) then
                case = 3
        else
                case = 2
        end if
endif
/* DELETION: CASE 1 */
iF(case = 1) then
        if(parent → LCHILD = ptr) then
                parent → LCHILD = NULL
        else
                Parent → RCHILD = NULL
        endif
        return Node(ptr)
endif
/*DELETION: CASE 2 */
if (case = 2) then                      // when the node contains only one child
        if(parent → LCHILD = ptr)then
                if(ptr → LCHILD = NULL) then
                        parent → LCHILD = ptr → RCHILD
                else
                        parent → LCHILD = ptr → LCHILD
                endif
        else
                if(parent → RCHILD = ptr) then
                        if(ptr →Lchild = NULL) then
                                parent → RCHILD = ptr → LCHILD
                        else
                                parent →Rchild = ptr → LCHILD
                        endif
                endif
        endif
        return Node(ptr)                // Return the deleted node to memory bank
endif
/*DELETION: CASE 3 */
if(case = 3) then
        ptr1 = SUCC(ptr)                //Find the inorder successor of the node
        item1 = ptr1→ DATA
        Delete_BST(item1)               //Delete the inorder successor
        ptr → DATA = item1 //Replace the data with the data of the inorder
successor
endif
stop
```

Algorithm: Preorder_BST_Traversal_Recursively

Input: ROOT is the pointer to the root node of the binary search tree
Output: Visiting all the nodes in preorder fashion.
Data Structure: Linked structure of binary search tree.
Steps:
1. ptr = ROOT
2. if (ptr!=NULL) then
3. visit(ptr)
4. preorder(ptr→LC)
5. preorder(ptr→RC)
6. endif
7. stop

Algorithm: Inorder_BST_Traversal_Recursively
Input: ROOT is the pointer to the root node of the binary search tree
Output: Visiting all the nodes in inorder fashion.
Data Structure: Linked structure of binary search tree.
Steps:
1. ptr = ROOT
2. if (ptr!=NULL) then
3. inorder(ptr->LC)
4. visit(ptr)
5. inorder(ptr->RC)
6. endif
7. Stop

Algorithm: Postorder_BST_Traversal_Recursively
Input: ROOT is the pointer to the root node of the binary search tree
Output: Visiting all the nodes in postorder fashion.
Data Structure: Linked structure of binary search tree.
Steps:
1. ptr = ROOT
2. if (ptr!=NULL) then
3. postorder(ptr->LC)
4. postorder(ptr->RC)
5. visit(ptr)
6. endif
7. Stop

Program: // to perform operations creation, insertion, deletion and traversing on a BST

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

// structure of a node
struct node
{
   int data;
   struct node *left;
   struct node *right;
};

// globally initialized root pointer
struct node *root = NULL;

// function prototyping
struct node *create_node(int);
void insert(int);
struct node *delete (struct node *, int);
int search(int);
void inorder(struct node *);
void postorder();
void preorder();
struct node *smallest_node(struct node *);
struct node *largest_node(struct node *);
int get_data();

int main()
{
   int userChoice;
   int userActive = 'Y';
   int data;
   struct node* result = NULL;

   while (userActive == 'Y' || userActive == 'y')
   {
      printf("\n\n------- Binary Search Tree ------\n");
      printf("\n1. Insert");
      printf("\n2. Delete");
      printf("\n3. Search");
      printf("\n4. Get Larger Node Data");
      printf("\n5. Get smaller Node data");
      printf("\n\n-- Traversals --");
```

```
printf("\n\n6. Inorder ");
printf("\n7. Post Order ");
printf("\n8. Pre Oder ");
printf("\n9. Exit");

printf("\n\nEnter Your Choice: ");
scanf("%d", &userChoice);
printf("\n");

switch(userChoice)
{
   case 1:
      data = get_data();
      insert(data);
      break;

   case 2:
      data = get_data();
      root = delete(root, data);
      break;

   case 3:
      data = get_data();
      if (search(data) == 1)
      {
         printf("\nData was found!\n");
      }
      else
      {
         printf("\nData does not found!\n");
      }
      break;

   case 4:
      result = largest_node(root);
      if (result != NULL)
      {
         printf("\nLargest Data: %d\n", result->data);
      }
      break;

   case 5:
      result = smallest_node(root);
      if (result != NULL)
```

```
        {
          printf("\nSmallest Data: %d\n", result->data);
        }
        break;

      case 6:
        inorder(root);
        break;

      case 7:
        postorder(root);
        break;

      case 8:
        preorder(root);
        break;

      case 9:
        printf("\n\nProgram was terminated\n");
        break;

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

    printf("\n__________\nDo you want to continue? ");
    fflush(stdin);
    scanf(" %c", &userActive);
  }

  return 0;
}

// creates a new node
struct node *create_node(int data)
{
  struct node *new_node = (struct node *)malloc(sizeof(struct node));

  if (new_node == NULL)
  {
    printf("\nMemory for new node can't be allocated");
    return NULL;
  }
```

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

    return new_node;
}

// inserts the data in the BST
void insert(int data)
{
    struct node *new_node = create_node(data);

    if (new_node != NULL)
    {
        // if the root is empty then make a new node as the root node
        if (root == NULL)
        {
            root = new_node;
            printf("\n* node having data %d was inserted\n", data);
            return;
        }

        struct node *temp = root;
        struct node *prev = NULL;

        // traverse through the BST to get the correct position for insertion
        while (temp != NULL)
        {
            prev = temp;
            if (data > temp->data)
            {
                temp = temp->right;
            }
            else
            {
                temp = temp->left;
            }
        }

        // found the last node where the new node should insert
        if (data > prev->data)
        {
            prev->right = new_node;
```

```
        }
        else
        {
            prev->left = new_node;
        }

        printf("\n* node having data %d was inserted\n", data);
    }
}

// deletes the given key node from the BST
struct node *delete (struct node *root, int key)
{
    if (root == NULL)
    {
        return root;
    }
    if (key < root->data)
    {
        root->left = delete (root->left, key);
    }
    else if (key > root->data)
    {
        root->right = delete (root->right, key);
    }
    else
    {
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        struct node *temp = smallest_node(root->right);
        root->data = temp->data;
        root->right = delete (root->right, temp->data);
    }
    return root;
```

```
}

// search the given key node in BST
int search(int key)
{
   struct node *temp = root;

   while (temp != NULL)
   {
      if (key == temp->data)
      {
         return 1;
      }
      else if (key > temp->data)
      {
         temp = temp->right;
      }
      else
      {
         temp = temp->left;
      }
   }
   return 0;
}

// finds the node with the smallest value in BST
struct node *smallest_node(struct node *root)
{
   struct node *curr = root;
   while (curr != NULL && curr->left != NULL)
  {
      curr = curr->left;
   }
   return curr;
}

// finds the node with the largest value in BST
struct node *largest_node(struct node *root)
{
   struct node *curr = root;
   while (curr != NULL && curr->right != NULL)
   {
      curr = curr->right;
```

```
    }
    return curr;
}

// inorder traversal of the BST
void inorder(struct node *root)
{
    if (root == NULL)
    {
        return;
    }
    inorder(root->left);
    printf("%d ",  root->data);
    inorder(root->right);
}

// preorder traversal of the BST
void preorder(struct node *root)
{
    if (root == NULL)
    {
        return;
    }
    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// postorder travsersal of the BST
void postorder(struct node *root)
{
    if (root == NULL)
    {
        return;
    }
    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}

// getting data from the user
int get_data()
{
    int data;
```

```
    printf("\nEnter Data: ");
    scanf("%d", &data);
    return data;
}
```

Output:

```
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 1
Enter Data: 9
* node having data 9 was inserted

__________
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 1
Enter Data: 8
```

```
* node having data 8 was inserted

__________
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 1
Enter Data: 7
* node having data 7 was inserted

__________
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 2
Enter Data: 8
__________
Do you want to continue? y
```

```
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 3
Enter Data: 9
Data was found!

__________
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 3
Enter Data: 8
Data does not found!

__________
Do you want to continue? y
------- Binary Search Tree ------
```

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 4
Largest Data: 9

____
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 5
Smallest Data: 7

____
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data

5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 6
7 9
____
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 7
7 9
____
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order

```
8. Pre Oder
9. Exit

Enter Your Choice: 8
9 7
__________
Do you want to continue? y
------- Binary Search Tree ------

1. Insert
2. Delete
3. Search
4. Get Larger Node Data
5. Get smaller Node data

-- Traversals --

6. Inorder
7. Post Order
8. Pre Oder
9. Exit

Enter Your Choice: 9
Program was terminated

__________
Do you want to continue? n
```

11) Write a program to implement BFS Graph Traversal.

Program Statement:

Write a program to implement Breadth First Search Graph Traversal Algorithm

Algorithm: BFS (Breadth First Search)

Input: V, the starting vertex

Output: A list VISIT giving the order of visit of vertices during traversal

Steps:

1.Define a queue of size total number of vertices in the graph.

2.Select any vertex as starting point for traversal. Visit that vertex and insert it into the queue.

3.Visit all the adjacent vertices of the vertex which is at point of the queue, which is not visited and insert them into the queue.

4.When there is no new vertex to be visit from the vertex at front of the queue then delete that vertex from the queue.

5.Repeat step 3 & 4 until queue becomes empty.

6.When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

Program: // to implement BFS Graph Traversal

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

#define MAX 5

struct Vertex {
  char label;
  bool visited;
};

//queue variables

int queue[MAX];
int rear = -1;
int front = 0;
int queueItemCount = 0;

//graph variables

//array of vertices
struct Vertex* lstVertices[MAX];
```

```
//adjacency matrix
int adjMatrix[MAX][MAX];

//vertex count
int vertexCount = 0;

//queue functions

void insert(int data) {
  queue[++rear] = data;
  queueItemCount++;
}

int removeData() {
  queueItemCount--;
  return queue[front++];
}

bool isQueueEmpty() {
  return queueItemCount == 0;
}

//graph functions

//add vertex to the vertex list
void addVertex(char label) {
  struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
  vertex->label = label;
  vertex->visited = false;
  lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
void addEdge(int start,int end) {
  adjMatrix[start][end] = 1;
  adjMatrix[end][start] = 1;
}

//display the vertex
void displayVertex(int vertexIndex) {
  printf("%c ",lstVertices[vertexIndex]->label);
}
```

```
//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
  int i;

  for(i = 0; i<vertexCount; i++) {
    if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false)
      return i;
  }

  return -1;
}

void breadthFirstSearch() {
  int i;

  //mark first node as visited
  lstVertices[0]->visited = true;

  //display the vertex
  displayVertex(0);

  //insert vertex index in queue
  insert(0);
  int unvisitedVertex;

  while(!isQueueEmpty()) {
    //get the unvisited vertex of vertex which is at front of the queue
    int tempVertex = removeData();

    //no adjacent vertex found
    while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      insert(unvisitedVertex);
    }

  }

  //queue is empty, search is complete, reset the visited flag
  for(i = 0;i<vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
}
```

```
int main() {
  int i, j;

  for(i = 0; i<MAX; i++) { // set adjacency
    for(j = 0; j<MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;
  }

  addVertex('S');   // 0
  addVertex('A');   // 1
  addVertex('B');   // 2
  addVertex('C');   // 3
  addVertex('D');   // 4

  addEdge(0, 1);    // S - A
  addEdge(0, 2);    // S - B
  addEdge(0, 3);    // S - C
  addEdge(1, 4);    // A - D
  addEdge(2, 4);    // B - D
  addEdge(3, 4);    // C - D

  printf("\nBreadth First Search: ");

  breadthFirstSearch();

  return 0;
}
```

Output:

Breadth First Search: S A B C D

12) Write a Program to implement DFS Graph Traversal.

Program Statement:
Write a Program to implement Depth First Search Graph Traversal Algorithm

Algorithm: DFS (Depth First Search)
Input: V is the starting vertex
Output: A list VISIT giving the order of visit of vertices during the traversal
Steps:
1.Define a stack of size total number of vertices in graph.
2.Select any vertex as starting point for traversal. Visit that vertex and push it on to the stack.
3.Visit any one of the adjacent vertex of the vertex which is at top of the stack, which is not visited, and push it on to the stack.
4.Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
5.When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
6.Repeat step 3,4,5 until stack becomes empty.
7.When stack becomes empty, then produce final spanning tree by removing unused edges from the graph.

Program: // to implement DFS Graph Traversal

```
#include <stdio.h>
#include <stdlib.h>
struct node {
 int vertex;
 struct node* next;
};
struct node* createNode(int v);
struct Graph {
 int totalVertices;
 int* visited;
 struct node** adjLists;
};
void DFS(struct Graph* graph, int vertex) {
 struct node* adjList = graph->adjLists[vertex];
 struct node* temp = adjList;
 graph->visited[vertex] = 1;
 printf("%d -> ", vertex);
 while (temp != NULL) {
   int connectedVertex = temp->vertex;
   if (graph->visited[connectedVertex] == 0) {
     DFS(graph, connectedVertex);
   }
   temp = temp->next;
 }
}
struct node* createNode(int v) {
 struct node* newNode = malloc(sizeof(struct node));
 newNode->vertex = v;
 newNode->next = NULL;
 return newNode;
}
struct Graph* createGraph(int vertices) {
 struct Graph* graph = malloc(sizeof(struct Graph));
 graph->totalVertices = vertices;
 graph->adjLists = malloc(vertices * sizeof(struct node*));
 graph->visited = malloc(vertices * sizeof(int));
```

```
  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
void displayGraph(struct Graph* graph) {
  int v;
  for (v = 1; v < graph->totalVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n%d =>  ", v);
    while (temp) {
      printf("%d, ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
  printf("\n");
}
int main() {
  struct Graph* graph = createGraph(8);
  addEdge(graph, 1, 5);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 3);
  addEdge(graph, 3, 6);
  addEdge(graph, 2, 7);
  addEdge(graph, 2, 4);

  printf("\nThe Adjacency List of the Graph is:");
  displayGraph(graph);
  printf("\nDFS traversal of the graph: \n");
  DFS(graph, 1);
  return 0;
}
```

Output:

The Adjacency List of the Graph is:

1 => 3, 2, 5,

2 => 4, 7, 1,

3 => 6, 1,

4 => 2,

5 => 1,

6 => 3,

7 => 2,

DFS traversal of the graph:

1 -> 3 -> 6 -> 2 -> 4 -> 7 -> 5 ->