1.Write a program that uses functions to perform the following operations on singly linkedlist.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

Solution :

C Program to implement Single Linked List

Single Linked List

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

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

int count()
{
    struct node *temp;
    int i=1;
    temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
        i++;
    }
    return(i);
}

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=value;
    temp->next=NULL;
    return temp;
}
void insert_begin(int value)
{
    struct node *newnode;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        newnode->next=head;
```

```
        head=newnode;
    }
}

void insert_end(int value)
{
    struct node *newnode, *temp;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
    }
}

void insert_pos(int value,int pos)
{
    struct node *newnode, *temp1,*temp2;
    int i,c=1;
    newnode=create(value);
    i=count();
    if(pos==1)
        insert_begin(value);
    else if(pos>i+1)
    {
        printf("insertion is not posible");
        return;
    }
    else
    {
        temp1=head;
        while(c<=pos-1 && temp1!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
            c++;
        }
        newnode->next=temp2->next;
        temp2->next=newnode;
    }
}

void delete_begin()
{
    struct node *temp;
    if(head==NULL)
```

```
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}
void delete_end()
{
    struct node *temp1,*temp2;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp1=head;
        while(temp1->next!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
        }
        temp2->next=NULL;
        free(temp1);
    }
}
void delete_pos(int pos)
{
    struct node *temp1,*temp2;
    int i,c=1;
    i=count();
    if(pos==1)
        delete_begin();
    else if(pos>i)
    {
        printf("Deletion is not posible");
        return;
    }
    else
    {
        temp1=head;
        while(c<=pos && temp1->next!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
            c++;
        }
        temp2->next=temp1->next;
        free(temp1);
    }
}
```

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

void main()
{
    int ch,pos,value;
    do
    {
        printf("\n1.Insert Begin\n2.Insert End\n3.Insert Position\n4.Delete 
Begin\n5.Delete End\n6.Delete Position\n7.Display\n8.Exit\n");
        printf("enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("enter the value:");
                scanf("%d",&value);
                insert_begin(value);
                break;
        case 2: printf("enter value:");
                scanf("%d",&value);
                insert_end(value);
                break;
        case 3: printf("enter value:");
                scanf("%d",&value);
                printf("enter position you want to insert: ");
                scanf("%d",&pos);
                insert_pos(value,pos);
                break;
        case 4: delete_begin();
                break;
        case 5: delete_end();
                break;
        case 6: printf("enter position you want to delete: ");
                scanf("%d",&pos);
                delete_pos(pos);
                break;
        case 7: display();
                break;
```

```
        case 8:break;
        default: printf("\nyour choice is wrong!.. ");
        }
    }while(ch!=8);
}
```

OUTPUT

```
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:1
enter the value:10
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:2
enter value:30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:3
enter value:20
enter position you want to insert: 2
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:7
10-> 20-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
```

```
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:3
enter value:40
enter position you want to insert: 4
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:7
10-> 20-> 30-> 40
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:4
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:5
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:7
20-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:6
```

```
enter position you want to delete: 2
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:7
20
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
enter your choice:8
```

2.Write a program that uses functions to perform the following operations on doubly linkedlist.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

Solution :

C Program to implement Doubly Linked List

Doubly Linked List

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

struct node{
    struct node *llink;
    int data;
    struct node *rlink;
}*head=NULL,*tail=NULL;

int count()
{
    struct node *temp;
    int i=1;
    temp=head;
    while(temp->rlink!=NULL)
    {
        temp=temp->rlink;
        i++;
    }
    return(i);
}

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=value;
    temp->rlink=NULL;
    temp->llink=NULL;
    return temp;
}
void insert_begin(int value)
{
    struct node *newnode;
    newnode=create(value);
    if(head==NULL)
    {
        head=tail=newnode;
    }
    else
    {
        newnode->rlink=head;
        head->llink=newnode;
```

```
        head=newnode;
    }
}

void insert_end(int value)
{
    struct node *newnode, *temp;
    newnode=create(value);
    if(head==NULL)
    {
        head=tail=newnode;
    }
    else
    {
        newnode->llink=tail;
        tail->rlink=newnode;
        tail=newnode;
    }
}

void insert_pos(int value,int pos)
{
    struct node *newnode, *temp1,*temp2,*temp;
    int i,c=1;
    newnode=create(value);
    i=count();
    if(pos==1)
        insert_begin(value);
    else if(pos>i+1)
    {
        printf("insertion is not posible");
        return;
    }
    else if(pos==i+1)
    {
        insert_end(value);
    }
    else
    {
        temp=head;
        while(c<=pos-1 && temp!=NULL)
        {
            temp=temp->rlink;
            c++;
        }
        temp1=temp->llink;
        temp1->rlink=newnode;
        temp->llink=newnode;
        newnode->llink=temp1;
        newnode->rlink=temp;
    }
}

void delete_begin()
```

```
{
    struct node *temp;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=head;
        head=head->rlink;
        if(head==NULL)
            tail=NULL;
        else
            head->llink=NULL;
        free(temp);
    }
}
void delete_end()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=tail;
        tail=tail->llink;
        if(tail==NULL)
            head=NULL;
        else
            tail->rlink=NULL;
        free(temp);
    }
}
void delete_pos(int pos)
{
    struct node *temp1,*temp2,*temp;
    int i,c=1;
    i=count();
    if(pos==1)
        delete_begin();
    else if(pos>i)
    {
        printf("Deletion is not posible");
        return;
    }
    else if(pos==i)
    {
        delete_end();
    }
    else
    {
        temp=head;
```

```
        while(c<pos && temp->rlink!=NULL)
        {
            temp=temp->rlink;
            c++;
        }
        temp1=temp->llink;
        temp2=temp->rlink;
        temp1->rlink=temp2;
        temp2->llink=temp1;
        free(temp);
    }
}
void display()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("list is empty");
    }
    else
    {
        temp=head;
        while(temp->rlink!=NULL)
        {
            printf("%d <-> ",temp->data);
            temp=temp->rlink;
        }
        printf("%d",temp->data);
    }
}

void main()
{
    int ch,pos,value;
    do
    {
        printf("\n1.Insert Begin\n2.Insert End\n3.Insert Position\n4.Delete
Begin\n5.Delete End\n6.Delete Position\n7.Display\n8.Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("Enter the value: ");
                scanf("%d",&value);
                insert_begin(value);
                break;
        case 2: printf("Enter value: ");
                scanf("%d",&value);
                insert_end(value);
                break;
        case 3: printf("Enter value: ");
                scanf("%d",&value);
                printf("Enter position you want to insert: ");
                scanf("%d",&pos);
```

```
                insert_pos(value,pos);
                break;
        case 4: delete_begin();
                break;
        case 5: delete_end();
                break;
        case 6: printf("Enter position you want to delete: ");
                scanf("%d",&pos);
                delete_pos(pos);
                break;
        case 7: display();
                break;
        case 8:break;
        default: printf("\nyour choice is wrong!.. ");
        }
    }while(ch!=8);
}
```

OUTPUT

```
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 1
Enter the value: 10
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 2
Enter value: 20
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7 10 <-> 20 1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
```

```
6.Delete Position
7.Display
8.Exit
Enter your choice: 3
Enter value: 30
Enter position you want to insert: 2
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 30 <-> 20
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 3
Enter value: 40
Enter position you want to insert: 4
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 30 <-> 20 <-> 40
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 6
Enter position you want to delete: 2
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
```

```
8.Exit
Enter your choice: 7
10 <-> 20 <-> 40
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 4
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 5
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7 20
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 9
your choice is wrong!..
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 8
```

3.Write a program that uses functions to perform the following operations on circular linkedlist.:
i) Creation ii) Insertion iii) Deletion iv) Traversal

Solution :

C Program to implement Circular Linked List

Circular Linked List

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

struct node{
    struct node *llink;
    int data;
    struct node *rlink;
}*head=NULL,*tail=NULL;

int count()
{
    struct node *temp;
    int i=1;
    temp=head;
    while(temp->rlink!=head)
    {
        temp=temp->rlink;
        i++;
    }
    return(i);
}

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=value;
    temp->rlink=temp;
    temp->llink=temp;
    return temp;
}
void insert_begin(int value)
{
    struct node *newnode;
    newnode=create(value);
    if(head==NULL)
    {
        head=tail=newnode;
    }
    else
    {
        newnode->llink=tail;
        newnode->rlink=head;
```

```
        head->llink=newnode;
        tail->rlink=newnode;
        head=newnode;
    }
}

void insert_end(int value)
{
    struct node *newnode, *temp;
    newnode=create(value);
    if(head==NULL)
    {
        head=tail=newnode;
    }
    else
    {
        newnode->rlink=head;
        newnode->llink=tail;
        tail->rlink=newnode;
        head->llink=newnode;
        tail=newnode;
    }
}

void insert_pos(int value,int pos)
{
    struct node *newnode, *temp1,*temp2,*temp;
    int i,c=1;
    i=count();
    if(pos==1)
        insert_begin(value);
    else if(pos>i+1)
    {
        printf("insertion is not posible");
        return;
    }
    else if(pos==i+1)
    {
        insert_end(value);
    }
    else
    {
        newnode=create(value);
        temp=head;
        while(c<pos )
        {
            temp=temp->rlink;
            c++;
        }
        temp1=temp->llink;
        temp1->rlink=newnode;
        temp->llink=newnode;
        newnode->llink=temp1;
        newnode->rlink=temp;
```

```
    }
}

void delete_begin()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=head;
        head=head->rlink;
        if(head==tail)
            head=tail=NULL;
        else
        {
            head->llink=tail;
            tail->rlink=head;
        }
        free(temp);
    }
}
void delete_end()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=tail;
        if(tail==head)
        {
            head=tail=NULL;
        }else
        {
            tail=tail->llink;
            tail->rlink=head;
            head->llink=tail;
        }
        free(temp);
    }
}
void delete_pos(int pos)
{
    struct node *temp1,*temp2,*temp;
    int i,c=1;
    i=count();
    if(pos==1)
        delete_begin();
    else if(pos>i)
```

```
    {
        printf("Deletion is not posible");
        return;
    }
    else if(pos==i)
    {
        delete_end();
    }
    else
    {
        temp=head;
        while(c<pos)
        {
            temp=temp->rlink;
            c++;
        }
        temp1=temp->llink;
        temp2=temp->rlink;
        temp1->rlink=temp2;
        temp2->llink=temp1;
        free(temp);
    }
}
void display()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("list is empty");
    }
    else
    {
        temp=head;
        while(temp!=tail)
        {
            printf("%d <-> ",temp->data);
            temp=temp->rlink;
        }
        printf("%d",temp->data);
    }
}

void main()
{
    int ch,pos,value;
    do
    {
        printf("\n1.Insert Begin\n2.Insert End\n3.Insert Position\n4.Delete 
Begin\n5.Delete End\n6.Delete Position\n7.Display\n8.Exit\n");
        printf("Enter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("Enter the value: ");
```

```c
                scanf("%d",&value);
                insert_begin(value);
                break;
        case 2: printf("Enter value: ");
                scanf("%d",&value);
                insert_end(value);
                break;
        case 3: printf("Enter value: ");
                scanf("%d",&value);
                printf("Enter position you want to insert: ");
                scanf("%d",&pos);
                insert_pos(value,pos);
                break;
        case 4: delete_begin();
                break;
        case 5: delete_end();
                break;
        case 6: printf("Enter position you want to delete: ");
                scanf("%d",&pos);
                delete_pos(pos);
                break;
        case 7: display();
                break;
        case 8:break;
        default: printf("\nyour choice is wrong!.. ");
        }
    }while(ch!=8);
}
```

OUTPUT

```
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 1
Enter the value: 10
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 3
Enter value: 20
Enter position you want to insert: 2
1.Insert Begin
```

```
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 20
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 2
Enter value: 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 20 <-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 3
Enter value: 40
Enter position you want to insert: 3
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 20 <-> 40 <-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
```

```
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 6
Enter position you want to delete: 3
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
10 <-> 20 <-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 4
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
20 <-> 30
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 5
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 7
1.Insert Begin
```

```
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 9
your choice is wrong!..
1.Insert Begin
2.Insert End
3.Insert Position
4.Delete Begin
5.Delete End
6.Delete Position
7.Display
8.Exit
Enter your choice: 8
```

4.i)Write a program that implement Stack (its operations) using Array

Solution :

C Program that implement Stack (its operations) using Array

Stack using Array

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

#define MAX_SIZE 5

int stack[MAX_SIZE],top=-1;

// Function to check if the stack is empty
bool isEmpty() {
    return top == -1;
}

// Function to add an item to the stack
void push(int item) {
    if (top == MAX_SIZE - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = item;
    }
}

// Function to remove an item from the stack
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow\n");
        return -1; // Indicating underflow
    } else {
        return stack[top--];
    }
}

// Function to get the top item of the stack
int peek() {
    if (isEmpty()) {
        printf("Stack is Empty\n");
        return -1; // Indicating empty stack
    } else {
        return stack[top];
    }
}

// Function to show all the items from stack
void show()
{
```

```c
    int i;
    if (isEmpty())
        printf("Stack is Empty\n");
    else{
    for(i=top;i>-1;i--)
        printf("%d\n",stack[i]);
    }
}

// Main function
int main() {
    int ch,data;
    do{
        printf("\n1. Push\n2. Pop\n3. Peek\n4. Show\n5. Exit");
        printf("\nEnter your choice:   ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("Enter data to push:  ");
                    scanf("%d",&data);
                    push(data);
                    break;
            case 2: printf("Popped: %d\n", pop());
                    break;
            case 3: printf("Top element: %d\n", peek());
                    break;
            case 4: show();
                    break;
            case 5: break;
            default: printf("Enter valid choice");
        }
    }while(ch!=5);
    return 0;
}
```

OUTPUT

```
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  10
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  20
1. Push
2. Pop
```

```
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  30
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  40
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   3
Top element: 40

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   4
40
30
20
10

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  50
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   1
Enter data to push:  60
Stack Overflow

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   4
50
```

```
40
30
20
10

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Popped: 50

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Popped: 40

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Popped: 30

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Popped: 20

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Popped: 10

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   2
Stack Underflow
Popped: -1
```

```
1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   4
Stack is Empty

1. Push
2. Pop
3. Peek
4. Show
5. Exit
Enter your choice:   5
```

4.ii).Write a program that implement Stack (its operations) using Linked List (Pointer)

Solution :

C Program to implement Stack using Linked List(Pointer)

Stack using Linked List

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

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

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=value;
    temp->next=NULL;
    return temp;
}

void push(int value)
{
    struct node *newnode;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        newnode->next=head;
        head=newnode;
    }
}

void pop()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("Stack is underflow");
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
```

```
}

void show()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("Stack is empty");
    }
    else
    {
        temp=head;
        while(temp!=NULL)
        {
            printf("%d\n",temp->data);
            temp=temp->next;
        }
    }
}

void main()
{
    int ch,pos,value;
    do
    {
        printf("\n1. Push\n2. Pop\n3. Show\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("\nEnter the value: ");
                scanf("%d",&value);
                push(value);
                break;
        case 2: pop();
                break;
        case 3: show();
                break;
        case 4:break;
        default: printf("\nyour choice is wrong!.. ");
        }
    }while(ch!=4);
}
```

OUTPUT

```
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 1
Enter the value: 10
1. Push
```

```
2. Pop
3. Show
4. Exit
Enter your choice: 1
Enter the value: 20
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 1
Enter the value: 30
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 3
30
20
10

1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 2
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 3
20
10

1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 2
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 2
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 2
Stack is underflow
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 3
```

```
Stack is empty
1. Push
2. Pop
3. Show
4. Exit
Enter your choice: 4
```

5.i).Write a program that implement Queue(its operations) using Array

Solution :

C Program to implement Queue (its operations) using Array

Queue using Array

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

#define MAX_SIZE 5

int queue[MAX_SIZE],front=-1, rear=-1;

// Function to add an element to the queue
void enqueue(int element)
{
    if (rear == MAX_SIZE - 1)
        {
        printf("Queue Overflow\n");
    }
        else
        {
                if(front == -1)
                        front = 0;
                queue[++rear] = element;
    }
}

// Function to remove an element from the queue
int dequeue()
{
        int item;
    if (front == -1)
        {
                printf("Queue Underflow\n");
        return -1; // Indicating underflow
    }
        else
        {
        int item = queue[front++];
        if (front > rear)   // Queue is now empty
                {
            front = -1;
            rear = -1;
        }
        return item;
    }
}

// Function to display all the items from Queue
void display()
```

```
{
    int i;
    if (front == -1)
        {
                printf("Queue is Empty\n");
    }
    else
    {
        for(i=front;i<=rear;i++)
            printf("%d\t",queue[i]);
    }
}

// Main function
int main() {
    int ch,data;
    do{
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit");
        printf("\nEnter your choice:   ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("Enter data to insert:  ");
                    scanf("%d",&data);
                    enqueue(data);
                    break;
            case 2: printf("Deleted: %d\n", dequeue());
                    break;
            case 3: display();
                    break;
            case 4: break;
            default: printf("your choice is wrong!..");
        }
    }while(ch!=4);
    return 0;
}
```

OUTPUT

```
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  10
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  20
1. Insert
2. Delete
```

```
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
10      20      30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  40
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  50
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
10      20      30      40      50
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  60
Queue Overflow

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
Deleted: 10

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
Deleted: 20

1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter your choice:   2
Deleted: 30

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
40      50
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
Deleted: 40

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
Deleted: 50

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
Queue Underflow
Deleted: -1

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
Queue is Empty

1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   4
```

5.ii).Write a program that implement Queue (its operations) using Linked List (Pointer)

Solution :

C Program to implement Queue using Linked List(pointer)

Queue using Linked List

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

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

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=value;
    temp->next=NULL;
    return temp;
}

void enqueue(int value)
{
    struct node *newnode, *temp;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
    }
}

void dequeue()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("Queue Underflow");
    }
    else
```

```
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}

void display()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("Queue is empty");
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            printf("%d, ",temp->data);
            temp=temp->next;
        }
        printf("%d",temp->data);
    }
}

void main()
{
    int ch,pos,value;
    do
    {
       printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit");
        printf("\nEnter your choice:   ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: printf("Enter data to insert:  ");
                scanf("%d",&value);
                enqueue(value);
                break;
        case 2: dequeue();
                break;
        case 3: display();
                break;
        case 4:break;
        default: printf("\nyour choice is wrong!..");
        }
    }while(ch!=4);
}
```

OUTPUT

```
1. Insert
```

```
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  10
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  20
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   1
Enter data to insert:  30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
10, 20, 30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
20, 30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
30
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   2
1. Insert
2. Delete
3. Display
4. Exit
```

```
Enter your choice:   2
Queue Underflow
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   3
Queue is empty
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:   4
```

6.i).Write a program that implements Quick sort sorting methods to sort a given list of integers in ascending order

Solution :

C program that implements Quick sort sorting methods to sort a given list of integers in ascending order

Quick sort

```
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void quicksort(int number[25],int first,int last)
{
    int i, j, pivot, temp;
    if(first<last)
    {
        pivot=first; // Choose the first element as pivot
        i=first;
        j=last;
        while(i<j)
        {
            while(number[i]<=number[pivot]&&i<last)
            i++;
            while(number[j]>number[pivot])
            j--;
            if(i<j) // swap two elements
            {
                swap(&number[i], &number[j]);
            }
        }
                 // Swap the pivot element with the element at i+1 position
                 swap(&number[pivot], &number[j]);
                 // Recursive call on the left of pivot
        quicksort(number,first,j-1);
                 // Recursive call on the right of pivot
        quicksort(number,j+1,last);
    }
}
int main()
{
    int i, count, number[25];
    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);
    for(i=0;i<count;i++)
    {
```

```
        printf("\nEnter %d element: ", i+1);
        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

```
How many elements are u going to enter?: 10
Enter 1 element: 3
Enter 2 element: 6
Enter 3 element: 9
Enter 4 element: 8
Enter 5 element: 5
Enter 6 element: 2
Enter 7 element: 1
Enter 8 element: 4
Enter 9 element: 7
Enter 10 element: 10
Order of Sorted elements:  1 2 3 4 5 6 7 8 9 10
```

6.ii).Write a program that implements Merge sort sorting methods to sort a given list of integers in ascending order

Solution :

C program that implements Merge sort sorting methods to sort a given list of integers in ascending order

Merge sort

```
 #include <stdio.h>
void merge(int A[], int mid, int low, int high)
{
    int i, j, k, B[100];
    i = low;
    j = mid + 1;
    k = low;

    while (i <= mid && j <= high)
    {
        if (A[i] < A[j])
        {
            B[k] = A[i];
            i++;
            k++;
        }
        else
        {
            B[k] = A[j];
            j++;
            k++;
        }
    }
    while (i <= mid)
    {
        B[k] = A[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        B[k] = A[j];
        k++;
        j++;
    }
        // It will copy data from temporary array to array
    for (int i = low; i <= high; i++)
    {
        A[i] = B[i];
    }
}

void mergeSort(int number[], int low, int high)
```

```c
{
    int mid;
    if(low<high)
        {
                // finding the mid value of the array.
        mid = (low + high) /2;
                // Calling the merge sort for the first half
        mergeSort(number, low, mid);
                // Calling the merge sort for the second half
        mergeSort(number, mid+1, high);
                // Calling the merge function
        merge(number, mid, low, high);
    }
}

int main()
{
    int i, count, number[25];
    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);
    for(i=0;i<count;i++)
    {
        printf("\nEnter %d element: ", i+1);
        scanf("%d",&number[i]);
    }
    mergeSort(number,0,count-1);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
    printf(" %d",number[i]);
    return 0;
}
```

OUTPUT

```
How many elements are u going to enter?: 10
Enter 1 element: 10
Enter 2 element: 1
Enter 3 element: 4
Enter 4 element: 7
Enter 5 element: 8
Enter 6 element: 5
Enter 7 element: 2
Enter 8 element: 3
Enter 9 element: 6
Enter 10 element: 9
Order of Sorted elements:  1 2 3 4 5 6 7 8 9 10
```

6.iii).Write a program that implements Heap sort sorting methods to sort a given list of integers in ascending order

Solution :

C program that implements Heap sort sorting methods to sort a given list of integers in ascending order

Heap sort

```
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

/* heapify the subtree with root i */
void heapify(int* arr, int n, int i)
{
    // store largest as the root element
    int largest = i;
    int left = 2 * i + 1;
    int right  = 2 * i + 2;
    // now check whether the right and left right is larger than the root or not
    if (left < n && arr[left] > arr[largest])
    {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest])
    {
        largest = right;
    }
    // if the root is smaller than the children then swap it with the largest
children's value
    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);
        // again heapify that side of the heap where the root has gone
        heapify(arr, n, largest);
    }
}

/* sorts the given array of n size */
void heapsort(int* arr, int n)
{
    // build the binary max heap
    for (int i = n / 2 - 1; i >= 0; i--)
    {
        heapify(arr, n, i);
    }
```

```c
    // sort the max heap
    for (int i = n - 1; i >= 0; i--)
    {
        // swap the root node and the last leaf node
        swap(&arr[i], &arr[0]);
        // again heapify the max heap from the root
        heapify(arr, i, 0);
    }
}

int main()
{
    int i, count, number[25];
    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);
    for(i=0;i<count;i++)
    {
        printf("\nEnter %d element: ", i+1);
        scanf("%d",&number[i]);
    }
    heapsort(number,count);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
    printf(" %d",number[i]);
    return 0;
}
```

OUTPUT

```
How many elements are u going to enter?: 10
Enter 1 element: 2
Enter 2 element: 5
Enter 3 element: 8
Enter 4 element: 10
Enter 5 element: 3
Enter 6 element: 1
Enter 7 element: 4
Enter 8 element: 6
Enter 9 element: 7
Enter 10 element: 9
Order of Sorted elements:  1 2 3 4 5 6 7 8 9 10
```

7.i).Write a program to implement the tree traversal methods using Recursive

Solution :

C program to implement the tree traversal methods using Recursive

Tree Traversal using Recursive

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

// Definition of a node in a binary tree
 struct Node {
    int data;
    struct Node *left;
    struct Node *right;
}*root=NULL;

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void preOrder(struct Node* root)
{
   if(root != NULL) {
      printf("%d ",root->data);
      preOrder(root->left);
      preOrder(root->right);
   }
}

void inOrder(struct Node* root)
{
   if(root != NULL) {
      inOrder(root->left);
      printf("%d ",root->data);
      inOrder(root->right);
   }
}

void postOrder(struct Node* root)
{
   if(root != NULL) {
      postOrder(root->left);
```

```c
        postOrder(root->right);
        printf("%d ", root->data);
    }
}


int main() {
    // Constructing a binary tree
    //        1
    //       / \
    //      2   3
    //     / \
    //    4   5
    root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    /* Traversals
    printf("Pre-order traversal: ");
    preOrder(root);
    printf("\n"); */

    printf("In-order traversal: ");
    inOrder(root);
    printf("\n");

   /* printf("Post-order traversal: ");
    postOrder(root);
    printf("\n"); */

    return 0;
}
```

OUTPUT

```
Pre-order traversal: 1 2 4 5 3
In-order traversal: 4 2 5 1 3
Post-order traversal: 4 5 2 3 1
```

7.ii).Write a program to implement the tree traversal methods using Non Recursive

Solution :

C program to implement the tree traversal methods using Recursive

Tree Traversal using Recursive

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

// Definition of a node in a binary tree
struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
}*root=NULL;
int top=-1;
struct Node *s[40];

//push into stack
int push(struct Node *x)
{
    s[++top]=x;
}

//pop from stack
struct Node* pop()
{
    struct Node *x=s[top--];
    return(x);
}

// Function to create a new node
struct Node* createNode(int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void preOrder(struct Node *root)
{
    struct Node *ptr;
    ptr=root;
    if(root==NULL){
```

```
        printf("\nTree is empty");
    }
    else
    {
        push(root);
        while(top!=-1)
        {
            ptr=pop();
            if(ptr!=NULL)
            {
               printf("%d ",ptr->data);
               push(ptr->right);
               push(ptr->left);
            }
        }
    }
}


void inOrder(struct Node *root)
{
    struct Node *ptr;
    ptr=root;
    if(root==NULL)
    {
        printf("\nTree is empty");
    }
    else
    {
        while(top!=-1||ptr!=NULL)
        {
            if(ptr!=NULL)
            {
                push(ptr);
                ptr=ptr->left;
            }
            else{
                ptr=pop();
                printf("%d ",ptr->data);
                ptr=ptr->right;
            }
        }
    }
}


void postOrder(struct Node *root)
{
    struct Node *ptr,*temp;
    ptr=root;
    temp=NULL;
    if(root==NULL)
    {
        printf("\nTree is empty");
```

```
    }
    else{
        while(ptr->left!=NULL)
        {
           push(ptr);
           ptr=ptr->left;
        }
        while(ptr!=NULL){
            if(ptr->right==NULL||ptr->right==temp)
            {
                printf("%d ",ptr->data);
                temp=ptr;
                ptr=pop();
            }
            else
            {
                push(ptr);
                ptr=ptr->right;
                while(ptr->left!=NULL)
                {
                    push(ptr);
                    ptr=ptr->left;
                }
            }
        }
    }
}

int main() {
    // Constructing a binary tree
    //       1
    //      / \
    //     2   3
    //    / \
    //   4   5
    root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    /* Traversals
    printf("Pre-order traversal: ");
    preOrder(root);
    printf("\n"); */

    printf("In-order traversal: ");
    inOrder(root);
    printf("\n");

   /* printf("Post-order traversal: ");
    postOrder(root);
    printf("\n"); */
```

```
    return 0;
}
```

OUTPUT

```
Pre-order traversal: 1 2 4 5 3
In-order traversal: 4 2 5 1 3
Post-order traversal: 4 5 2 3 1
```

8.i).Write a program to implement Binary Search Tree (its operations)

Solution :

C program to implement the Binary Search Tree

Binary Search Tree

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

struct node
{
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item)
{
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root)
{
    if (root != NULL)
    {   // Traverse left
        inorder(root->left);
        // Traverse root
        printf("%d -> ", root->key);
        // Traverse right
        inorder(root->right);
    }
}

// preorder Traversal
void preorder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d -> ", root->key);
        preorder(root->left);
        preorder(root->right);
    }
}

// postorder Traversal
```

```c
void postorder(struct node *root)
{
    if (root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d -> ", root->key);
  }
}

// Insert a node
struct node *insert(struct node *node, int key)
{   // Return a new node if the tree is empty
    if (node == NULL)
        return newNode(key);
    // Traverse to the right place and insert the node
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node)
{
    struct node *current = node;
    // Find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key)
{   // Return if the tree is empty
    if (root == NULL)
        return root;
    // Find the node to be deleted
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else
    {
        // If the node is with only one child or no child
        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;
        }
        // If the node has two children
        struct node *temp = minValueNode(root->right);
        // Place the inorder successor in position of the node to be deleted
        root->key = temp->key;
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// Function to free memory by deallocating nodes
void freeMemory(struct node *root)
{
    if (root == NULL)
        return;
    freeMemory(root->left);
    freeMemory(root->right);
    free(root);
}

int main() {
    int choice,value;
  struct node *root = NULL;
  do
    {
        printf("\n1. Insertion\n2. Deletion\n3. inorder\n4. preorder\n5. 
postorder\n6. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                root = insert(root, value);
                break;
        case 2: printf("Enter the value to be deleted: ");
                scanf("%d",&value);
                root = deleteNode(root, value);
                break;
        case 3: inorder(root);
                break;
        case 4: preorder(root);
                break;
        case 5: postorder(root);
                break;
        case 6: freeMemory(root);
                break;
        default: printf("\nWrong selection!!! Try again!!!");
        }
    }while(choice!=6);
```

```
    return (0);
}
```

OUTPUT

```
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 50
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 20
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 80
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 5
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 30
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1Enter the value to be insert: 65 1. Insertion
2. Deletion
```

```
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 1
Enter the value to be insert: 90
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 3
5 -> 20 -> 30 -> 50 -> 65 -> 80 -> 90 ->
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 4
50 -> 20 -> 5 -> 30 -> 80 -> 65 -> 90 ->
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 5
5 -> 30 -> 20 -> 65 -> 90 -> 80 -> 50 ->
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 2
Enter the value to be deleted: 50
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 4
65 -> 20 -> 5 -> 30 -> 80 -> 90 ->
1. Insertion
2. Deletion
3. inorder
4. preorder
5. postorder
6. Exit
Enter your choice: 6
```

8.iv).Write a program to implement AVL Tree (its operations)

Solution :

C program to implement the AVL Tree

AVL Tree

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

// Structure for a node in the AVL tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height; // Height of the node
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    newNode->height = 1; // Initialize height as 1 for a new node
    return newNode;
}

// Function to calculate the height of a node
int getHeight(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to perform right rotation
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;
```

```
    // Update heights
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}

// Function to perform left rotation
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}

// Function to get the balance factor of a node
int getBalanceFactor(struct Node* node) {
    if (node == NULL)
        return 0;
    return getHeight(node->left) - getHeight(node->right);
}

// Function to insert a node into the AVL tree
struct Node* insert(struct Node* root, int data) {
    if (root == NULL)
        return createNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    else
        return root; // Duplicate keys are not allowed

    // Update height of current node
    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    // Get the balance factor to check if rotation is needed
    int balance = getBalanceFactor(root);

    // Left-Left case (LL)
    if (balance > 1 && data < root->left->data)
        return rightRotate(root);

    // Right-Right case (RR)
    if (balance < -1 && data > root->right->data)
```

```c
        return leftRotate(root);

    // Left-Right case (LR)
    if (balance > 1 && data > root->left->data) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right-Left case (RL)
    if (balance < -1 && data < root->right->data) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function to find the node with the minimum value in the tree
struct Node* findMinValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a node from the AVL tree
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL)
        return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    else {
        // Node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node* temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else // One child case
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        } else {
            // Node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct Node* temp = findMinValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->data = temp->data;
```

```c
            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->data);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // Update height of current node
    root->height = 1 + max(getHeight(root->left), getHeight(root->right));

    // Get the balance factor to check if rotation is needed
    int balance = getBalanceFactor(root);

    // Left-Left case (LL)
    if (balance > 1 && getBalanceFactor(root->left) >= 0)
        return rightRotate(root);

    // Left-Right case (LR)
    if (balance > 1 && getBalanceFactor(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right-Right case (RR)
    if (balance < -1 && getBalanceFactor(root->right) <= 0)
        return leftRotate(root);

    // Right-Left case (RL)
    if (balance < -1 && getBalanceFactor(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function for in-order traversal of the AVL tree
void inOrderTraversal(struct Node* root)
{
   if(root != NULL) {
      inOrderTraversal(root->left);
      printf("%d ",root->data);
      inOrderTraversal(root->right);
   }
}
// Function to free memory by deallocating nodes
void freeMemory(struct Node* root) {
    if (root == NULL)
        return;
    freeMemory(root->left);
    freeMemory(root->right);
```

```c
    free(root);
}

int main() {
    int choice,value;
    struct Node* root = NULL;
    do
    {
        printf("\n1. Insertion\n2. Deletion\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                root = insert(root, value);
                break;
        case 2: printf("Enter the value to be deleted: ");
                scanf("%d",&value);
                root = deleteNode(root, value);
                break;
        case 3: inOrderTraversal(root);
                break;
        case 4: freeMemory(root);
                break;
        default: printf("\nWrong selection!!! Try again!!!");
        }
    }while(choice!=4);
    return 0;
}
```

OUTPUT

```
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 1
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 2
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 3
1. Insertion
2. Deletion
```

```
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 4
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 5
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 6
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
1 2 3 4 5 6
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2
Enter the value to be deleted: 2
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2
Enter the value to be deleted: 3
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2
Enter the value to be deleted: 1
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
4 5 6
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

8.v).Write a program to implement Red - Black Tree (its operations)

Solution :

C program to implement the Red Black Tree

Red Black Tree

```
// Red Black Tree operations in C

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

// Red-Black Tree Node Structure
typedef struct Node {
    int data;
    struct Node* parent;
    struct Node* left;
    struct Node* right;
    int color; // 0 for black, 1 for red
} Node;

// Red-Black Tree Structure
typedef struct RedBlackTree {
    Node* root;
} RedBlackTree;

// Create a new Red-Black Tree Node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->parent = newNode->left = newNode->right = NULL;
    newNode->color = 1; // New nodes are initially red
    return newNode;
}

// Create a new Red-Black Tree
RedBlackTree* createRedBlackTree() {
    RedBlackTree* newTree = (RedBlackTree*)malloc(sizeof(RedBlackTree));
    if (newTree == NULL) {
        printf("Memory allocation error\n");
        exit(1);
    }
    newTree->root = NULL;
    return newTree;
}

// Function to perform left rotation
void leftRotate(RedBlackTree* tree, Node* x) {
```

```
    Node* y = x->right;
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        tree->root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}

// Function to perform right rotation
void rightRotate(RedBlackTree* tree, Node* y) {
    Node* x = y->left;
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    x->parent = y->parent;
    if (y->parent == NULL)
        tree->root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    x->right = y;
    y->parent = x;
}

// Function to fix the Red-Black Tree properties after insertion
void insertFixup(RedBlackTree* tree, Node* z) {
    while (z->parent != NULL && z->parent->color == 1) {
        if (z->parent == z->parent->parent->left) {
            Node* y = z->parent->parent->right;
            if (y != NULL && y->color == 1) {
                z->parent->color = 0; // Black
                y->color = 0; // Black
                z->parent->parent->color = 1; // Red
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    leftRotate(tree, z);
                }
                z->parent->color = 0; // Black
                z->parent->parent->color = 1; // Red
                rightRotate(tree, z->parent->parent);
            }
        } else {
            Node* y = z->parent->parent->left;
            if (y != NULL && y->color == 1) {
```

```
                z->parent->color = 0; // Black
                y->color = 0; // Black
                z->parent->parent->color = 1; // Red
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rightRotate(tree, z);
                }
                z->parent->color = 0; // Black
                z->parent->parent->color = 1; // Red
                leftRotate(tree, z->parent->parent);
            }
        }
    }
    tree->root->color = 0; // Root must be black
}

// Function to insert a node into the Red-Black Tree
void insert(RedBlackTree* tree, int data) {
    Node* z = createNode(data);
    Node* y = NULL;
    Node* x = tree->root;

    while (x != NULL) {
        y = x;
        if (z->data < x->data)
            x = x->left;
        else
            x = x->right;
    }

    z->parent = y;
    if (y == NULL)
        tree->root = z;
    else if (z->data < y->data)
        y->left = z;
    else
        y->right = z;

    insertFixup(tree, z);
}

// Function to find the minimum value node in the tree rooted at a given node
Node* findMinValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Function to fix the Red-Black Tree properties after deletion
void deleteFixup(RedBlackTree* tree, Node* x) {
    while (x != tree->root && x->color == 0) {
```

```c
        if (x == x->parent->left) {
            Node* w = x->parent->right;
            if (w->color == 1) {
                w->color = 0; // Change sibling to black
                x->parent->color = 1; // Change parent to red
                leftRotate(tree, x->parent);
                w = x->parent->right;
            }
            if (w->left->color == 0 && w->right->color == 0) {
                w->color = 1; // Change sibling to red
                x = x->parent; // Move up the tree
            } else {
                if (w->right->color == 0) {
                    w->left->color = 0; // Change sibling's left child to black
                    w->color = 1; // Change sibling to red
                    rightRotate(tree, w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = 0; // Change parent to black
                w->right->color = 0; // Change sibling's right child to black
                leftRotate(tree, x->parent);
                x = tree->root; // This is to exit the loop
            }
        } else {
            // Same as then clause with "right" and "left" exchanged
            Node* w = x->parent->left;
            if (w->color == 1) {
                w->color = 0;
                x->parent->color = 1;
                rightRotate(tree, x->parent);
                w = x->parent->left;
            }
            if (w->right->color == 0 && w->left->color == 0) {
                w->color = 1;
                x = x->parent;
            } else {
                if (w->left->color == 0) {
                    w->right->color = 0;
                    w->color = 1;
                    leftRotate(tree, w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = 0;
                w->left->color = 0;
                rightRotate(tree, x->parent);
                x = tree->root;
            }
        }
    }
    x->color = 0; // Ensure the root is black
}
```

```c
// Transplant helper function
void transplant(RedBlackTree* tree, Node* u, Node* v) {
    if (u->parent == NULL) {
        tree->root = v;
    } else if (u == u->parent->left) {
        u->parent->left = v;
    } else {
        u->parent->right = v;
    }
    if (v != NULL) {
        v->parent = u->parent;
    }
}

// Function to delete a node from the Red-Black Tree
void delete(RedBlackTree* tree, int data) {
    Node* z = tree->root;
    while (z != NULL && z->data != data) {
        if (data < z->data)
            z = z->left;
        else
            z = z->right;
    }

    if (z == NULL) {
        printf("Node not found in the tree\n");
        return; // Node to be deleted not found
    }

    Node* y = z; // Node to be unlinked from the tree
    Node* x;     // y's only child or NULL
    int yOriginalColor = y->color;

    if (z->left == NULL) {
        x = z->right;
        transplant(tree, z, z->right);
    } else if (z->right == NULL) {
        x = z->left;
        transplant(tree, z, z->left);
    } else {
        y = findMinValueNode(z->right); // Find the minimum node of the right subtree
        yOriginalColor = y->color;
        x = y->right;

        if (y->parent == z) {
            x->parent = y; // Necessary when x is NULL
        } else {
            transplant(tree, y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }

        transplant(tree, z, y);
        y->left = z->left;
```

```c
        y->left->parent = y;
        y->color = z->color;
    }

    free(z);

    if (yOriginalColor == 0) {
        deleteFixup(tree, x);
    }
}

// Function to perform in-order traversal of the Red-Black Tree
void inOrderTraversal(Node* root) {
    char c[2][6]={"BLACK","RED"};
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d,%s -> ", root->data, c[root->color]);
        inOrderTraversal(root->right);
    }
}

// Function to free memory by deallocating nodes
void freeMemory(Node* root) {
    if (root == NULL)
        return;
    freeMemory(root->left);
    freeMemory(root->right);
    free(root);
}

int main() {
    int choice,value;
    RedBlackTree* tree = createRedBlackTree();
    do
    {
        printf("\n1. Insertion\n2. Deletion\n3. Display\n4. Exit");
        printf("\nEnter your choice: ");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                insert(tree, value);
                break;
        case 2: printf("Enter the value to be deleted: ");
                scanf("%d",&value);
                delete(tree, value);
                break;
        case 3: inOrderTraversal(tree->root);
                break;
        case 4: freeMemory(tree->root);
                break;
        default: printf("\nWrong selection!!! Try again!!!");
        }
```

```
    }while(choice!=4);
    return(0);
}
```

OUTPUT

```
/tmp/tnOjm2NG3L.o
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 1
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 2
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 3
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
1,RED -> 2,BLACK -> 3,RED ->
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 4
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 1
Enter the value to be insert: 5
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
1,BLACK -> 2,BLACK -> 3,RED -> 4,BLACK -> 5,RED ->
1. Insertion
2. Deletion
3. Display
4. Exit
```

```
Enter your choice: 2
Enter the value to be deleted: 3
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
1,BLACK -> 2,BLACK -> 4,BLACK -> 5,RED ->
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 2
Enter the value to be deleted: 5
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 3
1,BLACK -> 2,BLACK -> 4,BLACK ->
1. Insertion
2. Deletion
3. Display
4. Exit
Enter your choice: 4
```

8.ii).Write a program to implement B Trees (its operations)

Solution :

C program to implement the B-Tree

B-Tree

```
/* For simplicity, provide a basic implementation focusing on insertion, search, and
a simple traversal. We will not include deletion as it is quite complex and would
make the code exceedingly long. In practice, B-tree deletion requires handling
numerous cases to redistribute or merge nodes.  */

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

#define MAX_KEYS 3 // Maximum keys in a node (t-1 where t is the minimum degree)
#define MIN_KEYS 1 // Minimum keys in a node (ceil(t/2) - 1)
#define MAX_CHILDREN (MAX_KEYS + 1) // Maximum children in a node (t)

typedef struct BTreeNode {
    int keys[MAX_KEYS];
    struct BTreeNode* children[MAX_CHILDREN];
    int numKeys;
    int isLeaf;
} BTreeNode;

BTreeNode* createNode(int isLeaf) {
    BTreeNode* node = (BTreeNode*)malloc(sizeof(BTreeNode));
    node->isLeaf = isLeaf;
    node->numKeys = 0;
    for (int i = 0; i < MAX_CHILDREN; i++) {
        node->children[i] = NULL;
    }
    return node;
}

void splitChild(BTreeNode* parent, int index, BTreeNode* child) {
    BTreeNode* newChild = createNode(child->isLeaf);
    newChild->numKeys = MIN_KEYS;

    for (int i = 0; i < MIN_KEYS; i++) {
        newChild->keys[i] = child->keys[i + MIN_KEYS + 1];
    }

    if (!child->isLeaf) {
        for (int i = 0; i < MIN_KEYS + 1; i++) {
            newChild->children[i] = child->children[i + MIN_KEYS + 1];
        }
    }
```

```c
    child->numKeys = MIN_KEYS;

    for (int i = parent->numKeys; i >= index + 1; i--) {
        parent->children[i + 1] = parent->children[i];
    }

    parent->children[index + 1] = newChild;

    for (int i = parent->numKeys - 1; i >= index; i--) {
        parent->keys[i + 1] = parent->keys[i];
    }

    parent->keys[index] = child->keys[MIN_KEYS];
    parent->numKeys++;
}

void insertNonFull(BTreeNode* node, int key) {
    int i = node->numKeys - 1;

    if (node->isLeaf) {
        while (i >= 0 && node->keys[i] > key) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }

        node->keys[i + 1] = key;
        node->numKeys++;
    } else {
        while (i >= 0 && node->keys[i] > key) {
            i--;
        }

        if (node->children[i + 1]->numKeys == MAX_KEYS) {
            splitChild(node, i + 1, node->children[i + 1]);

            if (key > node->keys[i + 1]) {
                i++;
            }
        }

        insertNonFull(node->children[i + 1], key);
    }
}

void insert(BTreeNode** root, int key) {
    BTreeNode* r = *root;
    if (r->numKeys == MAX_KEYS) {
        BTreeNode* newRoot = createNode(0);
        newRoot->children[0] = r;
        splitChild(newRoot, 0, r);
        int i = 0;
        if (newRoot->keys[0] < key) {
            i++;
```

```
        }
        insertNonFull(newRoot->children[i], key);
        *root = newRoot;
    } else {
        insertNonFull(r, key);
    }
}

void traverse(BTreeNode* root) {
    if (root == NULL) return;
    int i;
    for (i = 0; i < root->numKeys; i++) {
        if (!root->isLeaf) {
            traverse(root->children[i]);
        }
        printf("%d ", root->keys[i]);
    }

    if (!root->isLeaf) {
        traverse(root->children[i]);
    }
}

int main() {
    BTreeNode* root = createNode(1);

    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 5);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 30);
    insert(&root, 7);
    insert(&root, 17);

    printf("Traversal of the constructed B-tree is:\n");
    traverse(root);

    return 0;
}
```

OUTPUT

```
Traversal of the constructed B-tree is:
5 6 7 10 12 17 20 30
```

8.iii).Write a program to implement B+ Trees (its operations)

Solution :

C program to implement the B+ Tree

B+ Tree

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

struct BPTreeNode {
    int *data;
    struct BPTreeNode **child_ptr;
    bool leaf;
    int n;
}*root = NULL, *np = NULL, *x = NULL;

struct BPTreeNode * init() {
    int i;
    np = (struct BPTreeNode *)malloc(sizeof(struct BPTreeNode));
    np->data = (int *)malloc(sizeof(int) * 5);
    np->child_ptr = (struct BPTreeNode **)malloc(sizeof(struct BPTreeNode *) * 6);
    np->leaf = true;
    np->n = 0;
    for (i = 0; i < 6; i++) {
        np->child_ptr[i] = NULL;
    }
    return np;
}

void traverse(struct BPTreeNode *p) {
    int i;
    for (i = 0; i < p->n; i++) {
        if (p->leaf == false) {
            traverse(p->child_ptr[i]);
        }
        printf(" %d", p->data[i]);
    }
    if (p->leaf == false) {
        traverse(p->child_ptr[i]);
    }
}

void sort(int *p, int n) {
    int i, j, temp;
    for (i = 0; i < n; i++) {
        for (j = i; j <= n; j++) {
            if (p[i] > p[j]) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
```

```
            }
        }
    }
}

int split_child(struct BPTreeNode *x, int i) {
    int j, mid;
    struct BPTreeNode *np1, *np3, *y;
    np3 = init();
    np3->leaf = true;
    if (i == -1) {
        mid = x->data[2];
        x->data[2] = 0;
        x->n--;
        np1 = init();
        np1->leaf = false;
        x->leaf = true;
        for (j = 3; j < 5; j++) {
            np3->data[j - 3] = x->data[j];
            np3->child_ptr[j - 3] = x->child_ptr[j];
            np3->n++;
            x->data[j] = 0;
            x->n--;
        }
        for(j = 0; j < 6; j++) {
            x->child_ptr[j] = NULL;
        }
        np1->data[0] = mid;
        np1->child_ptr[np1->n] = x;
        np1->child_ptr[np1->n + 1] = np3;
        np1->n++;
        root = np1;
    } else {
        y = x->child_ptr[i];
        mid = y->data[2];
        y->data[2] = 0;
        y->n--;
        for (j = 3; j < 5; j++) {
            np3->data[j - 3] = y->data[j];
            np3->n++;
            y->data[j] = 0;
            y->n--;
        }
        x->child_ptr[i + 1] = y;
        x->child_ptr[i + 1] = np3;
    }
    return mid;
}

void insert(int a) {
    int i, temp;
    x = root;
    if (x == NULL) {
        root = init();
```

```
        x = root;
    } else {
        if (x->leaf == true && x->n == 5) {
            temp = split_child(x, -1);
            x = root;
            for (i = 0; i < (x->n); i++) {
                if ((a > x->data[i]) && (a < x->data[i + 1])) {
                    i++;
                    break;
                } else if (a < x->data[0]) {
                    break;
                } else {
                    continue;
                }
            }
            x = x->child_ptr[i];
        } else {
            while (x->leaf == false) {
                for (i = 0; i < (x->n); i++) {
                    if ((a > x->data[i]) && (a < x->data[i + 1])) {
                        i++;
                        break;
                    } else if (a < x->data[0]) {
                        break;
                    } else {
                        continue;
                    }
                }
                if ((x->child_ptr[i])->n == 5) {
                    temp = split_child(x, i);
                    x->data[x->n] = temp;
                    x->n++;
                    continue;
                } else {
                    x = x->child_ptr[i];
                }
            }
        }
    }
    x->data[x->n] = a;
    sort(x->data, x->n);
    x->n++;
}

int main() {
    int i, n, t;
    printf("enter the no of elements to be inserted\n");
    scanf("%d", &n);
    for(i = 0; i < n; i++) {
        printf("enter the element\n");
        scanf("%d", &t);
        insert(t);
    }
    printf("traversal of constructed tree\n");
```

```
    traverse(root);
    return 0;
}
```

OUTPUT

```
enter the no of elements to be inserted
8
enter the element
10
enter the element
20
enter the element
5
enter the element
6
enter the element
12
enter the element
30
enter the element
7
enter the element
17
traversal of constructed tree
5 6 7 10 12 17 20 30
```

9.i).Write a program to implement the graph traversal methods (Breadth First Search)

Solution :

C program to implement the Breadth First Search a graph traversal methods

BFS

```
#include<stdio.h>

// creating queue data structure using arrays
int queue[10];

// defining pointers of the queue to perform pop and push
int front=0,back=0;

// defining push operation on the queue
void push(int var)
{
    queue[back] = var;
    back++;
}

// defining pop operation on queue
void pop()
{
    queue[front] = 0;
    front++;
}

// creating a visited array to keep the track of visited nodes
int visited[7] = {0};

int main()
{
    int v,n,i,j;
    // adjacenty matrix representing graph
    int graph[10][10];
    printf("Enter the number of vertices: ");
    scanf("%d", &n);
    printf("Enter graph data in matrix form:    \n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);

    // adding a starting node in the list
    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    push(v);
```

```
    while(front != back)
    {
        int current = queue[front];

        // printing current element
        printf("%d ", current);

        // popping the front element from the queue
        pop();

        for(int i=0;i < 6;i++)
        {
            // adding non-visited connected nodes of the current node to the queue
            if((graph[current-1][i] == 1) && (visited[i] == 0))
            {
                visited[i] = 1; // marking visisted
                push(i+1);
            }
        }
    }
    return 0;
}
```

OUTPUT

```
Enter the number of vertices: 6
Enter graph data in matrix form:
0 1 1 0 0 0
1 0 1 0 0 0
1 1 0 1 1 0
0 0 1 0 0 0
0 0 1 0 0 1
0 0 0 0 1 0
Enter the starting vertex: 2
2 1 3 2 4 5 6
```

9.ii).Write a program to implement the graph traversal methods (Depth First Search)

Solution :

C program to implement the Depth First search a graph traversal methods

DFS

```
#include <stdio.h>
int a[20][20], visited[20], n;
void dfs(int v)
{
        int i;
    visited[v] = 1;
    for (i = 1; i <= n; i++)
        {
                if (a[v][i] && !visited[i])
                {
                        printf("\n %d->%d", v, i);
            dfs(i);
        }
    }
}

int main( )
{
        int i, j,v, count = 0;
    printf("\n Enter number of vertices:");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        {
                visited[i] = 0;
        for (j = 1; j <= n; j++)
                        a[i][j] = 0;
        }
    printf("\n Enter the adjacency matrix:\n");
    for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++)
                        scanf("%d", &a[i][j]);
        printf("Enter the starting vertex: ");
    scanf("%d", &v);
        dfs(v);
    return 0;
}
```

OUTPUT

```
Enter number of vertices:6
Enter the adjacency matrix:
0 1 1 0 0 0
1 0 1 0 0 0
1 1 0 1 1 0
0 0 1 0 0 0
0 0 1 0 0 1
0 0 0 0 1 0
Enter the starting vertex: 2
2->1
1->3
3->4
3->5
5->6
```

10.i).Write a program to Implement a Pattern matching algorithms using Boyer-Moore

Solution :

C program to Implement a Pattern matching algorithms using Boyer- Moore

Boyer-Moore Pattern matching

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

int max(int a, int b)
{
    if(a > b)
                return a;
    else
                return b;
}
int boyermorre(char p[],char t[])
{
    int bctable[128],i,j,k;
    int n = strlen(t);
    int m = strlen(p);
    for(j=0; j<128; j++)
    {
        bctable[j]=m;
    }
    for(j=0; j<m; j++)
    {
        k=(int)p[j];
        bctable[k]=m-j-1;
    }
    i=m-1;
    while(i < n)
    {
        j=m-1;
        while(j >= 0 && p[j] == t[i])
        {
            i--;
            j--;
        }
        if(j == -1)
            return i+1;
        i = i + max((int)bctable[t[i]],m-j);
    }
    return 0;
}
```

```c
int main() {
   char t[]="kiss*miss*in*mississippi";
   char p[]="missi";
   int i;
   i=boyermorre(p,t);
   if(i)
                 printf("pattern is present in text at position %d",i+1);
   else
                 printf("pattern is not present in text");
   return 0;
}
```

OUTPUT

```
pattern is present in text at position 14
```

10.ii).Write a program to Implement a Pattern matching algorithms using Knuth-Morris-Pratt

Solution :

C program to Implement a Pattern matching algorithms using Knuth-Morris-Pratt

Knuth-Morris-Pratt Pattern matching

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

int lps[100];
void longestPrefixSuffix(char p[])
{
        int i=1,j=0;
        int m = strlen(p);
        lps[0] = 0;
        while(i < m)
        {
                if( p[j] == p[i])
                {
                        lps[i]=j+1;
                        i++;
                        j++;
                }
                else if(j>0)
                        j = lps[j-1];
                else
                {
                        lps[i]=0;
                        i++;
                }
        }
}

int kmp (char p[],char t[])
{
        int n,m;
        int i=0,j=0;
        n = strlen(t);
    m = strlen(p);
        longestPrefixSuffix(p);
        while( i < n )
        {
                if ( p[j] == t[i])
                {
                        if (j == m-1 )
                                return i-j;
```

```
                        i++;
                        j++;
                }
                else if(j>0)
                        j = lps[j-1];
                else
                        i++;
        }
        return 0;
}

int main() {
   char t[]="kiss*miss*in*mississippi";
   char p[]="missi";
   int i;
   i=kmp(p,t);
   if(i)
                printf("pattern is present in text at position %d",i+1);
   else
                printf("pattern is not present in text");
   return 0;
}
```

OUTPUT

```
pattern is present in text at position 14
```