1. Write a C program to reverse a string using stack.

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

struct Stack* createStack(unsigned capacity)
{
   struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
   stack->capacity = capacity;
   stack->top = -1;
   stack->array = (char*) malloc(stack->capacity * sizeof(char));
   return stack;
}

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

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

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

char pop(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top--];
}
void reverse(char str[])
{

   int n = strlen(str);
   struct Stack* stack = createStack(n);
```

```
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);


    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}
  int main()
{
    char str[] = "Nandini";

    reverse(str);
    printf("Reversed string is %s", str);

    return 0;
}
```

2. Write a C program to Infix to Postfix conversion using stack.

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

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

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

    if (!stack)
        return NULL;

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

    return stack;
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1 ;
}
char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}
char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}
void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}

int isOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

int Prec(char ch)
{
    switch (ch)
    {
    case '+':
    case '-':
        return 1;

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

    case '^':
        return 3;
    }
    return -1;
}
int infixToPostfix(char* exp)
```

```
{
    int i, k;

    struct Stack* stack = createStack(strlen(exp));
    if(!stack) // See if stack was created successfully
        return -1 ;

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

        if (isOperand(exp[i]))
            exp[++k] = exp[i];

        else if (exp[i] == '(')
            push(stack, exp[i]);

        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1; // invalid expression
            else
                pop(stack);
        }
        else
        {
            while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }

    }

    while (!isEmpty(stack))
        exp[++k] = pop(stack );

    exp[++k] = '\0';
    printf( "%s", exp );
}

int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

3. Write a C program to implement queue using two stacks.

```
#include <stdio.h>
#include <stdlib.h>
struct node
     {
        int data;
        struct node *next;
};
     void push(struct node** top, int data);
int pop(struct node** top);
struct queue
{
   struct node *stack1;
   struct node *stack2;
};
enqueue(struct queue *q, int x)
{
   push(&q->stack1, x);
}
void dequeue(struct queue *q)
{
   int x;
   if (q->stack1 == NULL && q->stack2 == NULL) {
      printf("queue is empty");
      return;
   }
   if (q->stack2 == NULL) {
    while (q->stack1 != NULL) {
      x = pop(&q->stack1);
      push(&q->stack2, x);
      }
   }
   x = pop(&q->stack2);
   printf("%d\n", x);
}
void push(struct node** top, int data)
{
   struct node* newnode = (struct node*) malloc(sizeof(struct node));
      if (newnode == NULL) {
         printf("Stack overflow \n");
         return;
      }
    newnode->data = data;
    newnode->next = (*top);
    (*top) = newnode;
}
int pop(struct node** top)
{
    int buff;
    struct node *t;
    if (*top == NULL) {
        printf("Stack underflow \n");
        return;
    }
    else {
        t = *top;
        buff = t->data;
             *top = t->next;
              free(t);
        return buff;
    }
}
void display(struct node *top1,struct node *top2)
{
    while (top1 != NULL) {
        printf("%d\n", top1->data);
        top1 = top1->next;
    }
    while (top2 != NULL) {
        printf("%d\n", top2->data);
        top2 = top2->next;
    }
}
int main()
{
    struct queue *q = (struct queue*)malloc(sizeof(struct queue));
    int f = 0, a;
    char ch = 'y';
    q->stack1 = NULL;
    q->stack2 = NULL;
    while (ch == 'y'||ch == 'Y') {
        printf("enter your choice\n1.add to queue\n2.remove
               from queue\n3.display\n4.exit\n");
        scanf("%d", &f);
        switch(f) {
            case 1 : printf("enter the element to be added to queue\n");
                     scanf("%d", &a);
                     enqueue(q, a);
```

```
                    break;
                case 2 : dequeue(q);
                    break;
                case 3 : display(q->stack1, q->stack2);
                    break;
                case 4 : exit(1);
                    break;
                default : printf("invalid\n");
                     break;
            }
        }
    }
```

4. Write a C program for insertion and deletion of BST.

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

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


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

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

struct node* insert(struct node* node, int key)
```

```
{
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
}

int main()
{
    inorder(root);

    return 0;
}
```

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

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

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

void inorder(struct node *root)
```

```c
{
   if (root != NULL)
   {
      inorder(root->left);
      printf("%d ", root->key);
      inorder(root->right);
   }
}

struct node* insert(struct node* node, int key)
{
   if (node == NULL) return newNode(key);

   if (key < node->key)
      node->left  = insert(node->left, key);
   else
      node->right = insert(node->right, key);

   return node;
}

struct node * minValueNode(struct node* node)
{
   struct node* current = node;

   while (current && current->left != NULL)
      current = current->left;

   return current;
}

struct node* deleteNode(struct node* root, int key)
{
   if (root == NULL) return root;

   if (key < root->key)
      root->left = deleteNode(root->left, key);

   else if (key > root->key)
      root->right = deleteNode(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 = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
int main()
{

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    return 0;
}
```