B.CHANDRA MAHESH
AP19110010149
CSE-G

ASSIGNMENT 5

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

```
// 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; }

// Stack is empty when top is equal to -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--];
}

// A stack based function to reverse a string
void reverse(char str[])
{
    // Create a stack of capacity
    //equal to length of string
    int n = strlen(str);
    struct Stack* stack = createStack(n);

    // Push all characters of string to stack
    int i;
    for (i = 0; i < n; i++)
      push(stack, str[i]);

    // Pop all characters of string and
    // put them back to str
    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

int main()
{
```

```
    char str[] = "Chandra Mahesh";

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

    return 0;
}
```

Output:
Reversed string is hsehaM ardnahC

2. Write a C program for Infix To Postfix Conversion Using Stack.

```
// C program to convert infix expression to postfix
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

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


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

    if (!stack)
        return NULL;

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

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

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


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

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

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

    case '^':
        return 3;
    }
    return -1;
}



// The main function that converts given infix expression
// to postfix expression.
int infixToPostfix(char* exp)
{
    int i, k;

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

    for (i = 0, k = -1; exp[i]; ++i)
    {
        // If the scanned character is an operand, add it to output.
        if (isOperand(exp[i]))
            exp[++k] = exp[i];

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

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

  }

// pop all the operators from the stack
   while (!isEmpty(stack))
      exp[++k] = pop(stack );

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

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

output:

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

3. Write a C Program to Implement Queue Using Two Stacks

```
/* C Program to implement a queue using two stacks */
#include <stdio.h>
#include <stdlib.h>

/* structure of a stack node */
struct sNode {
    int data;
    struct sNode* next;
};

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

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

/* structure of queue having two stacks */
struct queue {
    struct sNode* stack1;
    struct sNode* stack2;
};

/* Function to enqueue an item to queue */
void enQueue(struct queue* q, int x)
{
    push(&q->stack1, x);
}

/* Function to deQueue an item from queue */
int deQueue(struct queue* q)
{
    int x;

    /* If both stacks are empty then error */
    if (q->stack1 == NULL && q->stack2 == NULL) {
        printf("Q is empty");
```

```
        getchar();
        exit(0);
    }

    /* Move elements from stack1 to stack 2 only if
       stack2 is empty */
    if (q->stack2 == NULL) {
        while (q->stack1 != NULL) {
            x = pop(&q->stack1);
            push(&q->stack2, x);
        }
    }

    x = pop(&q->stack2);
    return x;
}


/* Function to push an item to stack*/
void push(struct sNode** top_ref, int new_data)
{
    /* allocate node */
    struct sNode* new_node = (struct sNode*)malloc(sizeof(struct sNode));
    if (new_node == NULL) {
        printf("Stack overflow \n");
        getchar();
        exit(0);
    }

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

    /* link the old list off the new node */
    new_node->next = (*top_ref);

    /* move the head to point to the new node */
    (*top_ref) = new_node;
}
```

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

    /*If stack is empty then error */
    if (*top_ref == NULL) {
        printf("Stack underflow \n");
        getchar();
        exit(0);
    }
    else {
        top = *top_ref;
        res = top->data;
        *top_ref = top->next;
        free(top);
        return res;
    }
}


/* Driver function to test anove functions */
int main()
{
    /* Create a queue with items 1 2 3*/
    struct queue* q = (struct queue*)malloc(sizeof(struct queue));
    q->stack1 = NULL;
    q->stack2 = NULL;
    enQueue(q, 1);
    enQueue(q, 2);
    enQueue(q, 3);

    /* Dequeue items */
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));
    printf("%d ", deQueue(q));
```

```
    return 0;
}
```

Output:
1 2 3

4. write a c program for insertion and deletion of BST.

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

struct node
{
    int info;
    struct node *lchild;
    struct node *rchild;
}*root;



void find(int item,struct node **par,struct node **loc)
{
    struct node *ptr,*ptrsave;

    if(root==NULL)  /*tree empty*/
    {
        *loc=NULL;
        *par=NULL;
        return;
    }
    if(item==root->info) /*item is at root*/
    {
        *loc=root;
        *par=NULL;
        return;
    }
    /*Initialize ptr and ptrsave*/
```

```
    if(item<root->info)
        ptr=root->lchild;
    else
        ptr=root->rchild;
    ptrsave=root;

    while(ptr!=NULL)
    {
        if(item==ptr->info)
        {       *loc=ptr;
            *par=ptrsave;
            return;
        }
        ptrsave=ptr;
        if(item<ptr->info)
            ptr=ptr->lchild;
        else
            ptr=ptr->rchild;
     }/*End of while */
     *loc=NULL;   /*item not found*/
     *par=ptrsave;
}/*End of find()*/

void insert(int item)
{       struct node *tmp,*parent,*location;
    find(item,&parent,&location);
    if(location!=NULL)
    {
        printf("Item already present");
        return;
    }

    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->info=item;
    tmp->lchild=NULL;
    tmp->rchild=NULL;

    if(parent==NULL)
```

```
        root=tmp;
    else
        if(item<parent->info)
            parent->lchild=tmp;
        else
            parent->rchild=tmp;
}/*End of insert()*/


void case_a(struct node *par,struct node *loc )
{
    if(par==NULL) /*item to be deleted is root node*/
        root=NULL;
    else
        if(loc==par->lchild)
            par->lchild=NULL;
        else
            par->rchild=NULL;
}/*End of case_a()*/

void case_b(struct node *par,struct node *loc)
{
    struct node *child;

    /*Initialize child*/
    if(loc->lchild!=NULL) /*item to be deleted has lchild */
        child=loc->lchild;
    else                /*item to be deleted has rchild */
        child=loc->rchild;

    if(par==NULL )   /*Item to be deleted is root node*/
        root=child;
    else
        if( loc==par->lchild)   /*item is lchild of its parent*/
            par->lchild=child;
        else                /*item is rchild of its parent*/
            par->rchild=child;
}/*End of case_b()*/
```

```
void case_c(struct node *par,struct node *loc)
{
   struct node *ptr,*ptrsave,*suc,*parsuc;

   /*Find inorder successor and its parent*/
   ptrsave=loc;
   ptr=loc->rchild;
   while(ptr->lchild!=NULL)
   {
      ptrsave=ptr;
      ptr=ptr->lchild;
   }
   suc=ptr;
   parsuc=ptrsave;

   if(suc->lchild==NULL && suc->rchild==NULL)
      case_a(parsuc,suc);
   else
      case_b(parsuc,suc);

   if(par==NULL) /*if item to be deleted is root node */
      root=suc;
   else
      if(loc==par->lchild)
         par->lchild=suc;
      else
         par->rchild=suc;

   suc->lchild=loc->lchild;
   suc->rchild=loc->rchild;
}/*End of case_c()*/
int del(int item)
{
   struct node *parent,*location;
   if(root==NULL)
   {
      printf("Tree empty");
```

```
        return 0;
    }

    find(item,&parent,&location);
    if(location==NULL)
    {
        printf("Item not present in tree");
        return 0;
    }

    if(location->lchild==NULL && location->rchild==NULL)
        case_a(parent,location);
    if(location->lchild!=NULL && location->rchild==NULL)
        case_b(parent,location);
    if(location->lchild==NULL && location->rchild!=NULL)
        case_b(parent,location);
    if(location->lchild!=NULL && location->rchild!=NULL)
        case_c(parent,location);
    free(location);
}/*End of del()*/

int preorder(struct node *ptr)
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return 0;
    }
    if(ptr!=NULL)
    {
        printf("%d  ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
    }
}/*End of preorder()*/

void inorder(struct node *ptr)
{
```

```
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        inorder(ptr->lchild);
        printf("%d  ",ptr->info);
        inorder(ptr->rchild);
    }
}/*End of inorder()*/

void postorder(struct node *ptr)
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        postorder(ptr->lchild);
        postorder(ptr->rchild);
        printf("%d  ",ptr->info);
    }
}/*End of postorder()*/

void display(struct node *ptr,int level)
{
    int i;
    if ( ptr!=NULL )
    {
        display(ptr->rchild, level+1);
        printf("\n");
        for (i = 0; i < level; i++)
            printf("    ");
        printf("%d", ptr->info);
```

```
        display(ptr->lchild, level+1);
    }/*End of if*/
}/*End of display()*/
main()
{
    int choice,num;
    root=NULL;
    while(1)
    {
        printf("\n");
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Inorder Traversal\n");
        printf("4.Preorder Traversal\n");
        printf("5.Postorder Traversal\n");
        printf("6.Display\n");
        printf("7.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
         case 1:
            printf("Enter the number to be inserted : ");
            scanf("%d",&num);
            insert(num);
            break;
         case 2:
            printf("Enter the number to be deleted : ");
            scanf("%d",&num);
            del(num);
            break;
         case 3:
            inorder(root);
            break;
         case 4:
            preorder(root);
            break;
```

```
        case 5:
           postorder(root);
           break;
        case 6:
           display(root,1);
           break;
        case 7:
           break;
        default:
           printf("Wrong choice\n");
       }/*End of switch */
    }/*End of while */
}/*End of main()*/
```