**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[] = "P.Ruthikvenkat";

reverse(str);

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

return 0;

}

Output:

Reversed string is taknevkihtuR.P

**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**

#include<stdio.h>

int s1[10],s2[10],top1=-1,top2=-1;

void s1push(int e)

{

s1[++top1]=e;

}

int s1empty()

{

if(top1==-1)

return 1;

else

return 0;

}

int s2empty()

{

if(top2==-1)

return 1;

else

return 0;

}

void s2push(int e)

{

s2[++top2]=e;

}

int s1top()

{

return s1[top1];

}

int s2top()

{

return s2[top2];

}

void s1pop()

{

top1--;

}

void s2pop()

{

top2--;

}

void pop()

{

if(top1==-1)

{

printf("queue is empty");

}

else

{

while(s1empty()!=1)

{

s2push(s1top());

s1pop();

}

s2pop();

while(s2empty()!=1)

{

s1push(s2top());

s2pop();

}

}

}

void display()

{

int i=top1;

if(top1==-1)

{

printf("queue is empty");

}

else

{

for(i=0;i<=top1;i++)

printf("%d ",s1[i]);

printf("\n");

}

}

main()

{

int i,n,e;

printf("1)push\n2)pop\n3)display\n4)exit\n");

do{

printf("enter the choice\n");

scanf("%d",&n);

switch(n)

{

case 1:

printf("enter the element to insert\n");

scanf("%d",&e);

s1push(e);

break;

case 2:

pop();

break;

case 3:

display();

brek;

}

}while(n!=4);

}

**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;  
}  
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;  
}  
struct node \*find\_min(struct node \*root)  
{  
    if(root==NULL)  
    {  
        return 0;  
    }  
    else if(root->left==NULL)  
    {  
        return root;  
    }  
    else  
    {  
        return(find\_min(root->left));  
    }  
}  
struct node\* removee(struct node \*root,int item)  
{  
    struct node \*temp;  
    if(root==NULL)  
    {  
        printf("the tree is empty");  
    }  
    else if(item<root->key)  
    {  
        root->left=removee(root->left,item);  
    }  
    else if(item>root->key)  
    {  
        root->right=removee(root->right,item);  
    }  
    else  
    {  
        struct node \*temp;  
        if(root->left==NULL && root->right==NULL)  
    {  
    free(root);  
    return NULL;  
    }  
        else if(root->left!=NULL && root->right!=NULL)  
        {  
            temp=find\_min(root->right);  
            root->key=temp->key;  
            root->right=removee(root->right,root->key);  
        }  
        else  
        {  
            temp=root;  
            if(root->left==NULL)  
            {  
                root=root->right;  
            }  
            else if(root->right==NULL)  
            {  
                root=root->left;  
            }  
            free(temp);  
        }  
        return root;  
    }  
}  
void inorder(struct node \*root)  
{  
    if (root != NULL)  
    {  
    inorder(root->left);  
    printf("%d \n", root->key);  
    inorder(root->right);  
    }  
}  
int main()  
{  
    int item;  
    struct node \*root = NULL;  
    root = insert(root, 3);  
    insert(root, 54);  
    insert(root, 87);  
    insert(root, 35);  
    insert(root, 25);  
    insert(root, 99);  
    insert(root, 4);  
    inorder(root);  
    root = removee(root, 99);  
    inorder(root);  
}