Assignment

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

struct Stack
{
int top;
capacity;
char* array;
};
# creating a stack
#initial size of stack is 0
struct Stack* createStack( 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;
}
# Stack is full when top is equal to the last index
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; }
#Function to add an item to stack.

# It increases top by 1
void push(struct Stack* stack, char item)
```

```
{
if (isFull(stack))
return;
stack->array[++stack->top] = item;
}
#Function to remove an item from stack.
# It decreases top by 1
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[] = "Jaynica";
reverse(str);
printf("Reversed string is %s", str);
return 0;
}
```

Out put:

```
The reversed string is “acinyaJ”
```

2)Write a c programme to convert infix to postfix conversion using stack.

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
stack[++top] = x;
}
char pop()
{
if(top == -1)
return -1;
else
return stack[top--];
}
int priority(char x)
{
if(x == '(')
return 0;
if(x == '+' || x == '-')
```

```
return 1;
if(x == '*' || x == '/')
return 2;
}
main()
{
char exp[20];
char *e, x;
printf("Enter the expression :: ");
scanf("%s",exp);

e = exp;
while(*e != '\0')
{
if(isalnum(*e))
printf("%c",*e);
else if(*e == '(')
push(*e);
else if(*e == '(')
{
while((x = pop()) != '(')
printf("%c", x);
}
else
{
while(priority(stack[top]) >= priority(*e))
printf("%c",pop());
push(*e);
}
e++;
}
while(top != -1)
{
```

```
printf("%c",pop());
}
}
```

OUTPUT:

```
Enter = z+b*c
zbc*+
```

3)Write a c programme 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;
};
void 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 ur 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")
```

4)Write a program for insertion and deletion in BST.

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

struct treeNode {
int data;
struct treeNode *left, *right;
};
struct treeNode *root = NULL;
/* create a new node with the given data */
struct treeNode* createNode(int data) {
```

```
struct treeNode *newNode;
newNode = (struct treeNode *) malloc(sizeof (struct treeNode));
newNode->data = data;
newNode->left = NULL;
newNode->right = NULL;
return(newNode);
}
/* insertion in binary search tree */
void insertion(struct treeNode **node, int data) {
if (*node == NULL) {
*node = createNode(data);
} else if (data < (*node)->data) {
insertion(&(*node)->left, data);
} else if (data > (*node)->data) {
insertion(&(*node)->right, data);
}
}

/* deletion in binary search tree */

void deletion(struct treeNode **node, struct treeNode **parent, int data) {
struct treeNode *tmpNode, *tmpParent;
if (*node == NULL)
return;
if ((*node)->data == data) {
/* deleting the leaf node */
if (!(*node)->left && !(*node)->right) {
if (parent) {
/* delete leaf node */
if ((*parent)->left == *node)
(*parent)->left = NULL;
else
(*parent)->right = NULL;
free(*node);
} else {
/* delete root node with no children */
free(*node);
}
/* deleting node with one child */
} else if (!(*node)->right && (*node)->left) {
/* deleting node with left child alone */
tmpNode = *node;
(*parent)->right = (*node)->left;
free(tmpNode);
*node = (*parent)->right;
} else if ((*node)->right && !(*node)->left) {
/* deleting node with right child alone */
tmpNode = *node;
(*parent)->left = (*node)->right;
free(tmpNode);
(*node) = (*parent)->left;
} else if (!(*node)->right->left) {
/*
* deleting a node whose right child
* is the smallest node in the right

* subtree for the node to be deleted.
*/

tmpNode = *node;
(*node)->right->left = (*node)->left;
(*parent)->left = (*node)->right;
free(tmpNode);
*node = (*parent)->left;
} else {
/*
```

```
* Deleting a node with two children.
* First, find the smallest node in
* the right subtree. Replace the
* smallest node with the node to be
* deleted. Then, do proper connections
* for the children of replaced node.
*/
tmpNode = (*node)->right;
while (tmpNode->left) {
tmpParent = tmpNode;
tmpNode = tmpNode->left;
}
tmpParent->left = tmpNode->right;
tmpNode->left = (*node)->left;
tmpNode->right =(*node)->right;
free(*node);
*node = tmpNode;
}
} else if (data < (*node)->data) {
/* traverse towards left subtree */
deletion(&(*node)->left, node, data);
} else if (data > (*node)->data) {
/* traversing towards right subtree*/

deletion(&(*node)->right, node, data);
}
}

break;
```