**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[] = "GreeshmaChowdary";

reverse(str);

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

return 0;

}

Output:

Reversed string is yradwohCamhseerG

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()\*/

Output:

#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, 12);

insert(root, 51);

insert(root, 43);

insert(root, 37);

insert(root, 98);

insert(root, 5);

inorder(root);

root = removee(root, 12);

inorder(root);

}