Experiment No. : 3

Batch: SY-IT (B3)   **Roll No.: 16010423076**   **Experiment No.:3**

Aim:

   a) WAP to create a stack using SLL by implementing following operations
         1. Create stack, 2. Insert an element, 3. Delete an element, 4. Display top element.

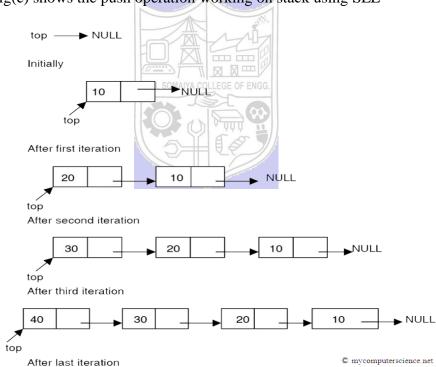   **b)** WAP to convert a given infix expression into equivalent postfix form using stack implemented in part (a).

Resources Used: Turbo C editor and compiler

Theory:

   a) Stack :- Stack can be implemented by using an array or by using a linked list. One important feature of stack, that the last element inserted into a stack is the first element deleted. Therefore stack is also called as Last in First out (LIFO) list.

   Linked List implementation –

   Fig(c) shows the push operation working on stack using SLL

Fig(c) Push operation using SLL

---

b) Expression conversion

        Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines.

Examples

Infix expression   *(2 * 5 - 1 * 2) / (11 – 9)*

Postfix Expression   *2 5 * 1 2 * - 11 9 - /*

Algorithm :

Infix String : a+b*c-d

1. Read an item from input infix expression
2. If item is an operand append it to postfix string
3. If item is "(" push it on the stack
4. If the item is an operator and top of the stack(tos) is also operator
   1. If the operator has higher precedence than the one on tos then push it onto the operator stack
   2. If the operator has lower or equal precedence than the one on tos then
      1. pop the operator on tos and append it to postfix string(repeat if tos is again an operator)
      2. push lower precedence operator onto the stack
5. If item is ")" pop all operators from tos one-by-one and append it to postfix string, until a "(" is encountered on stack. Remove "(" from tos and discard it.
6. If end of infix string, pop the items from tos one-by-one and append to postfix string. If other than operator anything is encountered on the stack at this step, then declare input as invalid input.

                    Infix String   : a+b*c-d
                    Postfix String : abc*+d-

Activity :

    a) Implement "dynamic stack" with following functions
1. **void createStack(void) :** Create and initializes the top to NULL, creating the empty stack.
2. **void push(char x) :** Creates a node with value 'x' and inserts on top of the stack.
3. **char pop(void) :** Deletes a node from top of the stack and returns the deleted value.
4. **boolean isEmpty(void) :** Returns "1" for stack empty; "0" otherwise.
5. **char peek(void) :** Return current stack top element.

A Constituent College of Somaiya Vidyavihar University

NOTE : Map appropriate methods of SLL with the methods of STACK and use.

b) Implement expression conversion

1. Implement the above algorithm given for INFIX to POSTFIX expression conversion.

Results: A program depicting the correct behaviour of stack in conversion of expression and capable of handling all possible exceptional conditions and the same is reflecting clearly in the output.

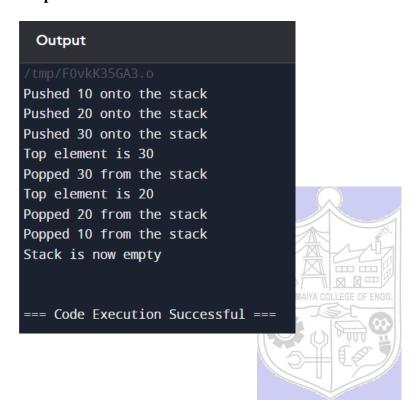**Program and Output:**

1) Code :

```c
#include<stdio.h>

#include<stdlib.h>


//declaring Node and stack structure

struct Node{

int data;

struct Node* next;

};


struct Stack{

struct Node* top;

};


//creating the stack

struct Stack* createStack(){

struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;
```

```
}


//push

void push(struct Stack* stack, int data){

struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

if(newNode == NULL){

printf("Memory allocation failed\n");

return;

}

newNode->data = data;

newNode->next = stack->top;

stack->top = newNode;

printf("Pushed %d onto the stack \n", data);

}


//element delete

int pop(struct Stack* stack){

if(stack->top == NULL){

printf("Stack is now empty\n");

return -1;

}

struct Node* temp = stack->top;

int poppedData = temp->data;

stack->top = stack->top->next;

free(temp);

printf("Popped %d from the stack\n", poppedData);

return poppedData;
```

```c
}

//Top element
void displayTop(struct Stack* stack){
if(stack -> top == NULL){
printf("Stack is empty\n");
}
else{
printf("Top element is %d \n", stack->top->data);
}
}

//main function
int main() {
 struct Stack* stack = createStack();

 push(stack, 10);
 push(stack, 20);
 push(stack, 30);

 displayTop(stack);

 pop(stack);
 displayTop(stack);

 pop(stack);
 pop(stack);
```

```
 pop(stack);


 return 0;

}
```

Output :

```
Output
/tmp/F0vkK35GA3.o
Pushed 10 onto the stack
Pushed 20 onto the stack
Pushed 30 onto the stack
Top element is 30
Popped 30 from the stack
Top element is 20
Popped 20 from the stack
Popped 10 from the stack
Stack is now empty


=== Code Execution Successful ===
```

2) Code :

```
#include<stdio.h>

#include<stdlib.h>

#include<ctype.h>


//declaring Node and stack structure

struct Node{

char data;

struct Node* next;
```
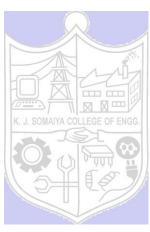
```
};

struct Stack{

struct Node* top;

};


//creating the stack

struct Stack* createStack(){

struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));

stack->top = NULL;

return stack;

}


//push

void push(struct Stack* stack, char data){

struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

if(newNode == NULL){

printf("Memory allocation failed\n");

return;

}

newNode->data = data;

newNode->next = stack->top;

stack->top = newNode;

printf("Pushed %c onto the stack \n", data);

}


//element delete
```

```c
char pop(struct Stack* stack){

if(stack->top == NULL){

printf("Stack is now empty\n");

return -1;

}

struct Node* temp = stack->top;

char poppedData = temp->data;

stack->top = stack->top->next;

free(temp);

printf("Popped %c from the stack\n", poppedData);

return poppedData;

}


//Top element

char peek(struct Stack* stack){

if(stack -> top == NULL){

printf("Stack is empty\n");

return -1;

}

else{

return stack->top->data;

}

}


//check if operator

int isOperator(char ch) {

return ch == '+' || ch == '-' || ch == '*' || ch == '/';
```

```
}
```

```c
//check precedence

int precedence(char ch) {

switch(ch) {

case '+':

case '-':

return 1;

case '*':

case '/':

return 2;

}

return 0;

}
```

```c
//convert infix to postfix

void infixToPostfix(char* exp) {

struct Stack* stack = createStack();

int i, k;

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

if(isalnum(exp[i]))

exp[++k] = exp[i];

else if(exp[i] == '(')

push(stack, exp[i]);

else if(exp[i] == ')') {

while(stack->top && peek(stack) != '(')

exp[++k] = pop(stack);
```

```c
pop(stack);

}

else if(isOperator(exp[i])) {

while(stack->top && precedence(peek(stack)) >= precedence(exp[i]))

exp[++k] = pop(stack);

push(stack, exp[i]);

}

}

while(stack->top)

exp[++k] = pop(stack);

exp[++k] = '\0';

printf("Postfix expression: %s\n", exp);

}


//main function

int main() {

char exp[] = "a+b*(c^d-e)^(f+g*h)-i";

printf("Infix expression: %s\n", exp);

infixToPostfix(exp);

return 0;

}
```

Output :

```
Output

/tmp/jcmZ4w4Lah.o
Infix expression: ((A+B)*C/D)+E
Pushed ( onto the stack
Pushed ( onto the stack
Pushed + onto the stack
Popped + from the stack
Popped ( from the stack
Pushed * onto the stack
Popped * from the stack
Pushed / onto the stack
Popped / from the stack
Popped ( from the stack
Pushed + onto the stack
Popped + from the stack
Postfix expression: AB+C*D/E+


=== Code Execution Successful ===
```

Outcomes:

CO2. Apply linear and non-linear data structure in application development.

Conclusion:

I learned in this experiment how to effectively implement and use a stack data structure in the solution to a very common problem in computer science, that of converting an infix expression into its equivalent postfix form. Going through the code helped me realize exactly how linear data structures like stacks could be used with real applications in scenarios like expression evaluation and parsing tasks. This exercise thus underlines the necessity of understanding operator precedence and associativity and, more practically, data structures in solving problems. Overall, this assignment enabled me to apply fundamental data structure concepts in developing applications.

.

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of faculty in-charge with date

References:

Books/ Journals/ Websites:

- Y. Langsam, M. Augenstin and A. Tannenbaum, "Data Structures using C", Pearson Education Asia, 1st Edition, 2002

A Constituent College of Somaiya Vidyavihar University