**NAME: SUVIDHI V. PAREEK**

**DIV: 3/ ROLL NO. 10**

**BATCH: A**

**EXPERIMENT NO. 2**

**AIM: To convert infix expression to postfix expression using stack ADT.**

**OBJECTIVE:**

**1. Understand the use of stack.**

**2. Understand how to import an ADT in an application program.**

**3. Understand the instantiation of stack ADT in an application program.**

**4. Understand how the member functions of an ADT are accessed in an application program.**

**THEORY:**

**An arithmetic expression consists of operands and operators.**

**INFIX EXPRESSION- An expression is said to be in infix notation if the operators in the expression are placed in between the operands on which the operator works.**

**POSTFIX EXPRESSION- An expression is said to be in postfix notation if the operators in the expression are placed after the operands on which the operator works.**

**To convert infix expression to postfix expression, use the stack data structure. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.**

**ALGORITHM:**

* **Scan all the symbols one by one from left to right in the given Infix Expression.**
* **If the reading symbol is an operand, then immediately append it to the Postfix Expression.**
* **If the reading symbol is left parenthesis ‘( ‘, then Push it onto the Stack.**
* **If the reading symbol is right parenthesis ‘)’, then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.**
* **If the reading symbol is an operator (+, –, \*, /), then Push it onto the Stack. However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix. If an open parenthesis is there on top of the stack then push the operator into the stack.**
* **If the input is over, pop all the remaining symbols from the stack and append them to the postfix.**

**EXAMPLE: - a + ( c / d ) \* ( e \* f )**

**Step1. ‘-‘ is an operator, push it into the stack**

**Stack: -**

**Step2. ‘a’ is an operand, put in the postfix expression.**

**Stack: -**

**Postfix Expression: a**

**Step3. ‘+‘ is an operator, push it into the stack**

**Stack: +**

**Postfix Expresson: a -**

**Step3. Push ‘(‘ into the stack.**

**Stack: + (**

**Postfix Expression: a -**

**Step4. ‘c’ is an operand, put in the postfix expression.**

**Stack: + (**

**Postfix Expression: a - c**

**Step5. ‘/‘ is an operator, push it into the stack**

**Stack: + ( /**

**Postfix Expression: a - c**

**Step6. ‘d’ is an operand, put in the postfix expression.**

**Stack: + ( /**

**Postfix Expression: a - c d**

**Step7: Scanned element is ‘)’ , then pop all elements till respective ‘(‘ and append popped element to postfix expression.**

**Stack: +**

**Postfix Expression: a - c d /**

**Step8: ‘\*‘ is an operator, push it into the stack.**

**Stack: + \***

**Postfix Expression: a - c d /**

**Step9: Push ‘(‘ into the stack.**

**Stack: + \* (**

**Postfix Expression: a - c d /**

**Step10. ‘e’ is an operand, put in the postfix expression.**

**Stack: + \* (**

**Postfix Expression: a - c d / e**

**Step11. ‘\*‘ is an operator, push it into the stack.**

**Stack: + \* ( \***

**Postfix Expression: a - c d / e**

**Step12. ‘f’ is an operand, put in the postfix expression.**

**Stack: + \* ( \***

**Postfix Expression: a - c d / e f**

**Step13. Scanned element is ‘)’ , then pop all elements till respective ‘(‘ and append popped element to postfix expression.**

**Stack: + \***

**Postfix Expression: a - c d / e f \***

**Step14. Pop \***

**Stack: +**

**Postfix Expression: a - c d / e f \* \***

**Step15. Pop +**

**Stack: empty**

**Postfix Expression: a - c d / e f \* \* +**

**CODE:**

**#include<stdio.h>**

**#include<ctype.h>**

**char stack[100];**

**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;**

**return 0;**

**}**

**int main()**

**{**

**char exp[100];**

**char \*e,x;**

**printf("Enter the expression:");**

**scanf("%s",exp);**

**printf("\n");**

**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)**

**{**

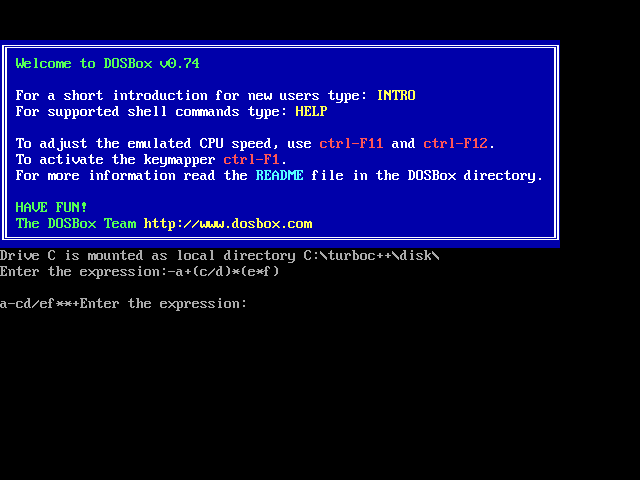
**printf("%c",pop());**

**}**

**return 0;**

**}**

**OUTPUT:**

****

**CONCLUSION:**

1. **Infix notation is the notation in which operators come between the required operands.**
2. **Postfix notation is the type of notation in which operator comes after the operand.**
3. **Infix expression can be converted to postfix expression using stack.**