

MIT WORLD PEACE UNIVERSITY

Fundamental Data Structures
Second Year B. Tech, Semester 1

EXPRESSION CONVERSION USING STACK

PRACTICAL REPORT
ASSIGNMENT 7

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

November 28, 2022

Contents

1 Aim	1
2 Objectives	1
3 Problem Statements	1
4 Theory	1
4.1 Stack	1
4.2 Expression Conversion	2
4.3 Infix to Postfix	2
4.4 Postfix to Prefix	2
5 Platform	3
6 Input	3
7 Output	3
8 Test Conditions	3
9 Code	3
9.1 Pseudo Code	3
9.1.1 Pseudo Code for checking if the stack is full	3
9.1.2 Pseudo Code for checking if the stack is empty	4
9.1.3 Pseudo Code for Pushing onto stack	4
9.1.4 Pseudo Code for popping from the stack	4
9.1.5 Pseudo Code for checking incoming character precedence and incoming sign precedence	4
9.1.6 Pseudo Code infix to postfix	5
9.1.7 Pseudo Code for postfix to infix	5
9.2 C Implementation of Problem Statement	6
9.3 Input and Output	11
10 Time Complexity	13
11 Conclusion	13
12 FAQs	13

1 Aim

Implement stack as an ADT and apply it for different expression conversions (infix to postfix or infix to prefix (Any one), prefix to postfix or prefix to infix, postfix to infix or postfix to prefix (Any one)).

2 Objectives

1. To study Stack and its operations
2. To study the importance of expression conversions

3 Problem Statements

Department of Computer Engineering has student's node named 'Pinnacle Node'. Students of second, third and final year of department can be granted membership on request. Similarly, one may cancel the membership of node. First node is reserved for president of node and last node is reserved for the secretary of the node. Write C program to maintain node members information using singly linked list. Store student PRN and Name. Write functions to:

1. *Add members as well as president or even secretary.*
2. *Compute total number of members of node*
3. *Display members*
4. *sorting of two linked list*
5. *merging of two linked list*
6. *Reversing using three pointers*
7. *Add and delete the*

4 Theory

4.1 Stack

A stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out). Mainly the following three basic operations are performed in the stack:

1. *Push*: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
2. *Pop*: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
3. *Peek or Top*: Returns top element of stack.
4. *isEmpty*: Returns true if stack is empty, else false.

4.2 Expression Conversion

Expression conversion is the process of converting an expression from one form to another. The following are the different forms of expression conversion:

1. *Infix to Postfix*: Infix expression is converted to postfix expression.
2. *Infix to Prefix*: Infix expression is converted to prefix expression.
3. *Postfix to Infix*: Postfix expression is converted to infix expression.
4. *Postfix to Prefix*: Postfix expression is converted to prefix expression.
5. *Prefix to Infix*: Prefix expression is converted to infix expression.
6. *Prefix to Postfix*: Prefix expression is converted to postfix expression.

4.3 Infix to Postfix

The algorithm for conversion of infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
 - (a) If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
 - (b) Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ')', push it to the stack.
5. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

4.4 Postfix to Prefix

The algorithm for conversion of postfix expression to prefix expression is as follows:

1. Read the postfix expression from right to left.
2. If the symbol is an operand, then push it onto the Stack.
3. If the symbol is an operator, then pop two operands from the Stack

4. Create a string by concatenating the two operands and the operator before them.
5. string = operator + operand1 + operand2
6. And push the resultant string back to Stack
7. Repeat the above steps until end of Prefix expression.

5 Platform

Operating System: Arch Linux x86-64

IDEs or Text Editors Used: Visual Studio Code

Compilers : gcc on linux for C

6 Input

- Elements of the Stack

7 Output

- Menu to display all the operations you can perform on the Stack.
- Display of All the elements of the Stack.

8 Test Conditions

1. Input at least 5 records.
2. Inserting an Element at All Positions
3. Delete an Element from All positions

9 Code

9.1 Pseudo Code

9.1.1 Pseudo Code for checking if the stack is full

```
1  isFull()
2      if (top == MAX_SIZE - 1)
3          return 1;
4      else
5          return 0;
```

9.1.2 Pseudo Code for checking if the stack is empty

```
1  isEmpty()
2      if (top == -1)
3          return 1;
4      else
5          return 0;
```

9.1.3 Pseudo Code for Pushing onto stack

```
1  push(char item)
2      if (!isFull())
3          top++;
4          stack[top] = item;
5      else
6          printf("\nSTACK OVERFLOW!\n");
```

9.1.4 Pseudo Code for popping from the stack

```
1  pop()
2      if (isEmpty())
3          printf("Stack is Empty \n\n STACK UNDERFLOW!!");
4          return 0;
5      else
6          printf("Removed this thing %c\n", stack[top]);
7          top--;
8          return stack[top + 1];
9
```

9.1.5 Pseudo Code for checking incoming character precedence and incoming sign precedence

```
1
2  icp(char ch) // incoming char precedence
3      if (ch == '+' || ch == '-')
4          return 1;
5      if (ch == '*' || ch == '/')
6          return 2;
7      if (ch == '^')
8          return 4;
9      if (ch == '(')
10         return 5;
11     else
12         return 0;
13  isp(char ch) // incoming sign precedence
14      if (ch == '+' || ch == '-')
15          return 1;
16      if (ch == '*' || ch == '/')
17          return 2;
18      if (ch == '^')
19          return 3;
20     else
21         return 0;
```

9.1.6 Pseudo Code infix to postfix

```
1  infix_to_postfix(char inexpr[10])
2      int postexp[10];
3      int k = 0, i = 0;
4      char tkn = inexpr[i];
5      while (tkn != '\0')
6          if (tkn >= 97 && tkn <= 122)
7              postexp[k] = inexpr[i];
8              k++;
9          else
10             if (tkn == '(')
11                 push('(');
12             else
13                 if (tkn == ')')
14                     while ((tkn = pop()) != '(')
15                         postexp[k] = tkn;
16                         k++;
17                 else
18                     while (!isEmpty() && isp(stack[top] >= icp(tkn))
19                         postexp[k] = pop();
20                         k++;
21                     push(tkn);
22             i++;
23             tkn = inexpr[i];
24         while (!isEmpty())
25             postexp[k] = pop();
26             k++;
27         postexp[k] = '\0';
28         for (int i = 0; i < k; i++)
29             printf("%c", postexp[i]);
30
```

9.1.7 Pseudo Code for postfix to infix

```
1  postfix_to_infix(char post[MAX_SIZE])
2      for (int i = 0; post[i] != '\0'; i++)
3          if (post[i] >= 97 && post[i] <= 127)
4              temp[0] = post[i];
5              temp[1] = '\0';
6              push_str(temp);
7              // temp[0] = '\0';
8          else
9              temp = pop_str();
10             temp1 = pop_str();
11             temp2[0] = post[i];
12             temp2[1] = '\0';
13             strcpy(inf, "(");
14             strcat(inf, temp1);
15             strcat(inf, temp2);
16             strcat(inf, temp);
17             strcat(inf, ")");
18             push_str(inf);
19         inf = pop_str();
20         printf("\nThe infix expression is: ");
21         printf("\n%s", inf);
22
```

9.2 C Implementation of Problem Statement

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #define MAX_SIZE 10
5 char stack[MAX_SIZE];
6 char stack_str[MAX_SIZE][MAX_SIZE];
7 int top = -1;
8 int top_str = -1;
9
10 int isFull()
11 {
12     if (top == MAX_SIZE - 1)
13         return 1;
14     else
15         return 0;
16 }
17 int isEmpty()
18 {
19     if (top == -1)
20         return 1;
21     else
22         return 0;
23 }
24 int isFull_str()
25 {
26     if (top_str == MAX_SIZE - 1)
27         return 1;
28     else
29         return 0;
30 }
31 int isEmpty_str()
32 {
33     if (top_str == -1)
34         return 1;
35     else
36         return 0;
37 }
38 int push(char item)
39 {
40     if (!isFull())
41     {
42         top++;
43         stack[top] = item;
44     }
45     else
46     {
47         printf("\nSTACK OVERFLOW!\n");
48     }
49 }
50 int pop()
51 {
52     if (isEmpty())
53     {
54         printf("Stack is Empty \n\n STACK UNDERFLOW!!");
55         return 0;
56     }
57     else
```



```
58     {
59         printf("Removed this thing %c\n", stack[top]);
60         top--;
61         return stack[top + 1];
62     }
63 }
64
65 char *pop_str()
66 {
67     char *str = (char*) malloc(sizeof(MAX_SIZE));
68     int st = isEmpty_str();
69     if(st == 1){
70         printf("\n Stack is Empty");
71     }
72     else{
73         strcpy(str, stack_str[top_str--]);
74         printf("%s", str);
75         return str;
76         // strcpy(str, temp)
77     }
78 }
79
80 void push_str(char str[MAX_SIZE])
81 {
82     int st = isFull_str();
83     if(st == 1){
84         printf("\n Stack is full");
85     }
86     else{
87         strcpy(stack_str[++top_str], str);
88     }
89 }
90
91 void display_stack()
92 {
93     printf("\n");
94     for (int i = top; i >= 0; i--)
95     {
96         printf("%c\n", stack[i]);
97     }
98     printf("\n");
99 }
100
101 void display_stack_str()
102 {
103     printf("\n");
104     for (int i = top; i >= 0; i--)
105     {
106         printf("%c\n", stack_str[i]);
107     }
108     printf("\n");
109 }
110
111 int icp(char ch) // incoming char precedence
112 {
113     if (ch == '+' || ch == '-')
114         return 1;
115     if (ch == '*' || ch == '/')
116         return 2;
```

```
117     if (ch == '^')
118         return 4;
119     if (ch == '(')
120         return 5;
121     else
122         return 0;
123 }
124 int isp(char ch) // incoming sign precedence
125 {
126     if (ch == '+' || ch == '-')
127         return 1;
128     if (ch == '*' || ch == '/')
129         return 2;
130     if (ch == '^')
131         return 3;
132     else
133         return 0;
134 }
135
136 // 1
137 int infix_to_postfix(char inexp[10])
138 {
139     int postexp[10];
140     int k = 0, i = 0;
141     char tkn = inexp[i];
142     while (tkn != '\0')
143     {
144         if (tkn >= 97 && tkn <= 122)
145         {
146             postexp[k] = inexp[i];
147             k++;
148         }
149         else
150         {
151             if (tkn == '(')
152             {
153                 push('(');
154             }
155             else
156             {
157                 if (tkn == ')')
158                 {
159                     while ((tkn = pop()) != '(')
160                     {
161                         postexp[k] = tkn;
162                         k++;
163                     }
164                 }
165                 else
166                 {
167                     while (!isEmpty() && isp(stack[top] >= icp(tkn)))
168                     {
169                         postexp[k] = pop();
170                         k++;
171                     }
172                     push(tkn);
173                 }
174             }
175         }
176     }
177 }
```

```
176         i++;
177         tkn = inexp[i];
178     }
179     while (!isEmpty())
180     {
181         postexp[k] = pop();
182         k++;
183     }
184     postexp[k] = '\0';
185     for (int i = 0; i < k; i++)
186     {
187         printf("%c", postexp[i]);
188     }
189 }
190
191 // 2
192 int postfix_to_infix(char post[MAX_SIZE])
193 {
194     char *temp, *temp1, *temp2, *inf;
195     for (int i = 0; post[i] != '\0'; i++)
196     {
197         if (post[i] >= 97 && post[i] <= 127)
198         {
199             temp[0] = post[i];
200             temp[1] = '\0';
201             push_str(temp);
202             // temp[0] = '\0';
203         }
204         else{
205             temp = pop_str();
206             temp1 = pop_str();
207             temp2[0] = post[i];
208             temp2[1] = '\0';
209             strcpy(inf, "(");
210             strcat(inf, temp1);
211             strcat(inf, temp2);
212             strcat(inf, temp);
213             strcat(inf, ")");
214             push_str(inf);
215         }
216     }
217     inf = pop_str();
218     printf("\nThe infix expression is: ");
219     printf("\n%s", inf);
220 }
221 int main()
222 {
223     int choice = 0;
224     char temp;
225     char fix[10];
226
227     while (choice != 8)
228     {
229
230         printf("Enter what you want to do: \n\
231 1. Push Element to the stack\n\
232 2. Pop Element from the stack\n\
233 3. See the Stack\n\
234 4. Check if stack is empty\n\
```

```
235     5. Check if stack is full\n\  
236     6. Infix to Postfix\n\  
237     7. Postfix to Prefix\n\  
238     8. Exit\n\n");  
239  
240     scanf("%d", &choice);  
241     switch (choice)  
242     {  
243     case 1:  
244         printf("Enter the element you want to add\n");  
245         scanf(" %c", &temp);  
246         push(temp);  
247         display_stack();  
248         break;  
249     case 2:  
250         printf("Removing the top element from the stack\n");  
251         pop();  
252         display_stack();  
253         break;  
254     case 3:  
255         printf("Here is the stack: \n");  
256         display_stack();  
257         break;  
258     case 4:  
259         if (isEmpty())  
260         {  
261             printf("Yup, stack is empty\n");  
262         }  
263         else  
264         {  
265             printf("Nope stack isnt empty\n");  
266             display_stack();  
267         }  
268         break;  
269     case 5:  
270         if (isFull())  
271         {  
272             printf("\nYes the Stack is full, if you add anything else, it will  
result in stackoverflow!\n");  
273         }  
274         else  
275         {  
276             printf("No stack isnt full!\n");  
277         }  
278         break;  
279     case 6:  
280         printf("\ninfix to postfix\n");  
281         scanf("%s", fix);  
282         infix_to_postfix(fix);  
283         display_stack();  
284         break;  
285     case 7:  
286         printf("\npostfix to infix\n");  
287         scanf("%s", fix);  
288         postfix_to_infix(fix);  
289         display_stack_str();  
290         break;  
291     default:  
292         printf("\nThank You\n");
```

```
293         break;
294     }
295 }
296 return 0;
297 }
```

Listing 1: Main.Cpp

9.3 Input and Output

```
1 Enter what you want to do:
2     1. Push Element to the stack
3     2. Pop Element from the stack
4     3. See the Stack
5     4. Check if stack is empty
6     5. Check if stack is full
7     6. Infix to Postfix
8     7. Postfix to Prefix
9     8. Exit
10
11 1
12 Enter the element you want to add
13 1
14
15 1
16
17 Enter what you want to do:
18     1. Push Element to the stack
19     2. Pop Element from the stack
20     3. See the Stack
21     4. Check if stack is empty
22     5. Check if stack is full
23     6. Infix to Postfix
24     7. Postfix to Prefix
25     8. Exit
26
27 1
28 Enter the element you want to add
29 2
30
31 2
32 1
33
34 Enter what you want to do:
35     1. Push Element to the stack
36     2. Pop Element from the stack
37     3. See the Stack
38     4. Check if stack is empty
39     5. Check if stack is full
40     6. Infix to Postfix
41     7. Postfix to Prefix
42     8. Exit
43
44 3
45 Here is the stack:
46
47 2
48 1
49
```

FDS Assignment 7

```
50 Enter what you want to do:
51     1. Push Element to the stack
52     2. Pop Element from the stack
53     3. See the Stack
54     4. Check if stack is empty
55     5. Check if stack is full
56     6. Infix to Postfix
57     7. Postfix to Prefix
58     8. Exit
59
60 2
61 Removing the top element from the stack
62 Removed this thing 2
63
64 1
65
66 Enter what you want to do:
67     1. Push Element to the stack
68     2. Pop Element from the stack
69     3. See the Stack
70     4. Check if stack is empty
71     5. Check if stack is full
72     6. Infix to Postfix
73     7. Postfix to Prefix
74     8. Exit
75
76 4
77 Nope stack isnt empty
78
79 1
80
81 Enter what you want to do:
82     1. Push Element to the stack
83     2. Pop Element from the stack
84     3. See the Stack
85     4. Check if stack is empty
86     5. Check if stack is full
87     6. Infix to Postfix
88     7. Postfix to Prefix
89     8. Exit
90
91 5
92 No stack isnt full!
93 Enter what you want to do:
94     1. Push Element to the stack
95     2. Pop Element from the stack
96     3. See the Stack
97     4. Check if stack is empty
98     5. Check if stack is full
99     6. Infix to Postfix
100    7. Postfix to Prefix
101    8. Exit
102
103 6
104
105 infix to postfix
106 a+b
107 Removed this thing +
108 Removed this thing 1
```

```
109 ab+1
110
111 Enter what you want to do:
112     1. Push Element to the stack
113     2. Pop Element from the stack
114     3. See the Stack
115     4. Check if stack is empty
116     5. Check if stack is full
117     6. Infix to Postfix
118     7. Postfix to Prefix
119     8. Exit
120
121 7
122
123 postfix to infix
124 ab+
125 ba(a+b)
126 The infix expression is:
127 (a+b)
128
129 Enter what you want to do:
130     1. Push Element to the stack
131     2. Pop Element from the stack
132     3. See the Stack
133     4. Check if stack is empty
134     5. Check if stack is full
135     6. Infix to Postfix
136     7. Postfix to Prefix
137     8. Exit
138
139 8
140
141 Thank You
```

Listing 2: Output

10 Time Complexity

- **Insertion:** $O(1)$
- **Deletion:** $O(1)$
- **Searching:** $O(n)$

11 Conclusion

- **Pros:** Easy to implement, Easy to understand, Easy to use.
- **Cons:** No random access, No reverse access, No search.

12 FAQs

1. What is the advantage of prefix and postfix over infix expression?

The advantage of prefix and postfix over infix expression is that there is no need of brackets in prefix and postfix expression. So there is no problem of precedence.

2. Explain how postfix/prefix expression is evaluated.

Postfix expression is evaluated by scanning the expression from left to right. If the scanned character is an operand, push it onto the stack. If the scanned character is an operator, pop two operands from the stack and apply the operator on them. Push the result back onto the stack. Repeat the steps until the end of the expression. The result obtained at the end of the expression is the final result.

3. What is ISP and ICP?

In the process of creating machine code from source code, compilers translate infix expressions to postfix expressions. Uses the 2 notions of precedence:

- incoming sign precedence $\text{isp}()$
- incoming character Precedence $\text{icp}()$.

ISP is the precedence of the operator in the stack and ICP is the precedence of the operator in the scanned expression.

4. Give various applications of stack

- Balancing of symbols:** In this application, we check whether the given expression has balanced symbols or not. For example, in the expression $(a+b)*(c-d)$, symbols are balanced. But in the expression $(a+b*(c-d)$, symbols are not balanced.
- Infix to Postfix Conversion:** In this application, we convert the given infix expression to postfix expression. For example, the infix expression $a+b*c-d/e$ is converted to $abc*+de/-$. The postfix expression is evaluated easily as compared to infix expression.
- Evaluation of Postfix Expression:** In this application, we evaluate the given postfix expression. For example, the postfix expression $abc*+de/-$ is evaluated as $(a+b*c)-(d/e)$.
- Redo-undo:** In this application, we can undo the last performed operation and redo it again.
- Forward and backward feature in web browsers:** In this application, we can go to the previous web page and next web page.
- Infix to Prefix Conversion:** In this application, we convert the given infix expression to prefix expression. For example, the infix expression $a+b*c-d/e$ is converted to $-+a*bc/de$. The prefix expression is evaluated easily as compared to infix expression.
- Evaluation of Prefix Expression:** In this application, we evaluate the given prefix expression. For example, the prefix expression $-+a*bc/de$ is evaluated as $(a+b*c)-(d/e)$.
- Parenthesis Matching:** In this application, we check whether the given expression has balanced parenthesis or not. For example, in the expression $(a+b)*(c-d)$, parenthesis are balanced. But in the expression $(a+b*(c-d)$, parenthesis are not balanced.
- Tower of Hanoi:** In this application, we solve the Tower of Hanoi problem using stack.

- (j) **Expression Conversion:** In this application, we convert the given expression from one form to another form. For example, we can convert the given infix expression to postfix expression or prefix expression.

5. Why is stack used in expression conversion?

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form. As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., $A+B$, $C*D$, D/A etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. $(A+B)*C(D/(J+D))$.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

6. Give stack full and stack empty conditions for stack.

if $Top == size - 1$, then stack is full. If $top == -1$ then stack is empty.