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 27, 2022

Contents

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 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

- Atleast 5 Elements to Input, including the President and the Secretary
- Details of Every Element like Name and PRN
- Options to Select what to do.

7 Output

- Menu to display all the operations you can perform on the Linked list.
- Display of All the elements of the Linked list, before and after performing operations on it.

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

```
isFull()
  if (top == MAX_SIZE - 1)
    return 1;
  else
    return 0;
```

9.1.2 Pseudo Code for checking if the stack is empty

```
isEmpty()
  if (top == -1)
    return 1;
  else
    return 0;
```

9.1.3 Pseudo Code for Pushing onto stack

```
push(char item)
  if (!isFull())
    top++;
    stack[top] = item;
  else
    printf("\nSTACK OVERFLOW!\n");
```

9.1.4 Pseudo Code for popping from the stack

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

```

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

```

icp(char ch) // incoming char precedence
  if (ch == '+' || ch == '-')
    return 1;
  if (ch == '*' || ch == '/')
    return 2;
  if (ch == '^')
    return 4;
  if (ch == '(')
    return 5;
  else
    return 0;
isp(char ch) // incoming sign precedence
  if (ch == '+' || ch == '-')
    return 1;
  if (ch == '*' || ch == '/')
    return 2;
  if (ch == '^')
    return 3;
  else
    return 0;
```

9.1.6 Pseudo Code infix to postfix

```
infix_to_postfix(char inexp[10])
  int postexp[10];
  int k = 0, i = 0;
  char tkn = inexp[i];
  while (tkn != '\0')
    if (tkn >= 97 && tkn <= 122)
      postexp[k] = inexp[i];
      k++;
    else
      if (tkn == '(')
        push('(');
      else
        if (tkn == ')')
          while ((tkn = pop()) != '(')
            postexp[k] = tkn;
            k++;
        else
          while (!isEmpty() && isp(stack[top] >= icp(tkn)))
            postexp[k] = pop();
            k++;
          push(tkn);
    i++;
    tkn = inexp[i];
  while (!isEmpty())
    postexp[k] = pop();
    k++;
  postexp[k] = '\0';
  for (int i = 0; i < k; i++)
    printf("%c", postexp[i]);
```

9.1.7 Pseudo Code for postfix to infix

```
postfix_to_infix(char post[MAX_SIZE])
  for (int i = 0; post[i] != '\0'; i++)
    if (post[i] >= 97 && post[i] <= 127)
      temp[0] = post[i];
      temp[1] = '\0';
      push_str(temp);
      // temp[0] = '\0';
    else
      temp = pop_str();
      temp1 = pop_str();
      temp2[0] = post[i];
      temp2[1] = '\0';
      strcpy(inf, "(");
      strcat(inf, temp1);
      strcat(inf, temp2);
      strcat(inf, temp);
      strcat(inf, ")");
      push_str(inf);
  inf = pop_str();
  printf("\nThe infix expression is: ");
  printf("\n%s", inf);
```

9.2 C Implementation of Problem Statement

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_SIZE 10
char stack[MAX_SIZE];
char stack_str[MAX_SIZE][MAX_SIZE];
int top = -1;
int top_str = -1;

int isFull()
{
    if (top == MAX_SIZE - 1)
        return 1;
    else
        return 0;
}
int isEmpty()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
int isFull_str()
{
    if (top_str == MAX_SIZE - 1)
        return 1;
    else
        return 0;
}
int isEmpty_str()
{
    if (top_str == -1)
        return 1;
    else
        return 0;
}
int push(char item)
{
    if (!isFull())
    {
        top++;
        stack[top] = item;
    }
    else
    {
        printf("\nSTACK OVERFLOW!\n");
    }
}
int pop()
{
    if (isEmpty())
    {
        printf("Stack is Empty \n\n STACK UNDERFLOW!!");
        return 0;
    }
    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;
```

```
    if (ch == '^')
        return 4;
    if (ch == '(')
        return 5;
    else
        return 0;
}
int isp(char ch) // incoming sign precedence
{
    if (ch == '+' || ch == '-')
        return 1;
    if (ch == '*' || ch == '/')
        return 2;
    if (ch == '^')
        return 3;
    else
        return 0;
}

// 1
int infix_to_postfix(char inexp[10])
{
    int postexp[10];
    int k = 0, i = 0;
    char tkn = inexp[i];
    while (tkn != '\0')
    {
        if (tkn >= 97 && tkn <= 122)
        {
            postexp[k] = inexp[i];
            k++;
        }
        else
        {
            if (tkn == '(')
            {
                push('(');
            }
            else
            {
                if (tkn == ')')
                {
                    while ((tkn = pop()) != '(')
                    {
                        postexp[k] = tkn;
                        k++;
                    }
                }
                else
                {
                    while (!isEmpty() && isp(stack[top] >= icp(tkn)))
                    {
                        postexp[k] = pop();
                        k++;
                    }
                    push(tkn);
                }
            }
        }
```

```
        i++;
        tkn = inexp[i];
    }
    while (!isEmpty())
    {
        postexp[k] = pop();
        k++;
    }
    postexp[k] = '\0';
    for (int i = 0; i < k; i++)
    {
        printf("%c", postexp[i]);
    }
}

// 2
int postfix_to_infix(char post[MAX_SIZE])
{
    char *temp, *temp1, *temp2, *inf;
    for (int i = 0; post[i] != '\0'; i++)
    {
        if (post[i] >= 97 && post[i] <= 127)
        {
            temp[0] = post[i];
            temp[1] = '\0';
            push_str(temp);
            // temp[0] = '\0';
        }
        else{
            temp = pop_str();
            temp1 = pop_str();
            temp2[0] = post[i];
            temp2[1] = '\0';
            strcpy(inf, "(");
            strcat(inf, temp1);
            strcat(inf, temp2);
            strcat(inf, temp);
            strcat(inf, ")");
            push_str(inf);
        }
    }
    inf = pop_str();
    printf("\nThe infix expression is: ");
    printf("\n%s", inf);
}
int main()
{
    int choice = 0;
    char temp;
    char fix[10];

    while (choice != 8)
    {

        printf("Enter what you want to do: \n\
    1. Push Element to the stack\n\
    2. Pop Element from the stack\n\
    3. See the Stack\n\
    4. Check if stack is empty\n\
```

```
    5. Check if stack is full\n\
    6. Infix to Postfix\n\
    7. Postfix to Prefix\n\
    8. Exit\n\n");

        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter the element you want to add\n");
            scanf(" %c", &temp);
            push(temp);
            display_stack();
            break;
        case 2:
            printf("Removing the top element from the stack\n");
            pop();
            display_stack();
            break;
        case 3:
            printf("Here is the stack: \n");
            display_stack();
            break;
        case 4:
            if (isEmpty())
            {
                printf("Yup, stack is empty\n");
            }
            else
            {
                printf("Nope stack isnt empty\n");
                display_stack();
            }
            break;
        case 5:
            if (isFull())
            {
                printf("\nYes the Stack is full, if you add anything else, it will result in stackoverflow!\n");
            }
            else
            {
                printf("No stack isnt full!\n");
            }
            break;
        case 6:
            printf("\ninfix to postfix\n");
            scanf("%s", fix);
            infix_to_postfix(fix);
            display_stack();
            break;
        case 7:
            printf("\npostfix to infix\n");
            scanf("%s", fix);
            postfix_to_infix(fix);
            display_stack_str();
            break;
        default:
            printf("\nThank You\n");
```

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

Listing 1: Main.Cpp

9.3 Input and Output

```
Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

1
Enter the element you want to add
1

1

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

1
Enter the element you want to add
2

2
1

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

3
Here is the stack:

2
1

```

```
Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

2
Removing the top element from the stack
Removed this thing 2

1

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

4
Nope stack isnt empty

1

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

5
No stack isnt full!
Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

6

infix to postfix
a+b
Removed this thing +
Removed this thing 1
```

```
ab+1

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

7

postfix to infix
ab+
ba(a+b)
The infix expression is:
(a+b)

Enter what you want to do:
    1. Push Element to the stack
    2. Pop Element from the stack
    3. See the Stack
    4. Check if stack is empty
    5. Check if stack is full
    6. Infix to Postfix
    7. Postfix to Prefix
    8. Exit

8

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 isp()
   - incoming character Presedence 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**

   (a) **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.

   (b) **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.

   (c) **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).

   (d) **Redo-undo:** In this application, we can undo the last performed operation and redo it again.

   (e) **Forward and backward feature in web browsers:** In this application, we can go to the previous web page and next web page.

   (f) **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.

   (g) **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).

   (h) **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.

   (i) **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.