CC-216: Data Structures & Algorithms
BS SE/CS Fall 2020
Practice - 06

Issue Date: 16-May-2022

Objective:

- Exploring the LIFO behavior. Getting a drip on usage of stack in different problems.

Task – 1:

1). Check whether a string is of the form $a^n b^n$ where n = 0, 1, 2, 3, 4, ……. E.g. the string aaabbb is of $a^n b^n$ form but bbbaaa, aabbb are not.
2). Check whether a string is of the form $a^s b^t$ where s = 0, 1, 2, 3, 4, ……. and t = 0, 1, 2, 3, 4, ……. E.g. the string aaaaaabbb is of $a^s b^t$ form.
3). Check whether a string is of the form $a^n b^n c^n$ where n = 0, 1, 2, 3, 4, …….
4). Reverse the order of elements in stack S
   a. Using two additional stacks
   b. Using one additional stack and some additional non-array variables
5). Review the algorithm 'addingLargeNumbers()' discussed in Section 4.1 of Text Book-B
6). Put the elements on stack S in ascending order using one additional stack and some additional non-array variables.
7). Transfer the elements of stack S1 to Stack S2 so that the elements from S2 are in the same order as on S1
   a. using one additional Stack
   b. Using no additional stack but only some additional non-array variables
8). Convert infix to prefix
9). Checks whether a string is palindrome or not
10). Reverse the words in a given string.
11). Implement a function, which receives an infix expression and removes all the branching/parenthesis without affecting the result of expression. E.g. (a+B)-C and A+B-C are equivalent expressions.
12). Implement a stack which supports all the standard operations of LIFO in O(1) time along with getMin function (returns the minimum value in stack) in O(1)

NOTE:

- While implementing the above function you are not allowed to change the standard ADT of stack discussed in class, so all the function asked above will only be able to access the public part of Stack ADT discussed in class if needed anywhere.
- Most of the problems have been taken from Text-Book-B.

Task – 2:

You are to develop an Undo/Redo behavior as in all word processing applications.
All these applications can remember a limited number of operations. Let's try to explore the behavior of Undo/Redo in MSWord through following example:

Example: MS-Word

In MSWord 2007, whenever you perform an operation it somehow uniquely identify those operations and memorize them. By default, MS Office (word, power point, excel) save the last 100 undoable operations. https://tinyurl.com/3yvbr8f9 (https://support.microsoft.com/en-us/office/undo-redo-or-repeat-an-action-84bdb9bc-4e23-4f06-ba78-f7b893eb2d28)

An
Undo
Redo
Facility

1 MSWord Document

So, If I type the data as given on right side in MSWord:
Assumption: MSWord can save up to 10 operations at max.
I typed these words in small caps but MSWord automatically does the auto correction as well. So, there are not only four typing operations rather the sequence of operations are as follows:
Typing, auto correction, typing, auto correction, typing, auto correction, typing, auto correction, …
Notice that we haven't done any undo/redo so far. So when I do undo, the last operation will be undone i.e. in the above text the 'Facility' will become 'facility'
Another point to note is that what if I do more N=10 operations in that case 11th operations will become 10th, 10th will become 9th and so on… And 1st operations will be dropped/forgotten.

You can explore the behavior of Undo/Redo yourself.

In the following ADT we have to exhibit the same behavior. To avoid any complexity, for us, an operation is an integer, which will be sent to 'memoriseOperation'. 'undo' method will exhibit the behavior of undo as in MSWord and 'redo' will exhibit redo behavior.

```
class UndoRedo
{
        int capacity; //N // number of operations that can be memorized
        //decide rest of the data structure yourself
public:
        UndoRedo (int c = 10);
        void memoriseOperation(int op);
        int undo();
        int redo();
};
```

Task – 2:

Write a function, which receives a postfix expression in the form 134,21,+79,/3,* and return the final result.
Each operand and operator is separated by comma character.
Note: not allowed to use C++ string class anywhere.

Example 1:	***Example 2:***
134,21,+,79,/,3,* ------→ **5.88** **Steps:** • 134 + 21 = 155 • 155 / 79 = 1.96 • 1.96 * 3 = **5.88**	1.34,2.66,+,2,/,10,+ ------→ **12** **Steps:** • 1.34 + 2.66 = 4 • 4 / 2 = 2 • 2 + 10 = **12**

Task – 3:

You have to redo Task-2 considering the fact that input string will be in infix form.
Note: There will be no delimiter character like comma between operand and operators.

And a few Programming Competition Questions

1). https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=196
2). https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=455
3). https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=56

STL

Last but not the least, explore: https://www.cplusplus.com/reference/stack/stack/

Issue Date: 18-May-2022

Expression Evaluation Algorithms

Homogeneous Brackets Validity

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' )
        {
                push next_character into stack
        }
        if ( next_character is closing bracket ')' )
        {
                if (stack is empty)
                        return false
                stacktop = pop from stack
                if (stacktop is exactly opposite of next_character)
                {
                        pop from stack
                }
                else
                        return false
        }

}
if (stack is empty)
        return true
else
        return false
```

****

Heterogeneous Brackets Validity

```
Input = Infix Expression
Output = 1 means valid 0 means invalid expression
Create a stack that can hold opening brackets in it
While (Scan the infixstring from left to left till the end)
{
        get next_character from infixstring
        if ( next_character is opening bracket '(' OR '{' OR '[')
        {
                push next_character into stack
        }
        if ( next_character is closing bracket ')' OR '}' OR ']')
        {
                if (stack is empty)
                        return false
                stacktop = pop from stack
                if (! (stacktop is exactly opposite of next_character ) )
                {
                        Return false
                }
        }

        if (stack is empty)
                return true
        else
                return false
}
```

****

Infix To Postfix Conversion (Reverse Polish Notation (RPN))

__Reverse Polish notation (RPN)__ is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation, which puts the operator in the prefix position. It is also known as __postfix notation__ and is parenthesis-free as long as operator arities are fixed. The description "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented (prefix) Polish notation in the 1920s.

The reverse Polish scheme was proposed in 1954 by Burks, Warren, and Wright[1] and was independently reinvented by F. L. Bauer and E. W. Dijkstra in the early 1960s to reduce computer memory access and utilize the stack to evaluate expressions. The algorithms and notation for this scheme were extended by Australian philosopher and computer scientist Charles Hamblin in the mid-1950s.[2][3]

During the 1970s and 1980s, RPN was known to many calculator users, as it was used in some handheld calculators of the time designed for advanced users: for example, the HP-10C series and Sinclair Scientific calculators. [see http://en.wikipedia.org/wiki/Reverse_Polish_notation for detail]

Algorithm

```
Input = infix string
Output = postfix string
Create a stack that can store operators in it
While (Scan the infix_string from left to right till the end)
{
        get next_character from infix_string
        if (next_character is operand)
        {
                Append next_character to postfix_string
        }
        else if (next_character is operator)
        {
                while (stack is not empty AND precedence(stacktop) > precedence(next_character))
                {
                        pop the operator from stack and append it to postfix_string
                }
                if ( next_character is not ')' )
                {
                        push next_character to stack
                }
                else if ( next_character is ')' )
                {
                        pop from stack //it will pop '(' bracket
                }
        }
}
while(stack is not empty)
{
        pop the operator from stack and append it to postfix_string
}
```

Some Rules about Precedence and about push and pop of operators in stack

- Closing bracket can never be pushed in stack.
- If operand then append in postfix string.
- If operator then push in stack.
- A low precedence operator can never on top of high precedence operator in the stack.
  E.g. if in stack, the operator are in this order from bottom to top
  + / ....this is right but
  / + ...this is wrong and
  / / ....this is also wrong but what about this
  + / (+ ....this is also right
  It means that this rules implements in the stack from the start of an '(' till the next '(' occurs.
- If operator is opening bracket then push it in stack but If operator is closing bracket ')' then pop the stack until '(' not found in the stack.
  After popping all the operators until '('also pop that '('.

****

Postfix Expression Evaluation

```
Input = postfix string
Output = result of postfix expression
Create a stack that can store operands in it
While (Scan the postfixstring from left to right till the end)
{
        get next_character from postfix_string
        if (next_character is operand)
        {
                push next_charcter into stack.
        }
        else if (next_character is operator)
        {
                operand2 = pop operand from stack
                operand1 = pop operand from stack
                Perform operation on the operand1 and operand2 depending on the operator.
                Push the result of this operation in stack.
        }
}
Pop the result of "Postfix Expression Evaluation" from stack.
```

****

Infix To Prefix Conversion (Polish Notation (PN))

Part (a) using two stack

```
Input = infix string
Output = postfix string
Create a stack that can store operators in it called it as operatorstack
Create a stack that can store operands in it called it as operandstack
While (Scan the infix_string from left to right till the end)
{
        get next_character from infix_string
        if (next_character is operand)
        {
                push next_character into operandstack
        }
        else if (next_character is operator)
        {
                while (operatorstack is not empty AND precedence(stacktop) >
                        precedence(next_character) )
                {
                        operand2 = pop the operand from operandstack
                        operand1 = pop the operand from operandstack
                        operator = pop the operator from operatorstack
                        concatenate operator+operand1+operand2
                        push the concatenated result into operandstack
                }
                if ( next_character is not ')' )
                {
                        push next_character to operatorstack
                }
                else if ( next_character is ')' )
                {
                        pop from operatorstack //it will pop '(' bracket
                }
        }
}
while(operatorstack is not empty)
{
        operand2 = pop the operand from operandstack
        operand1 = pop the operand from operandstack
        operator = pop the operator from operatorstack
        concatenate operator+operand1+operand2
        push the concatenated result into operandstack
}
Pop the operandstack and store it in postfix string.
```

Some Rules about Precedence and about push and pop of operators in stack

- Precedence Rules are Same as for Postfix conversion

- Its advantage is that you get same string as you draw through pen and paper.
  e.g. try to find out prefix expression for "A$B*C-D+E/F/(G+H)"
  Manually it comes "-*$ABC+D//EF+GH"
  Applying Part (a) the same result will be produced
  But applying part (b) "-*$ABC+D/E/F+GH" is produced.

****

Part (b) using one stack

```
Input = Infix string
Output = Prefix string
Create a stack that can store operators in it
While (Scan the postfix_string from right to left till the end)
{
        get next_character from infix_string
        if (next_character is operand)
        {
                Append next_character to postfix_string
        }
        else if (next_character is operator)
        {
                while (stack is not empty AND precedence(stacktop) > precedence(next_character))
                {
                        pop the operator from stack and append it to prefix_string
                }
                if ( next_character is not '(' )
                {
                        push next_character to stack
                }
                else if ( next_character is '(' )
                {
                        pop from stack //it will pop ')' bracket
                }
        }
}
while(stack is not empty)
{
        pop the operator from stack and append it to prefix_string
}
```

Some Rules about Precedence and about push and pop of operators in stack

- Opening bracket can never be pushed in stack.
- If operand then append in prefix string.
- If operator then push in stack.
- A low precedence operator can never on top of high precedence operator in the stack.
  E.g. if in stack the operator are in this order from bottom to top
  + / ....this is right but
  / + ...this is wrong and
  / / ....this is also wrong but what about this
  + /) + ...this is also right
  It means that this rules implements in the stack from the start of an ')' till the next ')' occurs.
- If operator is closing bracket then push it in stack but If operator is opening bracket '(' then pop the stack until ')' not found in the stack.
- After popping all the operators until ')' found, also pop that ')'.

Disadvantage:

- You will not get exactly the same prefix expression as you find out manually.

****

Prefix Expression Evaluation
Prefix Evaluation for expression produced in part (b) of Infix into Prefix Conversion

```
Input = prefix string
Output = result of prefix expression
Create a stack that can store operands in it
While (Scan the prefixstring from right to left till the end)
{
        get next_character from prefix_string
        if (next_character is operand)
        {
                push next_charcter into stack.
        }
        else if (next_character is operator)
        {
                operand1 = pop operand from stack
                operand2 = pop operand from stack
                Perform operation on the operand1 and operand2 depending on the operator.
                Push the result of this operation in stack.
        }
}
Pop the result of "Prefix Expression Evaluation" from stack.
```

****

Advantage of Evaluating Expressions, which are in Postfix and Prefix form

- No nesting of brackets
- Both prefix and postfix form expressions can be evaluating in one pass (reading the expression only once) rather than multiple passes in Infix Expression.

→ For interest: you may also look at shunting yard algorithm (a method for parsing mathematical expressions specified in infix notation) invented by Edsger Dijkstra http://en.wikipedia.org/wiki/Shunting-yard_algorithm