**Theory assignment 2**

Question : 6

**Algorithm of sorting a stack of integers using another stack:**

**Step 1:** Create two stacks, 'Stx' to hold the original sorted list and hold the sorted elements 'Sty'.

**Step 2:** Initialize 'Stx' with the given list of integers [3,4,6,2,5] and Set 'Sty' to an empty list [].

**Step 3:** 'Stx' is not empty, do the following:

1. When 'Stx' is not empty, pop the top element of 'Stx' & put it in a temporary variable called temp.
2. If 'Sty' is not empty and the top element of 'Sty' is larger than 'temp', do the following :

* pop the top element of 'Sty' & push the temp onto 'Stx'
* push the temp onto 'Sty' .

1. else Push the temp onto 'Sty'.

**Step 4:** Continue Step 3 until 'Stx’ is empty.

**Step 5:** 'Sty' will now have a growing elements in ascending order.

The steps can be imagined as follows:

Stx temp Sty

[3,4,6,2,5] [ ] [ ]

[3,4,6,2] [5] [5]

[3,4,6] [2] [2,5]

[3,4] [6] [2,5,6]

[3] [4] [2,4,5,6]

[ ] [3] [2,4,5,6]

[ ] [ ] [2,3,4,5,6]

7.

**Algorithm of converting the infix expression to postfix expression using a stack:**

**Step 1:** Initialize an empty stack and an empty postfix expression.

**Step 2:** Start parsing the expression from left to right.

**Step 3:** If you encounter an operand (eg, ‘a’ to ‘z’), add it to the postfix expression(String).

**Step 4:** If you encounter an operator (eg, +, -, \*, /), pop all operators from the stack that have greater or equal precedence than the current operator and append them to the postfix expression, then push the current operator onto the stack.

**Step 5:** Repeat steps 3 and 4 until you reach the end of the expression.

**Step 6:** Pop all the remaining operators from the stack and add them to the postfix expression.

The expression: **a+b\*c+d\*e**

**Step -1:** Creat Stack ST: [empty]

**Step-2:** Create String S: [empty]

**Step-3:** Reading a, add to S: Stack: [empty] S: [a]

**Step-4:** Reading +, add to stack: Stack: [+] S: [a]

**Step-5:** Reading b, add to S: Stack: [+] S: [a, b]

**Step-6:** Reading \*, add to stack: Stack: [+, \*] S: [ab]

**Step-7:** Reading c, add to S: Stack: [+, \*] S: [abc]

**Step-8:** Reading +, pop \* and add to postfix, add + to stack: Stack: [+] S: [abc\*]

**Step-9:** Reading d, add to S: Stack: [+] S: [abc \*d]

**Step-10:** Reading \*, add to stack: Stack: [+, \*] S: [abc \*d]

**Step-11:** Reading e, add to S: Stack: [+, \*] S: [abc \*de]

**Step-12:** End of expression: Stack: [+, \*] S: [abc\*de]

**Step-13:** Pop \* and add to S: Stack: [+] S: [ abc\*de\*]

**Step-14:** Pop + and add to S: Stack: [empty] S: [abc\*de\*+]

**Step-15:** End of stack: Stack: [empty] S: [abc\*de\*+]

**Postfix expression:** **abc\*de\*+**

1. Which of the best in array based stack implementation and linked-list-based stack implementation is good when my stack needs random access? Explain the reasons.

**Array-based implementation** is better when we stack requires random access because array allows constant access to any element, where link-lists only allow single-side sequential access.

The entire list should be exceeded for random access to the Linked-List material, which takes linear time. The reason for this is that a dynamic data structure is a dynamic data structure that does not have an adjacent block of memory like array, instead points each element to the latter.

In contrast, there is an adjacent block of the memory of the arrays and thus allowing the constant access to any element through index. This makes arrays more suitable for situations where fast and skilled random access is needed, such as a stack implementation where we may need to access the components at any time of the stack.

2.

|  |  |
| --- | --- |
| **Stack Operation** | **Time Complexity** |
| PUSH | O(1) |
| POP | O(1) |
| TOP | O(1) |

3.

4.A postfix expression is an arithmetic expression in which the operators appear after the operands. It is also known as Reverse Polish Notation (RPN).

The need for postfix expressions arises because it eliminates the need for parentheses and is easier for computers to parse. This makes it easy to evaluate expressions using stack data structures.

abc\*+de\*+

Here are all the steps for the given expression:

**Step-1:** Create an empty stack named st.

**Step-2:** Read the first letter a. It's an operand, so push it onto the stack. Stack: [a].

**Step-3:** Read the second letter b. It's an operand, so push it onto the stack. Stack: [a, b].

**Step-4:** Read the third letter c. It's an operand, so push it onto the stack. Stack: [a, b, c].

**Step-5:** Read the fourth letter \*. It's an operator, so pop the top two operands (c and b) off the stack, perform the operation (c \* b), and push the result onto the stack. Stack: [a, c \* b].

**Step-6:** Read the fifth letter +. It's an operator, so pop the top two operands (c \* b and a) off the stack, perform the operation (c \* b + a), and push the result onto the stack. Stack: [c \* b + a].

**Step-7:** Read the sixth letter d. It's an operand, so push it onto the stack. Stack: [c \* b + a, d].

**Step-8:** Read the seventh letter e. It's an operand, so push it onto the stack. Stack: [c \* b + a, d, e].

**Step-9:** Read the eighth letter \*. It's an operator, so pop the top two operands (e and d) off the stack, perform the operation (e \* d), and push the result onto the stack. Stack: [c \* b + a, e \* d].

**Step-10:** Read the ninth letter +. It's an operator, so pop the top two operands (e \* d and c \* b + a) off the stack, perform the operation (e \* d + c \* b + a), and push the result onto the stack. Stack: [e \* d + c \* b + a].

The expression is fully processed and the stack contains a single value, which is the result of the expression: **e \* d + c \* b + a.**

5.

**Algorithm of checking balanced parentheses using stack:**

**Step 1:** Create an empty stack.

**Step 2:** Iterate through each character of the input string:

* If the character is an opening bracket ( (,{,[ ) push it onto the stack
* If the character is a closing parenthesis ( ),},] ) pop the top item from the stack and check if it matches the current character. If it does not match, the bracket is not balanced.

Here are all the steps for the given expression:

**Input String :** ( ( [ ][ ] { ( ) } ) )

Stack: [ empty ]

**Step-1:** ( ( [ ][ ] { ( ) } ) )

Reading char: ( Stack: [ ( ]

**Step-2:** ( ( [ ][ ] { ( ) } ) )

Reading char: ( Stack: [ (, ( ]

**Step-3:** ( ( [ ][ ] { ( ) } ) )

Reading char: [ Stack: [ (, (, [ ]

**Step-4:** ( ( [ ][ ] { ( ) } ) )

Reading char: ] Stack: [ (, ( ]

**Step-5:** ( ( [ ][ ] { ( ) } ) )

Reading char: [ Stack: [ (, (, [ ]

**Step-6:** ( ( [ ][ ] { ( ) } ) )

Reading char: ] Stack: [ (, ( ]

**Step-7:** ( ( [ ][ ] { ( ) } ) )

Reading char: { Stack: [ (, (, { ]

**Step-8:** ( ( [ ][ ] { ( ) } ) )

Reading char: ( Stack: [ (, (,{ ,( ]

**Step-9:** ( ( [ ][ ] { ( ) } ) )

Reading char: ) Stack: [ (, (, { ]

**Step-10:** ( ( [ ][ ] { ( ) } ) )

Reading char: } Stack: [ (, ( ]

**Step-11:** ( ( [ ][ ] { ( ) } ) )

Reading char: ) Stack: [ ( ]

**Step-11:** ( ( [ ][ ] { ( ) } ) )

Reading char: ) Stack: [ ]

**Since the stack is empty at the end of the iteration, the parentheses are balanced.**