|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Assignment No** | **02** | |  |  | | **Title** | **Searching & Sorting** | |  |  | | **Roll No** |  | |  |  | | **Date of performance** |  | |  |  | | **Date of completion** |  | |  |  | | **Marks out of 10** |  | |  |  | | **Signature of staff** |  | |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

2. Implementation of Stack

**AIM:** Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix and prefix expression.

## **OBJECTIVE:**

1. To understand the concept of abstract data type.
2. How different data structures such as arrays and a stacks are represented as an ADT.

## **THEORY:**

1. **What is an abstract data type?**

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles. In general terms, an abstract data type is a *specification* of the values and the operations that has two properties:

* + It specifies everything you need to know in order to use the data type
  + It makes absolutely no reference to the manner in which the data type will be implemented.

When we use abstract data types, our programs divide into two pieces:

* + The Application: The part that uses the abstract data type.
  + The Implementation: The part that implements the abstract data type.

## **What is stack? Explain stack operations with neat diagrams.**

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **stack** is a last in, first out ([LIFO](http://en.wikipedia.org/wiki/LIFO_(computing))) [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) and [data](http://en.wikipedia.org/wiki/Data_structure) [structure](http://en.wikipedia.org/wiki/Data_structure). A stack can have any [abstract data type](http://en.wikipedia.org/wiki/Abstract_data_type) as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty. A pop either reveals previously concealed items, or results in an empty stack. A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest. A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and isEmpty are available, too. Also known as "last-in, first-out" or LIFO.

## **Operations**

An abstract data type (ADT) consists of a data structure and a set of **primitive**

* + **Push** adds a new element
  + **Pop** removes a element

Additional primitives can be defined:

* + **IsEmpty** reports whether the stack is empty
  + **IsFull** reports whether the stack is full
  + **Initialise** creates/initialises the stack
  + **Destroy** deletes the contents of the stack (may be implemented by re-initialising
  + the stack)

## **Explain how stack can be implemented as an ADT.**

User can Add, Delete, Search, and Replace the elements from the stack. It also checks for Overflow/Underflow and returns user friendly errors. You can use this stack implementation to perform many useful functions. In graphical mode, this C program displays a startup message and a nice graphic to welcome the user.

The program performs the following functions and operations:

* + **Push**: Pushes an element to the stack. It takes an integer element as argument. If the stack is full then error is returned.
  + **Pop**: Pop an element from the stack. If the stack is empty then error is returned. The element is deleted from the top of the stack.
  + **DisplayTop** : Returns the top element on the stack without deleting. If the stack is

empty then error is returned.

**ALGORITHM:**

**Abstract Data Type Stack:**

Define Structure for stack(Data, Next Pointer)

## **Stack Empty:**

Return True if Stack Empty else False.

Top is a pointer of type structure stack.

Empty(Top)

Step 1: If Top = = NULL Step 2: Return 1;

Step 3: Return 0;

## **Push Opertion:**

Top & Node pointer of structure Stack. Push(element)

Step1:

Node- >data=element;

Step 2:

Node->Next = Top;

Step 3: Top = Node

### **Algorithm for infix to postfix conversion**

Initialize result as a blank string, Iterate through given expression, one character at a time

1. If the character **is an operand, add it to the result.**
2. **If the character is an operator.**
   * If the operator stack is empty then push it to the operator stack.
   * Else If the operator stack is not empty,
     + If the **operator’s precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.**

**(ICP >= ISP) ---->push**

* + - If the **operator’s precedence is less than the precedence of the stack top of operator stack then “*pop*** *out an operator from the stack and add it to the result until the stack is empty or operator’s precedence is greater than or equal to the precedence of the stack top of operator stack*“. then push the operator to stack.
    - (**ICP<ISP)---->POP**

1. If the character is “(“, then push it onto the operator stack.
2. If the character is “)”, then “*pop out an operator from the stack and add it to the result until the corresponding* “*(*“ *is encountered in operator stack. Now just pop out the “(“*.

|  |  |  |  |
| --- | --- | --- | --- |
| **A \* B + C** |  |  |  |
| **current symbol** | **operator stack** | **postfix string** | **OUTPUT** |
|  |  |  |  |
| **1** | **A** |  | **A** |
| **2** | **\*** | **\*** | **A** |
| **3** | **B** | **\*** | **A B** |
| **4** | **+** | **+** | **A B \* {pop and print the '\*' before pushing the '+'}** |
| **5** | **C** | **+** | **A B \* C** |
| **6** |  |  | **A B \* C +** |

|  |  |  |  |
| --- | --- | --- | --- |
| **A + B \* C** |  |  |  |
| **current symbol** | **operator stack** | **postfix string** | **OUTPUT** |
|  |  |  |  |
| **1** | **A** |  | **A** |
| **2** | **+** | **+** | **A** |
| **3** | **B** | **+** | **A B** |
| **4** | **\*** | **+ \*** | **A B** |
| **5** | **C** | **+ \*** | **A B C** |
| **6** |  |  | **A B C \* +** |

## **Algorithm to convert infix expression to prefix**

To convert an infix to prefix expression alogorithm

**Step 1:**

Reverse the infix expression i.e A+B\*C will become C\*B+A. Note while reversing each ‘(‘ will become ‘)’ and each ‘)’ becomes ‘(‘.

**Step 2**:

Obtain the postfix expression of the modified expression i.e CB\*A+.

**Step 3**:

Reverse the postfix expression. Hence in our example prefix is +A\*BC.

|  |  |  |  |
| --- | --- | --- | --- |
| **(a+b)\*c**  **Step 1) c\*(b+a)**  **Step 2)** |  |  |  |
|  | **current symbol** | **operator stack** | **postfix string** |
|  |  |  |  |
| **1** | **c** |  | **c** |
| **2** | **\*** | **\*** | **c** |
| **3** | **(** | **(**  **\*** | **c** |
| **4** | **b** | **(**  **\*** | **cb** |
| **5** | **+** | **+**  **(**  **\*** | **cb** |
| **6** | **a** | **+**  **(**  **\*** | **cba** |
| **7** | **)** | **\*** | **Cba+** |
| **8** | **\0** |  | **Cba+\*** |

**Step 3:Reverse string- \*+abc**

**Algorithm for postfix Evalution**

EVALUATE\_PREFIX(STRING)

Step 1: Put a pointer P at the end of the end

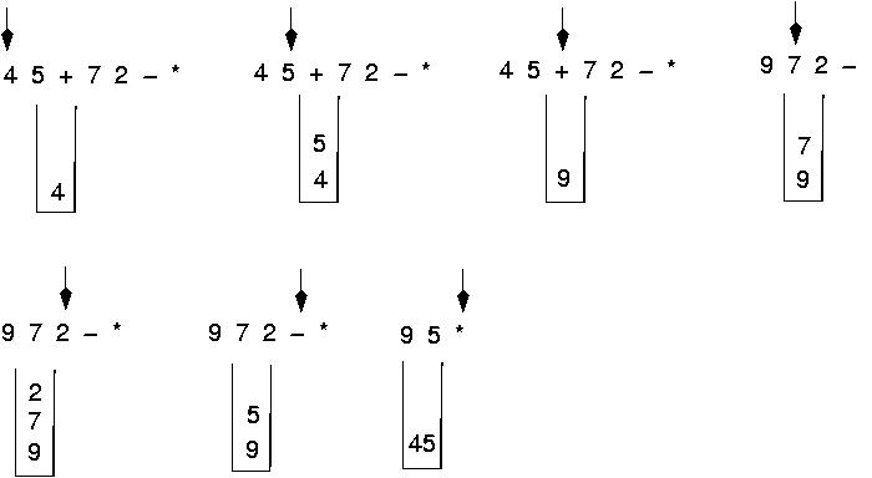
Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack, return it

Step 6: End



**Algorithm for prefix Evalution**

EVALUATE\_PREFIX(STRING)

Step 1: Put a pointer P at the end of the end

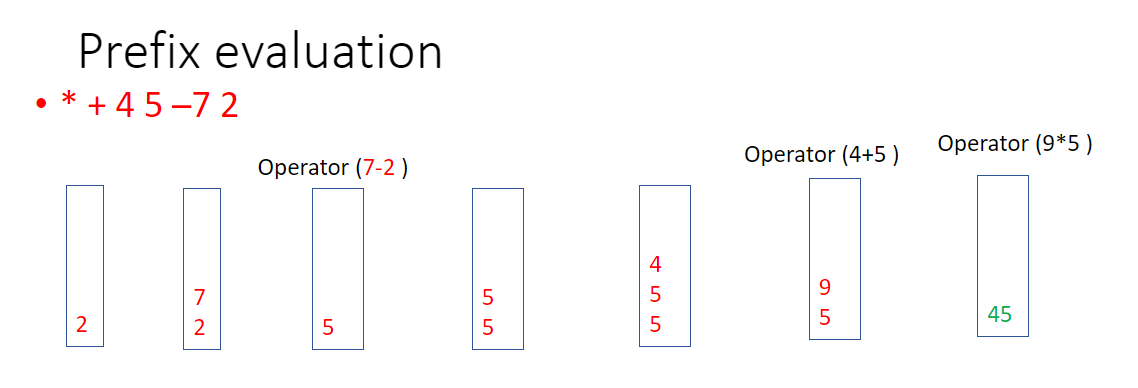
Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack, return it

Step 6: End



**Conclusion:**

Thus we implemented infix to postfix and prefix using stack successfully

## **FAQ:**

1. What is data structure?
2. Types of data structure?
3. Examples of linear data structure & Non-linear data structure?
4. What are the operations can implement on stack?
5. Explain recursion using stack.
6. Explain how a string can be reversed using stack.
7. How does a stack similar to list? How it is different?
8. List advantages and disadvantages of postfix and prefix expression over infix expression.