CSE-2321

Data Structure

Part-4

Presented by

Asmaul Hosna Sadika
Adjunct Faculty
Dept of CSE, IIUC

Contents

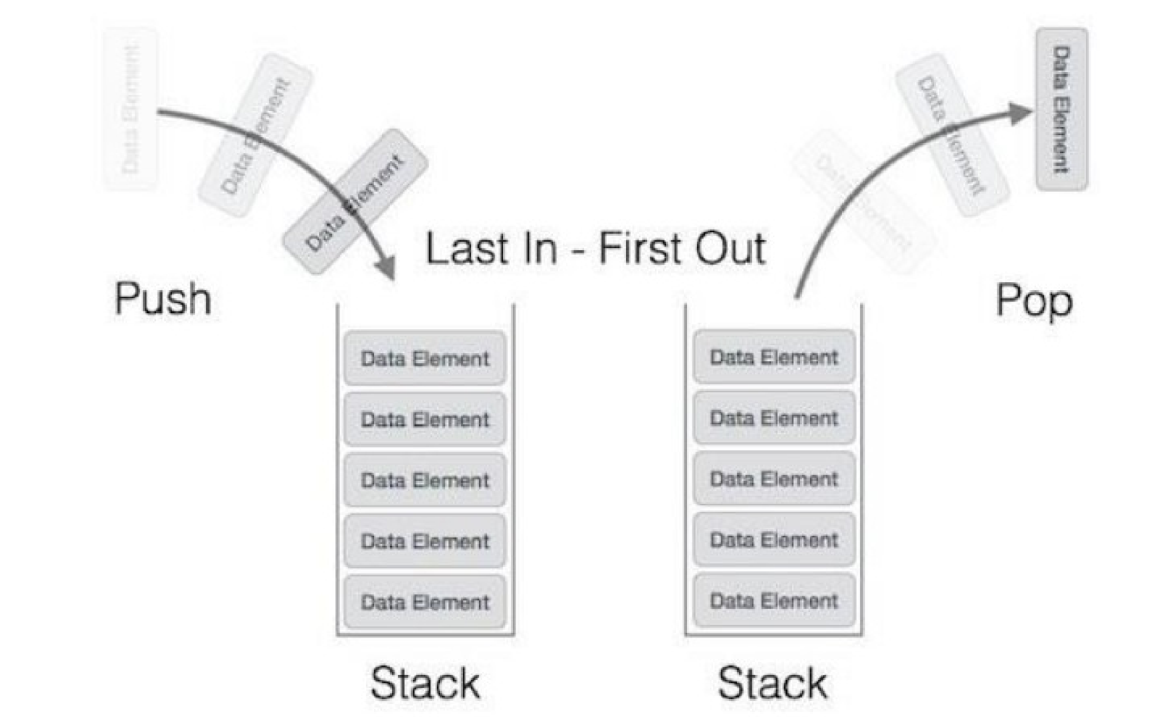
- What is stack?
- Stack Representation
- Basic Operations on Stacks
- Implementation of stack
- Array implementation of Stack
- Linked list implementation of stack
- Application of stack
- Polish notation & Expression parsing

What is stack?

- A stack is a **linear data structure** where elements are stored in the LIFO (Last In First Out) principle where the last element inserted would be the first element to be deleted. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages.
- It is named stack because it has similar operations as the real-world stacks, for example − a pack of cards or a pile of plates, etc.
- Stack is considered a complex data structure because it uses other data structures for implementation, such as Arrays, Linked lists, etc.

Stack Representation

- A stack allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- The following diagram depicts a stack and its operations −



Stack Representation

- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Here we are going to implement stack by means of

1. A linear array (Fixed size)
2. An one way linked list (Dynamic size)

Summary:

A stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

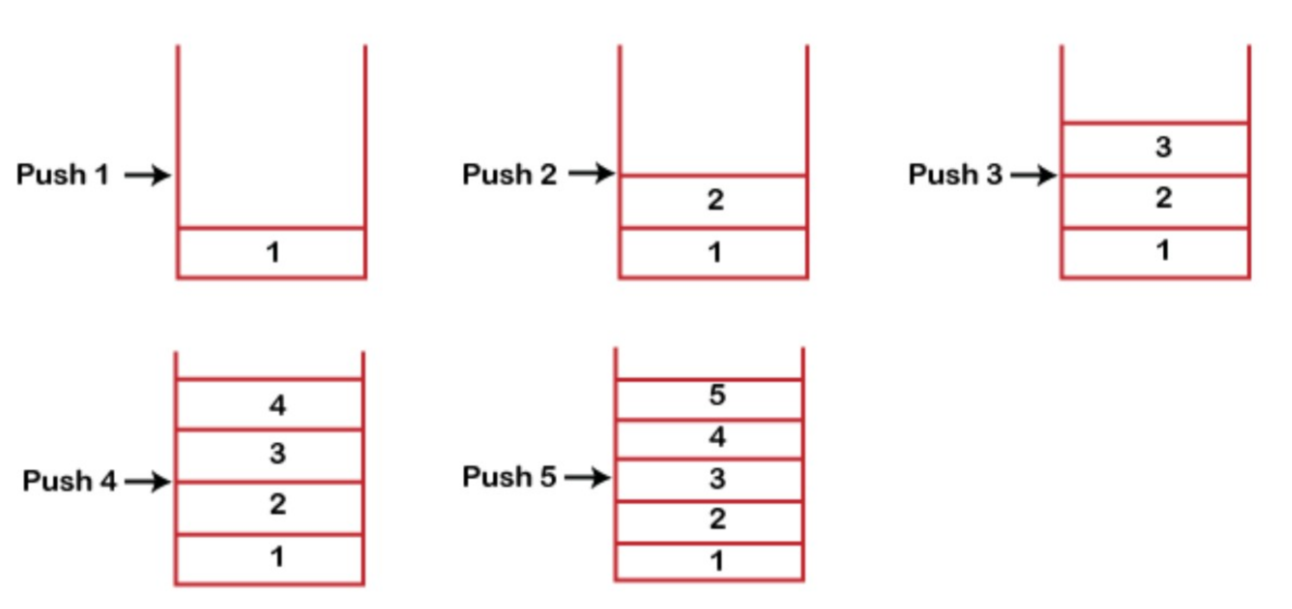
Basic operations of stack

The following are some common operations implemented on the stack:

1. **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
2. **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. **isEmpty():** It determines whether the stack is empty or not.
4. **isFull():** It determines whether the stack is full or not.
5. **peek():** It returns the topmost element of the stack without deleting it.

Working of Stack

- Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown in which we are pushing the elements one by one until the stack becomes full.
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.



Array implementation of stack

Array implementation of stack

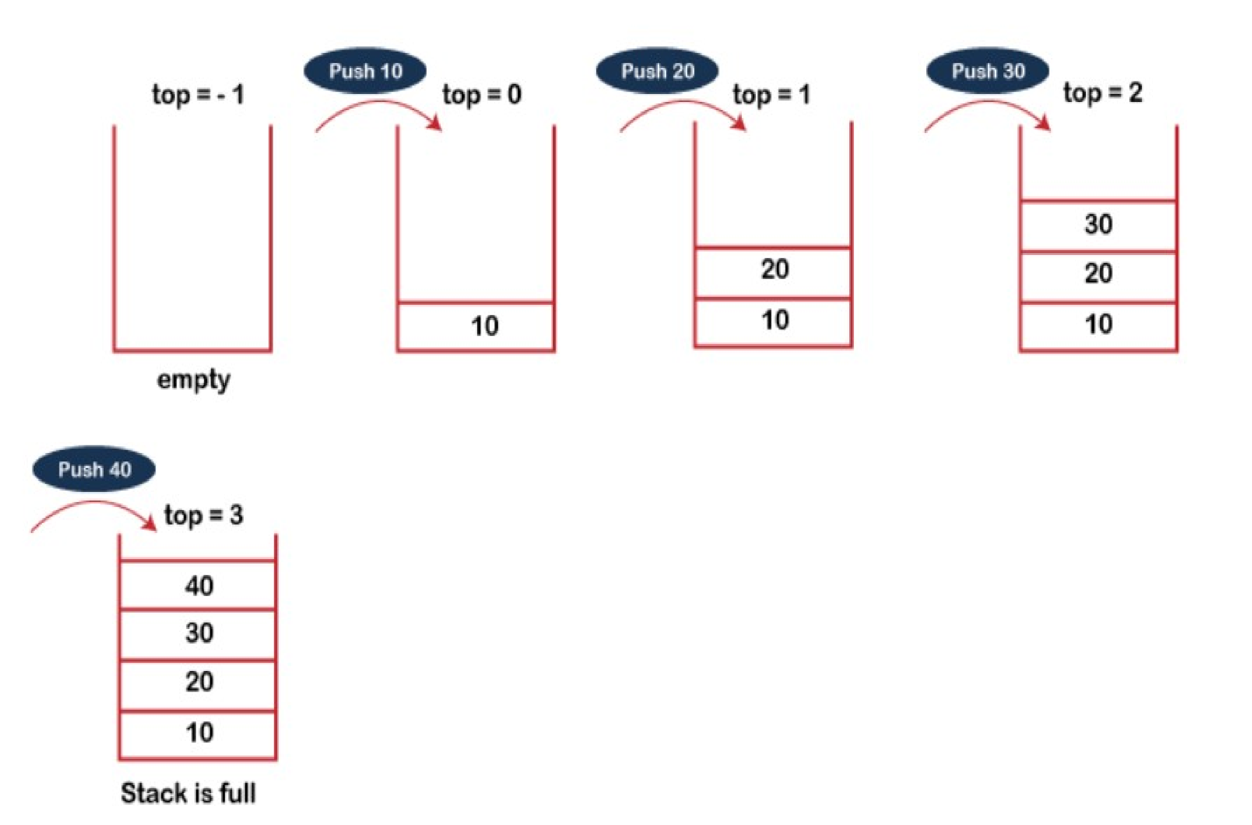
- In array implementation, the stack is formed by using the array.
- All the operations regarding the stack are performed using arrays

1. Visiting each element of the stack (Peek operation)
2. Adding an element onto the stack (push operation)
3. Deletion of an element from a stack (Pop operation)

Push operation

The push() is an operation that inserts elements into the stack.

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



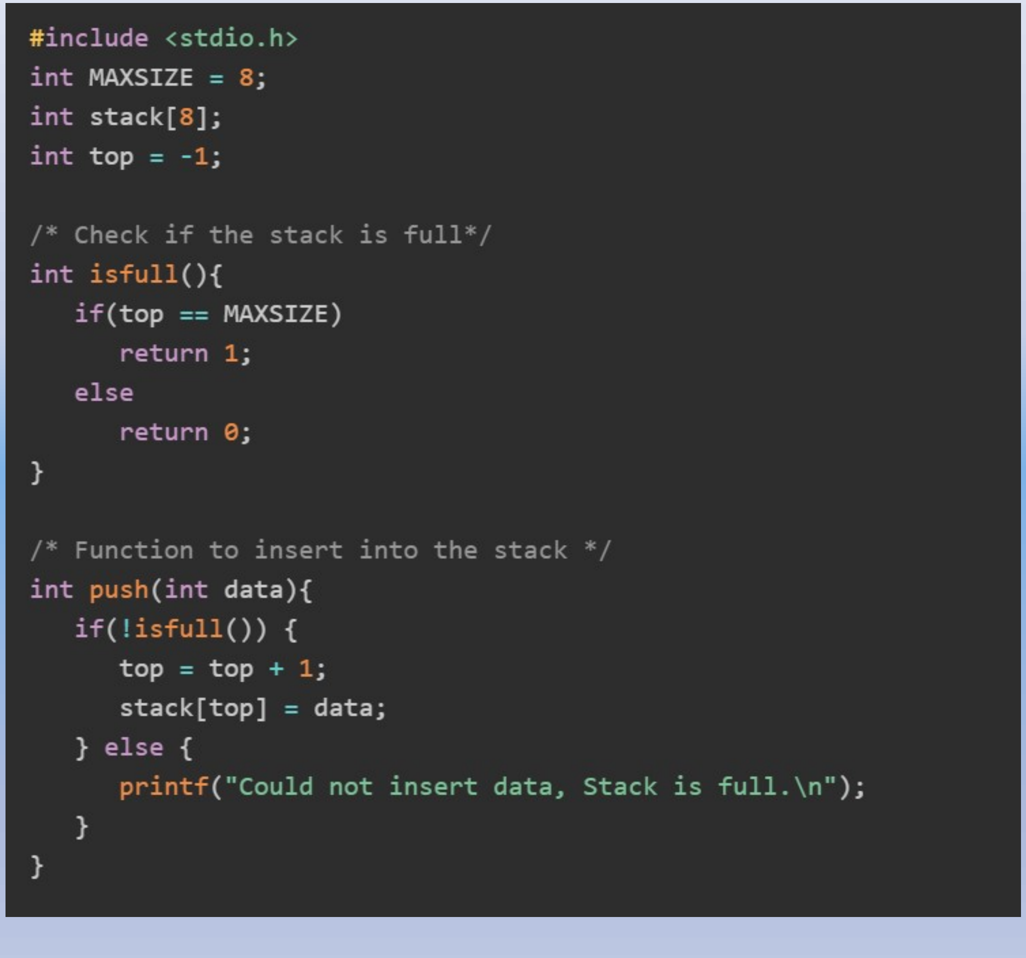
Push algorithm in array

Algorithm

PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an item onto a stack.

1. [Stack already filled]? If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP: = TOP + 1. [Increase TOP by 1].
3. Set STACK [TOP]: = ITEM. [Inserts ITEM in new TOP position].
4. Return

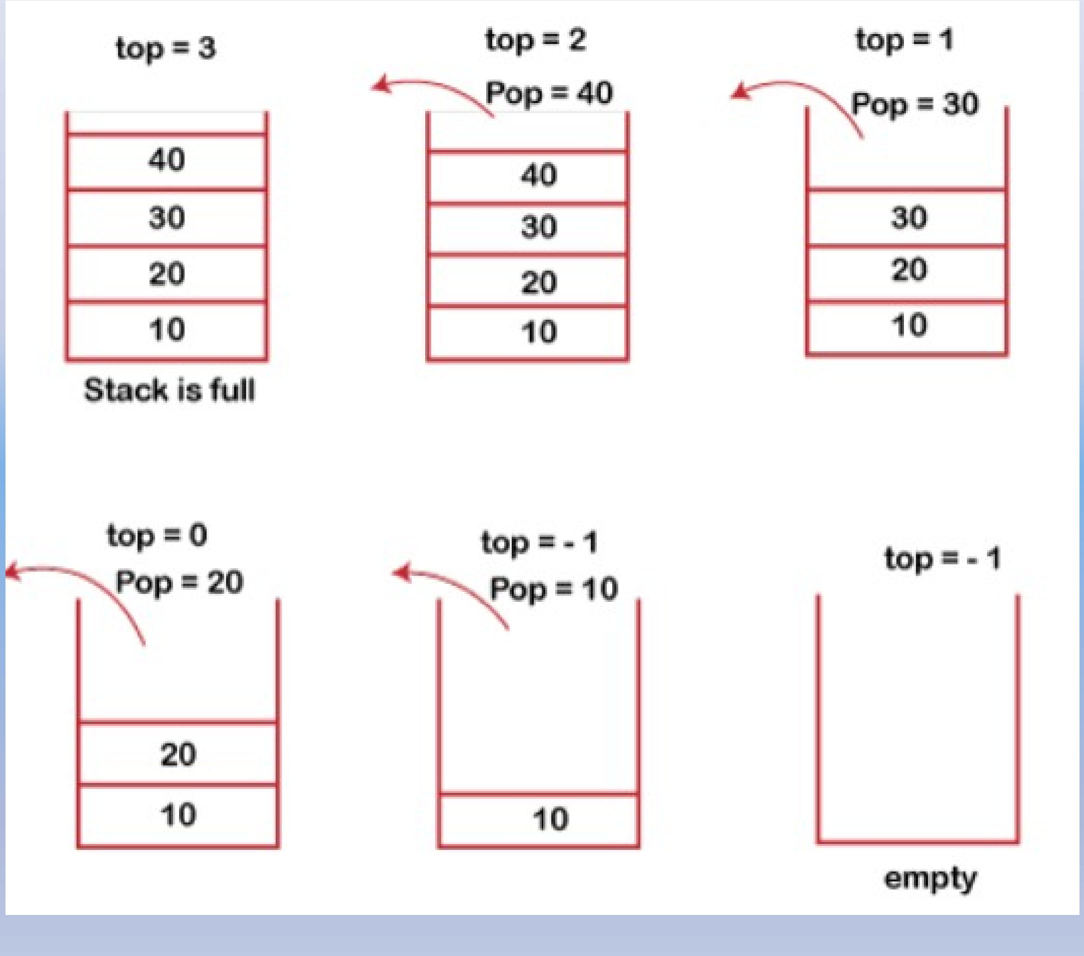


```
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is full*/
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

/* Function to insert into the stack */
int push(int data){
   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



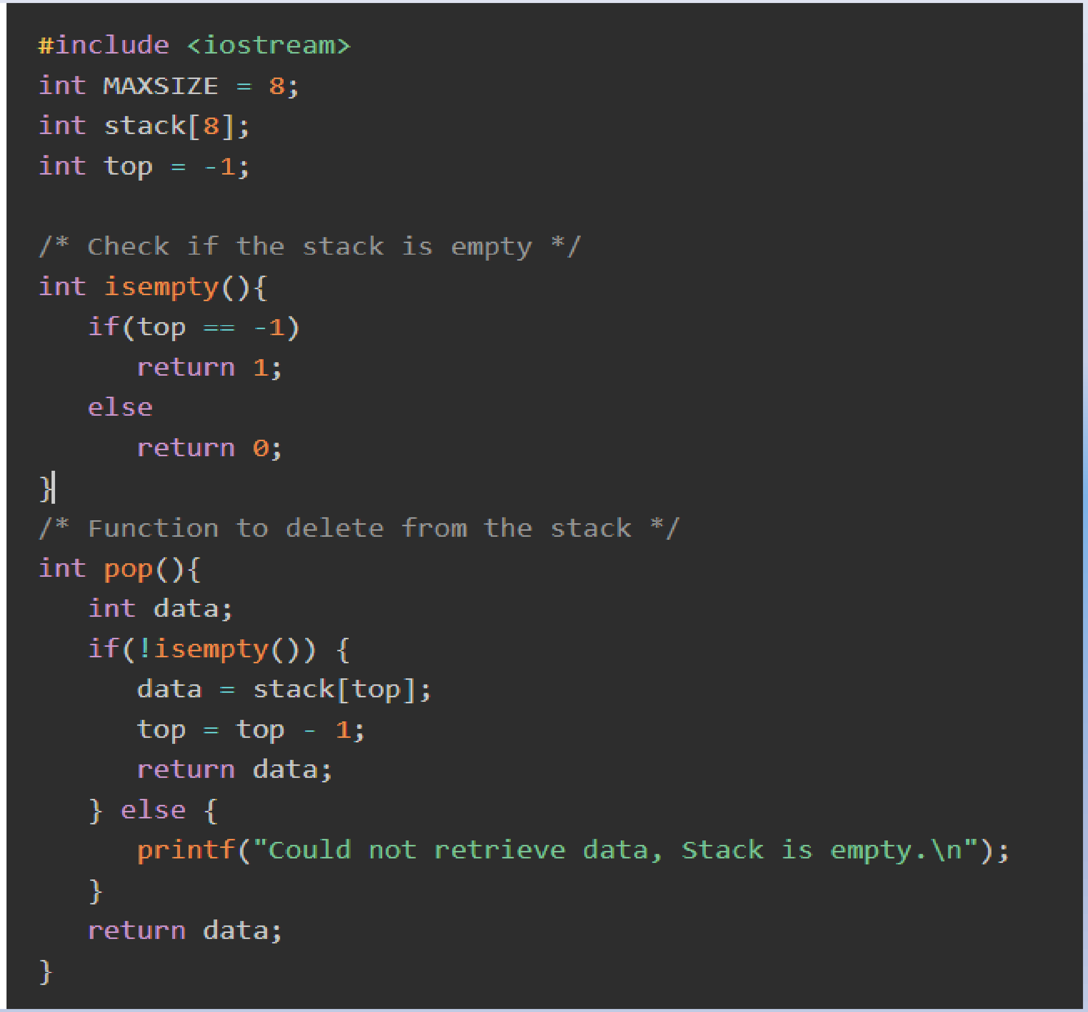
POP operation

Algorithm

POP (STACK, TOP, ITEM)

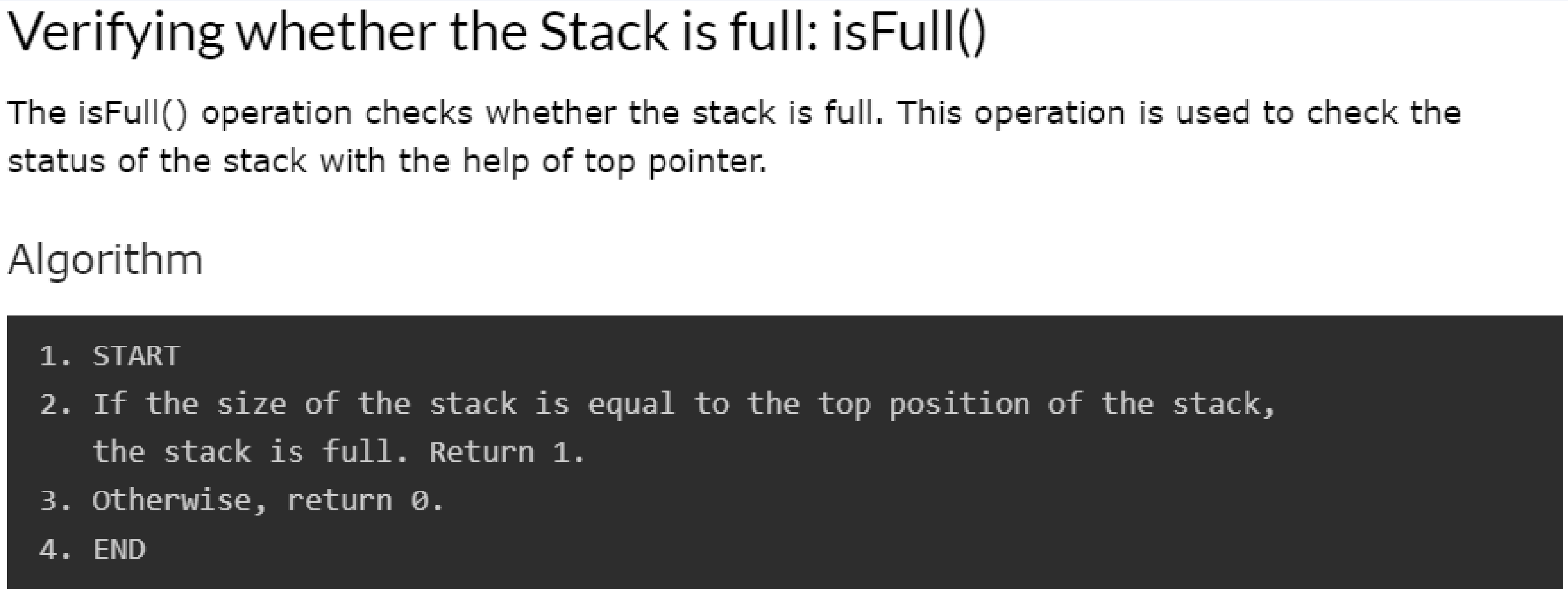
This procedure deletes the TOP element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?] If TOP = 0, then: Print: UNDERFLOW and Return.
2. Set ITEM: =STACK [TOP]. [Assign TOP element to ITEM].
3. Set TOP: = TOP – 1 [Decrease TOP by 1].
4. Return.



```
#include <iostream>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}
/* Function to delete from the stack */
int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
   return data;
}
```

isFull() and isEmpty()

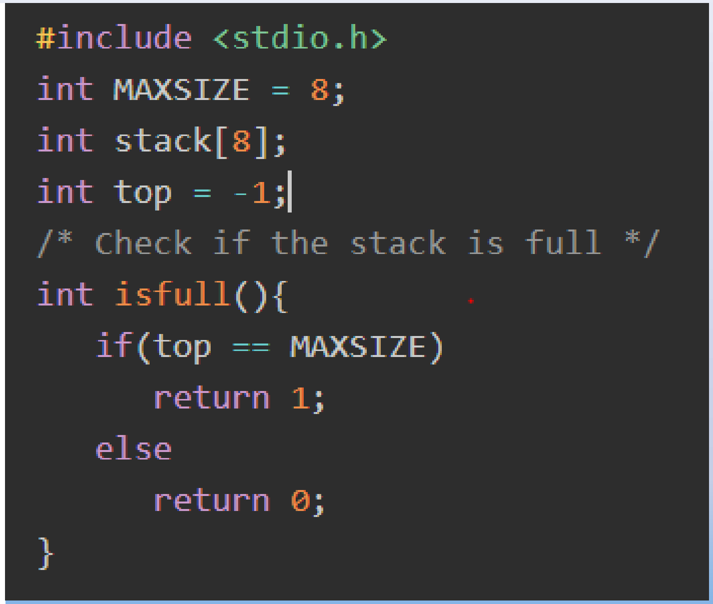


Verifying whether the Stack is full: isFull()

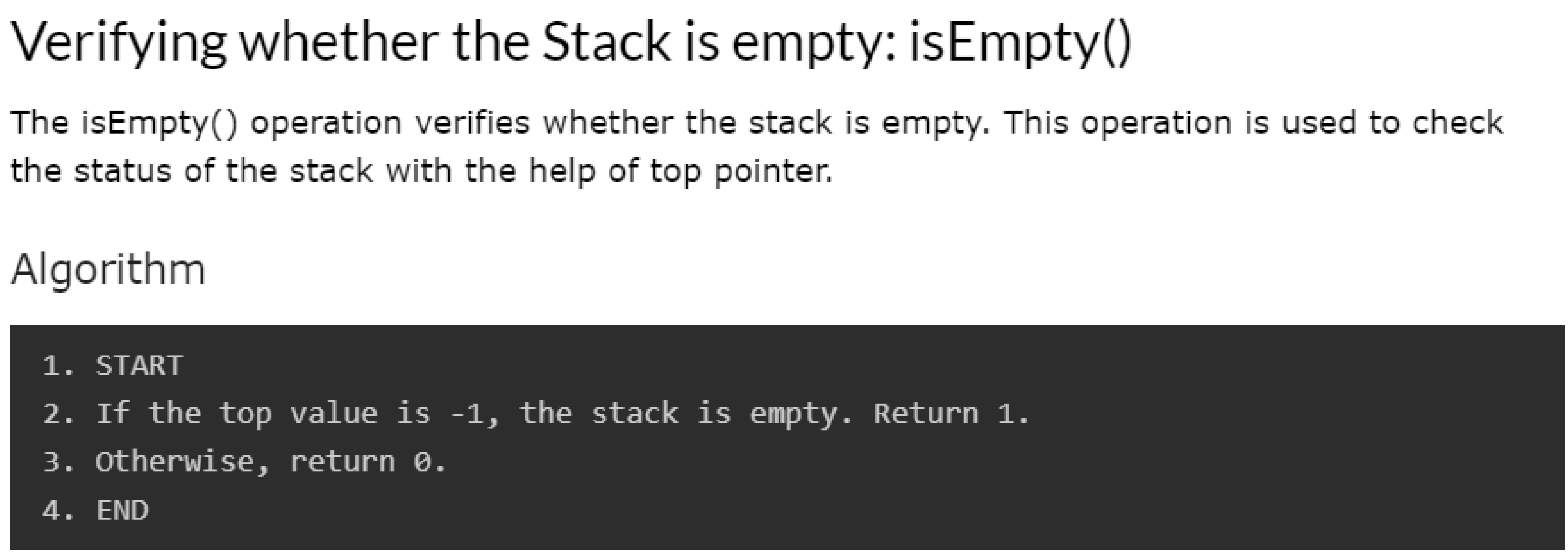
The isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

```
1. START
2. If the size of the stack is equal to the top position of the stack,
   the stack is full. Return 1.
3. Otherwise, return 0.
4. END
```



```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
/* Check if the stack is full */
int isfull(){
   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}
```

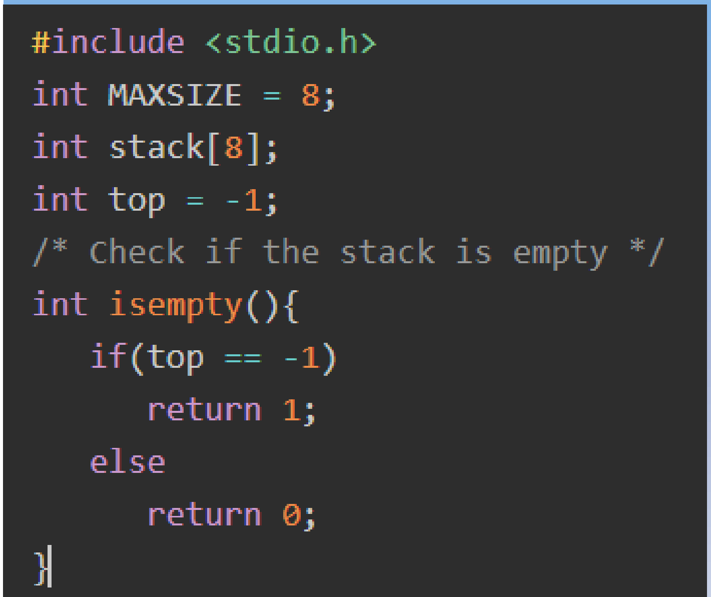


Verifying whether the Stack is empty: isEmpty()

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

```
1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END
```

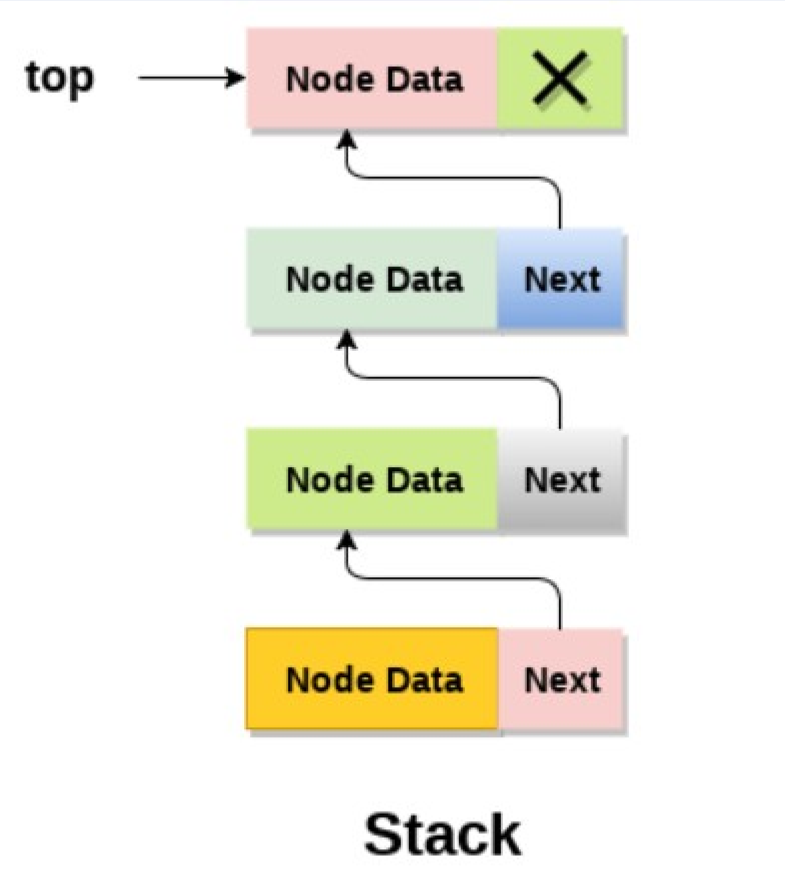


```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;
/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}
```

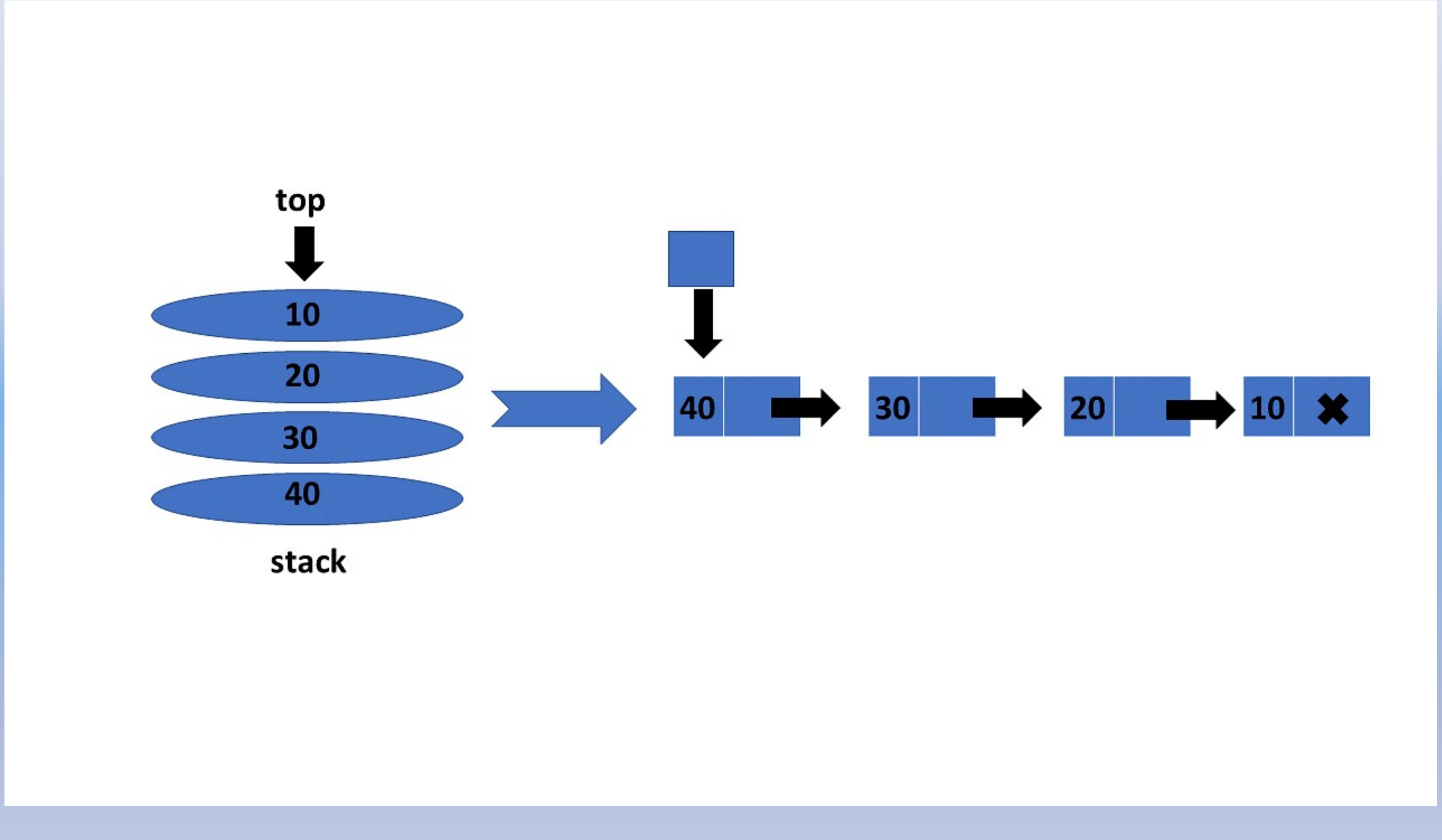
Linked list implementation of Stack

Linked list implementation of Stack

- ❑ Instead of using array, we can also use linked list to implement stack.
- ❑ Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- ❑ In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- ❑ Each node contains a pointer to its immediate successor node in the stack.
- ❑ Stack is said to be overflown if the space left in the memory heap is not enough to create a node.
- ❑ The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.



Push operation



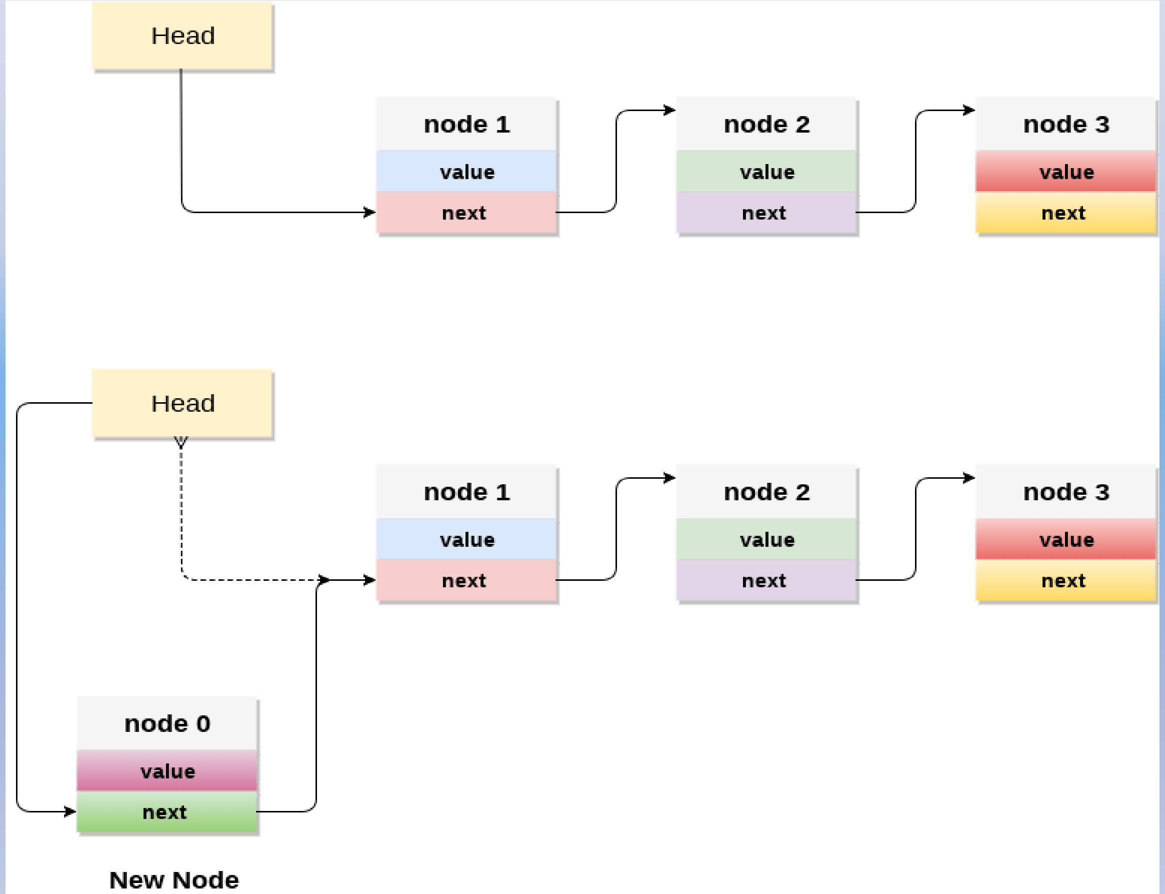
Push operation

Algorithm

PUSH(INFO, LINK, TOP, ITEM, AVAIL)

This algorithm pushes an element to the top of the stack

```
1.  If AVAIL==NULL, then Write: ‘OVERFLOW’ Return
2.  Set NEW=AVAIL and AVAIL = AVAIL->LINK
3.  Set NEW->INFO =ITEM
4.  Set NEW->LINK=TOP
5.  Set TOP=NEW
6.  EXIT
```



POP operation

Algorithm

POP(INFO, LINK, TOP, AVAIL, ITEM)

This algorithm deletes an element from the top of thestack

```
1.  If TOP==NULL , then:Write: ‘UNDERFLOLW’ AND Exit
2.  Set ITEM = TOP->INFO
3.  Set TEMP = TOP AND TOP= TOP->LINK
4.  TEMP->LINK = AVAIL AND AVAIL = TEMP
5.  EXIT
```

Self study: Example 6.1, 6.2, 6.4

Application of stack

Evaluation of arithmetic expression

For most common arithmetic operations, operator symbol is placed between its operands. This is called **infix notation** of an expression. To use stack to evaluate an arithmetic expression, we have to convert the expression into its prefix or postfix notation.

Polish notation

Refers to the notation in which operator symbol is placed before its two operands. This is also called **prefix notation** of an arithmetic expression. The fundamental property of polish notation is that the order in which operations are to be performed is completely determined by positions of operators and operands in expression. Accordingly, one never needs parentheses when writing expression in polish notation.

Reverse Polish notation

Refers to notation in which operator is placed after its two operands. This notation is frequently called **postfix notation**.

Notations

Infix Notation

We write expression in **infix** notation,

a - b + c

where operators are used **in**-between operands.

It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example,

+ - a b c

Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example,

a b – c +

Infix	Prefix	Postfix
a + b	+ a b	a b +
(a + b) ∗ c	∗ + a b c	a b + c ∗
a ∗ (b + c)	∗ a + b c	a b c + ∗
a / b + c / d	+ / a b / c d	a b / c d / +
(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing expressions

In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**.

Operator precedence

Operator precedence means the precedence of any operator over another operator

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Parsing expressions

Associativity

When the operators with the same precedence exist in the expression, we see the associativity of operators. For example, in the expression, i.e., A + B - C, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as (A + B) - C.

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

Conversion of Infix to Prefix using Stack

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.
- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the **TOP** of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a **TOP** of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the **TOP** of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of lower precedence or same precedence.

- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is **')'**, then push it into the stack.
- If the operator is **'('**, then pop all the operators from the stack till it finds the **')'** bracket in the stack.
- If the top of the stack is **')'**, push the operator on the stack.
- In the end, reverse the output. And print it.

Example:

K + L - M * N + (O^P) * W/U/V * T + Q

The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++

Infix to postfix

Rules for the conversion from infix to postfix expression

- Print the operand as they arrive.
- If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
- If the incoming symbol is '(', push it on to the stack.
- If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
- If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
- If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
- If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
- At the end of the expression, pop and print all the operators of the stack.

Algorithm

POSTFIX (Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P

Step 1: Push “(“ onto the STACK and add “)” to the end of Q

Step 2: Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty

Step 3: If an operand is encountered, add it to P

Step 4: If left parentheses is encountered, add it to STACK

Step 5: If an operator **op** is encountered, then:

(a) Repeatedly pop from STACK and add to P each operator which has same precedence or higher precedence than **op**

(b) Add **op** to STACK

Step 6: If a right parenthesis is encountered, then:

(a) Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered

(b) Remove the left parentheses [End of Step 2 Loop]

Step 7: Exit

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

Homework

Infix to prefix/Postfix

1. ((A – (B + C)) * D) ↑ (E + F)
2. (A+(((B*C)-(D/E^F))*G))*H)
3. A ^ B * C - D + E / F /(G + H)
4. (A + B) * C – (D - E)) ^ (F + G)
5. A + B / C + D * (E – F) ^ G