Sairam INSTITUTIONS

Sri SAI RAM ENGINEERING COLLEGE
INSTITUTE OF TECHNOLOGY
West Tambaram, Chennai - 44

YEAR	SEM
II	III

CS8391

DATA STRUCTURES
(Common to CSE & IT)

UNIT NO 2

LINEAR DATA STRUCTURES - STACKS, QUEUES

2.2 APPLICATIONS OF STACK

2.2.1 EVALUATING POSTFIX EXPRESSIONS & CONVERSION OF INFIX TO POSTFIX EXPRESSIONS

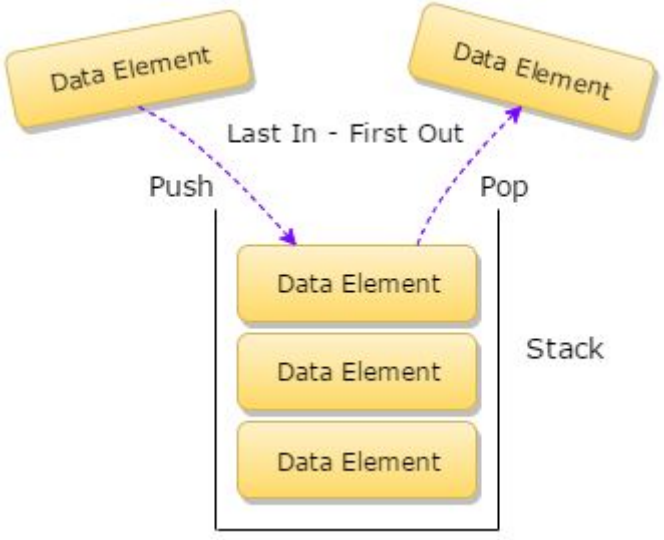
Version: 1.XX

4 QUALITY EDUCATION

NBA NAAC (A+) nirf

Applications of Stack

Stack is an abstract data type and a data structure that follows LIFO (last in first out) strategy. It means the element added last will be removed first. Stack allows two operations push and pop. Push adds an element at the top of the stack and pop removes an element from top of the stack.



Applications of Stack

1. **Expression Evaluation**: Stack is used to evaluate prefix, postfix and infix expressions.
2. **Expression Conversion**: An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
3. **Syntax Parsing**: Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
4. **Backtracking**: Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.
5. **Parenthesis Checking**: Stack is used to check the proper opening and closing of parenthesis.

6. **String Reversal**: Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.
7. **Function Call**: Stack is used to keep information about the active functions or subroutines.

INFIX EXPRESSIONS

If an operator is preceded and succeeded by an operand then such an expression is termed infix expression.

It follows the scheme of <operand><operator><operand>

E.g., A+B.

POSTFIX EXPRESSIONS

If an operator is succeeded by both the operand then such an expression is termed postfix expression.

It follows the scheme of <operand><operand><operator>

E.g., AB+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence). Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

ALGORITHM TO CONVERT INFIX TO POSTFIX EXPRESSIONS

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.

5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.

    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.

    [End of If]

    [End of If]
7. END.

EXAMPLE 1

	RPN	Stack	Input Expression
1			A+(B*(C-D)/E)
2	A		+(B*(C-D)/E)
3	A	+	(B*(C-D)/E)
4	A	(+	B*(C-D)/E)
5	AB	(+	*(C-D)/E)
6	AB	* (+	(C-D)/E)
7	AB	(* (+	C-D)/E)
8	ABC	(* (+	-D)/E)
9	ABC	(* (+	D)/E)
10	ABCD	(* (+	)/E)
11	ABCD-	* (+	/E)
12	ABCD-*	/ (+	E)
13	ABCD-*E	/ (+	)
14	ABCD-*E/	+	
15	ABCD-*E/+		

EXAMPLE 2

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: **ABC*DEF^/G*-H*+**

ALGORITHM TO EVALUATE POSTFIX EXPRESSION

1. Read the postfix expression from left to right.
2. If an operand is encountered then push the element in the stack.
3. If an operator is encountered then pop the two operands from the stack and then evaluate it.
4. Push back the result of the evaluation onto the stack.
5. Repeat it till the end of the expression.

EXAMPLE

Expression: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)