International Islamic University Chittagong
Department of Computer Science & Engineering
Autumn - 2022

Course Code: CSE-2321
Course Title: Data Structures

Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

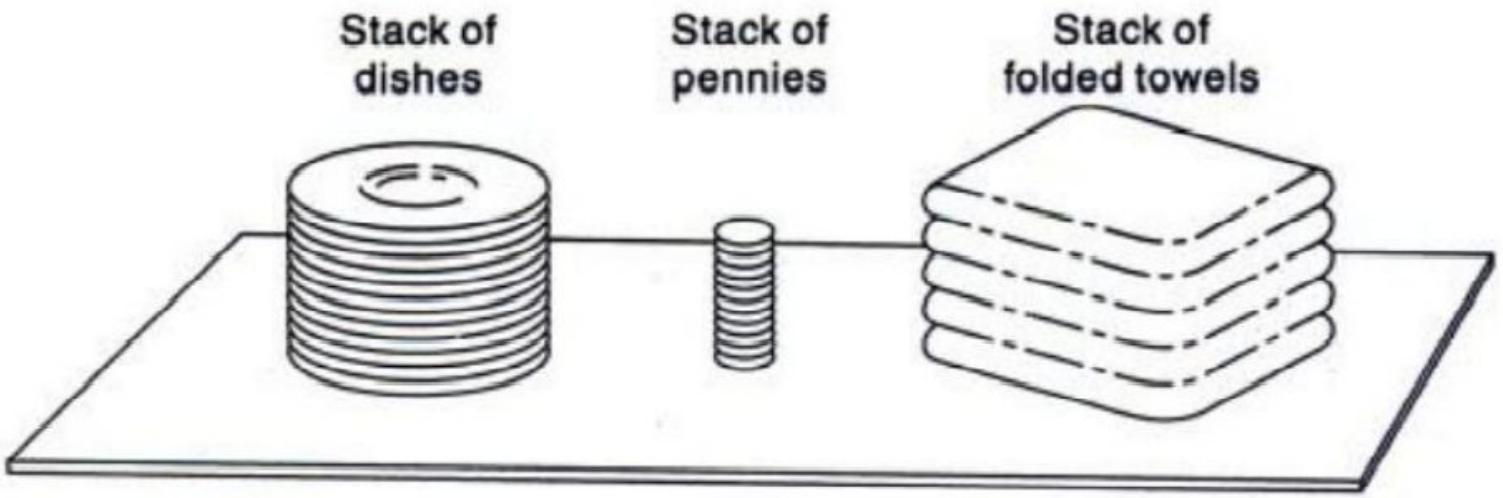
Lecture – 7

Stacks

Stacks

A **stack** is a list of elements in which an element may be inserted or deleted only at one end, called the ***top*** of the stack.

- Stacks are also called ***last-in first-out*** **(LIFO)** lists.



Example 6.1

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Figure 6.3 shows three ways of picturing such a stack. For notational convenience, we will frequently designate the stack by writing:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

Stacks

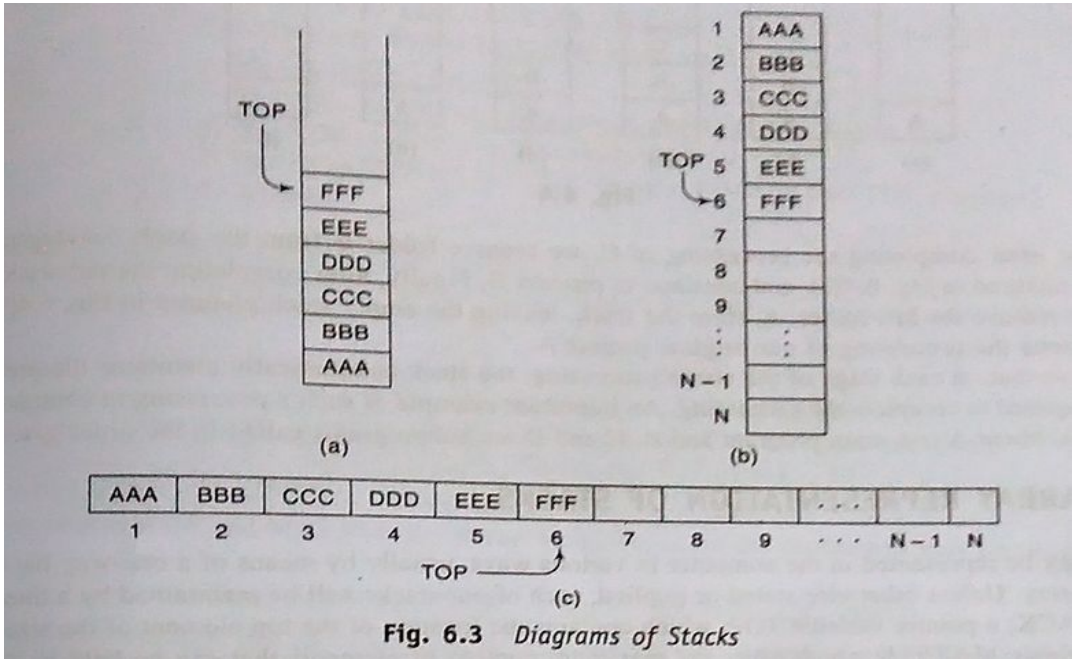


Fig. 6.3 *Diagrams of Stacks*

- There are two basic operations are associated with stacks:
  - PUSH - to insert an element into a stack
  - POP - to delete an element from a stack

- Stacks may be represented in the computer in various ways, usually by means of

  i) a Linear Array or

  ii) a one-way Linked List

Array Representation of Stacks

Stack can be represented by a linear array **STACK**, a pointer variable **TOP**, which contains the location of the top elements of the stack; and a variable **MAXSTK** which gives the maximum number of elements that can be held by the stack. The condition, TOP = 0 will indicate that the stack is empty.

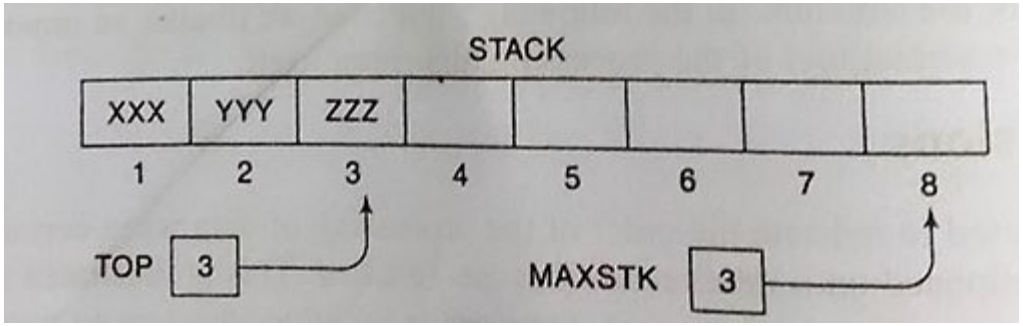


Fig shows such an array representation of a stack. Since TOP = 3, the attacks has three elements XXX, YYY and ZZZ. And since MAXSTK = 8, we can add 5 more items in the stack.

Array Representation of Stacks

Algorithm 6.1: PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an item on to 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

Algorithm 6.2: POP (STACK, TOP, ITEM)

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

- [Stack has an item to be removed] If TOP = 0, then: Print: UNDERFLOW and Return.

1. Set ITEM: =STACK [TOP]. [Assign TOP element to ITEM].
2. Set TOP: = TOP – 1 [Decrease TOP by 1].
3. Return.

Array Representation of Stacks

6.1 Consider the following stack of characters, where STACK is allocated N = 8 memory cells:

STACK: A, C, D, F, K, ___, ___, ___,

(For notational convenience, we use "___" to denote an empty memory cell.) Describe the stack as the following operations take place:

(a) POP(STACK, ITEM)
(b) POP(STACK, ITEM)
(c) PUSH(STACK, L)
(d) PUSH(STACK, P)
(e) POP(STACK, ITEM)
(f) PUSH(STACK, R)
(g) PUSH(STACK, S)
(h) POP(STACK, ITEM)

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly:

(a) STACK: A, C, D, F, ___, ___, ___, ___
(b) STACK: A, C, D, ___, ___, ___, ___, ___
(c) STACK: A, C, D, L, ___, ___, ___, ___
(d) STACK: A, C, D, L, P, ___, ___, ___
(e) STACK: A, C, D, L, ___, ___, ___, ___
(f) STACK: A, C, D, L, R, ___, ___, ___
(g) STACK: A, C, D, L, R, S, ___, ___
(h) STACK: A, C, D, L, R, ___, ___, ___

Minimizing Overflow

Minimizing Overflow

There is an essential difference between underflow and overflow in dealing with stacks. Underflow depends exclusively upon the given algorithm and the given input data, and hence there is no direct control by the programmer. Overflow, on the other hand, depends upon the arbitrary choice of the programmer for the amount of memory space reserved for each stack, and this choice does influence the number of times overflow may occur.

Generally speaking, the number of elements in a stack fluctuates as elements are added to or removed from a stack. Accordingly, the particular choice of the amount of memory for a given stack involves a time-space tradeoff. Specifically, initially reserving a great deal of space for each stack will decrease the number of times overflow may occur; however, this may be an expensive use of the space if most of the space is seldom used. On the other hand, reserving a small amount of space for each stack may increase the number of times overflow occurs; and the time required for resolving an overflow, such as by adding space to the stack, may be more expensive than the space saved.

Various techniques have been developed which modify the array representation of stacks so that the amount of space reserved for more than one stack may be more efficiently used. Most of these techniques lie beyond the scope of this text. We do illustrate one such technique in the following example.

Example 6.3

Suppose a given algorithm requires two stacks, A and B. One can define an array STACKA with n_1 elements for stack A and an array STACKB with n_2 elements for stack B. Overflow will occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose instead that we define a single array STACK with $n = n_1 + n_2$ elements for stacks A and B together. As pictured in Fig. 6.6, we define STACK[1] as the bottom of stack A and let A "grow" to the right, and we define STACK[n] as the bottom of stack B and let B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. This technique will usually decrease the number of times overflow occurs even though we have not increased the total amount of space reserved for the two stacks. In using this data structure, the operations of PUSH and POP will need to be modified.

Polish Notation

Example 6.5

Suppose we want to evaluate the following parenthesis-free arithmetic expression:

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

First we evaluate the exponentiations to obtain

$$8 + 5 * 4 - 12 / 6$$

Then we evaluate the multiplication and division to obtain $8 + 20 - 2$. Last, we evaluate the addition and subtraction to obtain the final result, 26. Observe that the expression is traversed three times, each time corresponding to a level of precedence of the operations.

Polish Notation

For most common arithmetic operations, the operator symbol is placed between its two operands. For example,

$$A + B \qquad C - D \qquad E * F \qquad G/H$$

This is called *infix notation*. With this notation, we must distinguish between

$$(A + B) * C \qquad \text{and} \qquad A + (B * C)$$

by using either parentheses or some operator-precedence convention such as the usual precedence levels discussed above. Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands. For example,

$$+AB \qquad -CD \qquad *EF \qquad /GH$$

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

$$(A + B) * C = [+AB] * C = *+ ABC$$
$$A + (B * C) = A + [*BC) = + A*BC$$
$$(A + B)/(C - D) = [+AB]/[-CD] = / + AB - CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands:

$$AB+ \quad CD- \quad EF* \quad GH/$$

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called *postfix* (or *suffix*) *notation*, whereas *prefix notation* is the term used for Polish notation, discussed in the preceding paragraph.

Evaluation of a Postfix notation

Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm 6.5: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate $B \otimes A$.
   (c) Place the result of (b) back on STACK.

   [End of If structure.]

   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

Evaluation of a Postfix notation

Example 6.6

Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, −

(Commas are used to separate the elements of P so that 5, 6, 2 is not interpreted as the number 562.) The equivalent infix expression Q follows:

Q: 5 * (6 + 2) − 12 / 4

Note that parentheses are necessary for the infix expression Q but not for the postfix expression P.

We evaluate P by simulating Algorithm 6.5. First we add a sentinel right parenthesis at the end of P to obtain

P:	5,	6,	2,	+,	*,	12,	4,	/,	−,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

The elements of P have been labeled from left to right for easy reference. Figure 6.11 shows the contents of STACK as each element of P is scanned. The final number in STACK, 37, which is assigned to VALUE when the sentinel ")" is scanned, is the value of P.

Symbol Scanned		STACK
		5
(1)	5	5, 6
(2)	6	5, 6, 2
(3)	2	5, 8
(4)	+	40
(5)	*	40, 12
(6)	12	40, 12, 4
(7)	4	40, 3
(8)	/	37
(9)	−	
(10)	)	

Transforming Infix Expressions into Postfix Expressions

Algorithm 6.6: POLISH(Q, P)

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

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (b) Add ⊗ to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]

[End of Step 2 loop.]

7. Exit.

Transforming Infix Expressions into Postfix Expressions

Example 6.7

Consider the following arithmetic infix expression Q:

Q: A + (B * C – (D / E ↑ F) * G) * H

We simulate Algorithm 6.6 to transform Q into its equivalent postfix expression P. First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

Q:	A	+	(	B	*	C	–	(	D	/	E	↑	F	)	*
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
											G	)	*	H	)
											(16)	(17)	(18)	(19)	(20)

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	–	(+ (–	A B C *
(8)	(	(+ (– (	A B C *
(9)	D	(+ (– (	A B C * D
(10)	/	(+ (– (/	A B C * D
(11)	E	(+ (– (/	A B C * D E
(12)	↑	(+ (– (/ ↑	A B C * D E
(13)	F	(+ (– (/ ↑	A B C * D E F
(14)	)	(+ (–	A B C * D E F ↑ /
(15)	*	(+ (– *	A B C * D E F ↑ /
(16)	G	(+ (– *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * –
(18)	*	(+ *	A B C * D E F ↑ / G * –
(19)	H	(+ *	A B C * D E F ↑ / G * – H
(20)	)		A B C * D E F ↑ / G * – H * +