Data Structure

تركيب بيانات
الفرقة الثالثة علوم إحصاء وعلوم الحاسب

By
Dr. Reda Elbarougy
د/ رضا الباروجى
Lecturer of computer sciences
In Mathematics Department
Faculty of Science
Damietta University

رقم المحاضرة

رقم المحاضرة	التاريخ	ملاحظات
المحاضرة 1	2020-02	مقدمة – الفصل الاول
المحاضرة 2	2020-03-03	اساسيات تحليل الخورازميات
المحاضرة 3	2020-03-10	
المحاضرة 4	2020-03-17	
المحاضرة 5	2020-03-24	
المحاضرة 6	2020-03-31	
المحاضرة 7	2020-04-07	
المحاضرة 8		
المحاضرة 9		
المحاضرة 10		
المحاضرة 11		

Chapter 6: Part-I
Stacks, Queues, Recursion

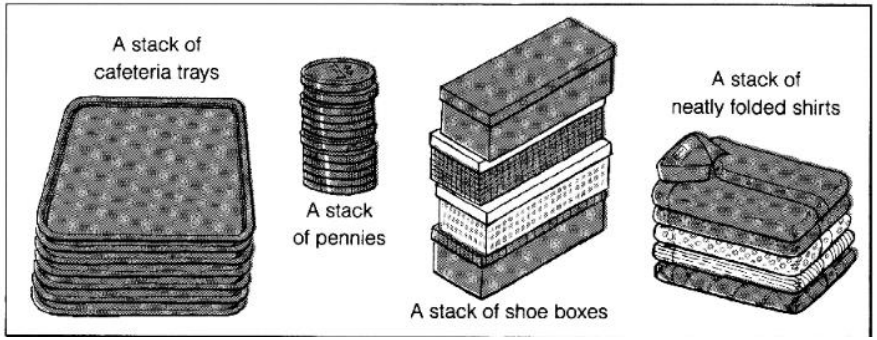
Chapter 6: Stacks

6.1 Introduction

6.1 Introduction

- Linear list and linear array allowed one to insert and delete elements at any place in the list (Beginning, end or middle).

Stack: Last-in first-out (LIFO)



Queue: First-in first-out (FIFO)

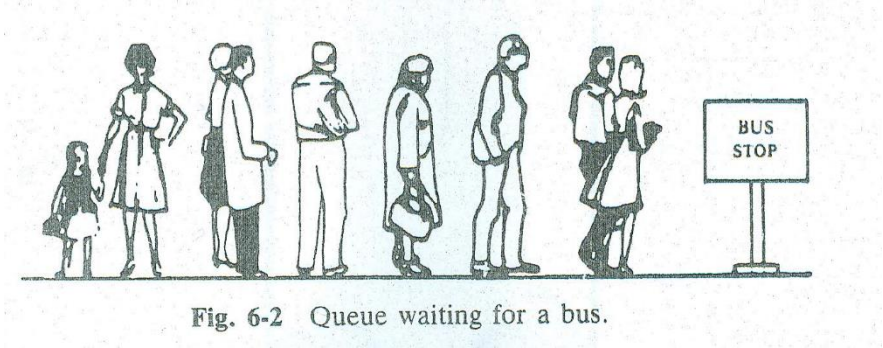


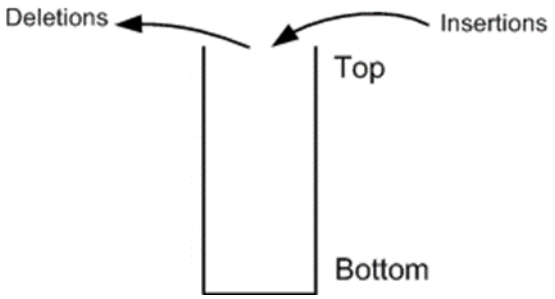
Fig. 6-2 Queue waiting for a bus.

6.1 Introduction

- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.

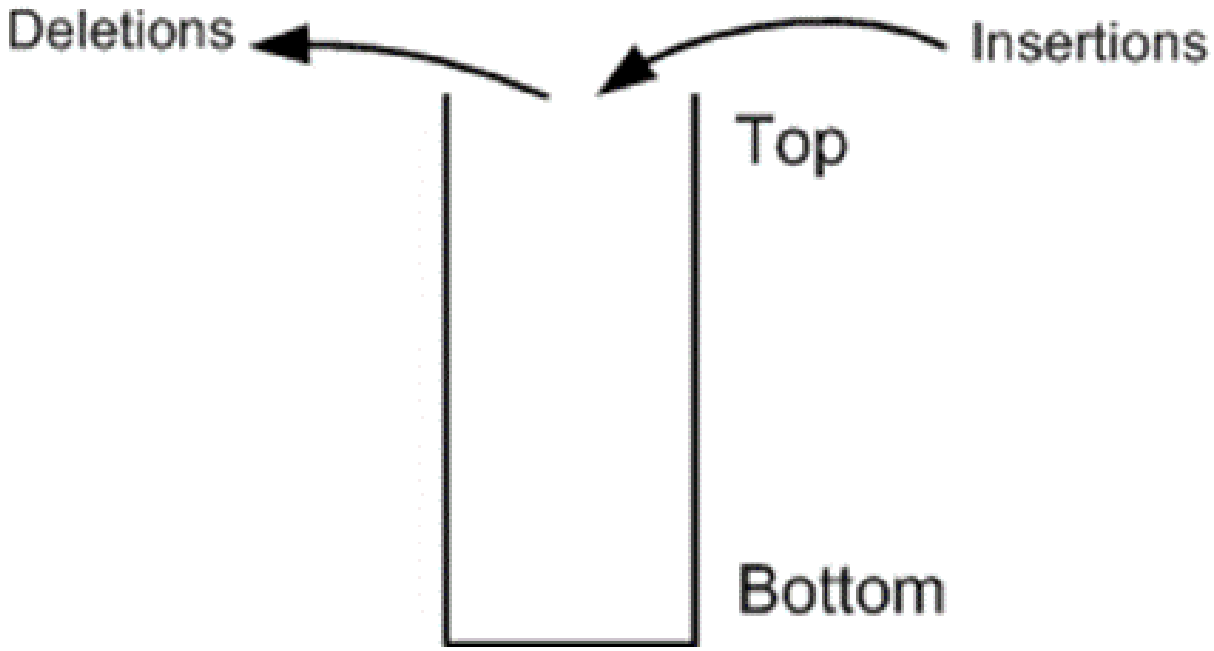
6.2 Stacks

- 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. The other end is called the **bottom**.
- This means, in particular, that elements are removed from a stack in the **reverse** order of that in which they were inserted into the stack.
- Special terminology is used for two basic operations associated with stacks.
- "Push" is the term used to insert an element into a stack.
- "Pop" is the term used to delete an element from a stack.



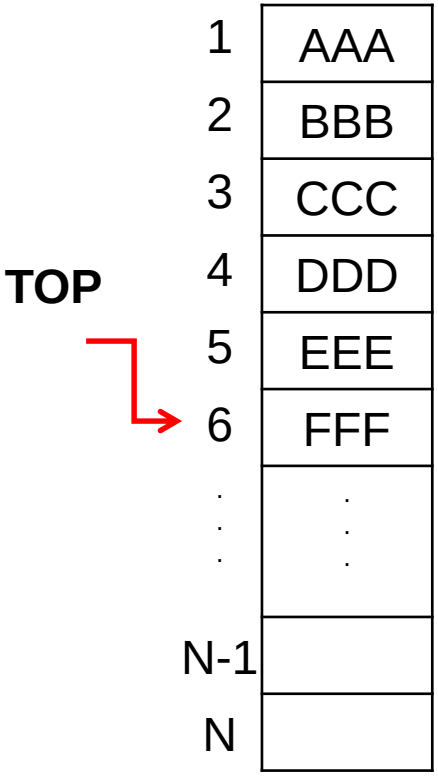
6.2 Stacks

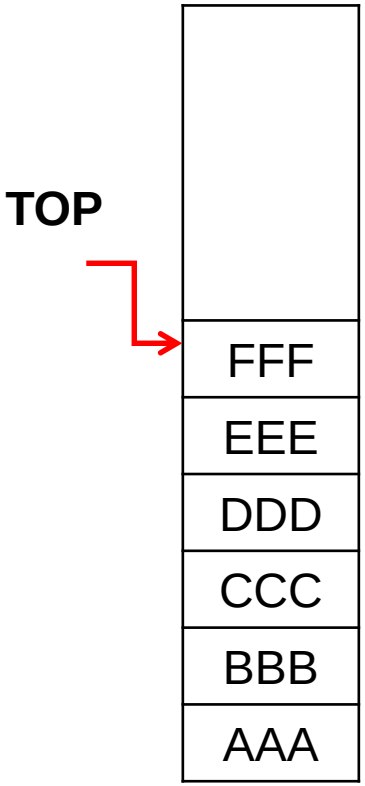
- An item may be added or removed only from the top of a stack. This means, in particular, that the last item to be added to a stack is the first item to be removed. Accordingly, stacks are also called last-in first-out (LIFO) lists.



Example 6.1

- Suppose the following 6 elements are pushed, in order, onto an empty stack: AAA, BBB, CCC, DDD, EEE, FFF





Example 6.1

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

The implication is that the right-most element is the top element. We emphasize that, regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted, and so on. Consequently, the elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

Consider again the AVAIL list of available nodes discussed in Chap. 5. Recall that free nodes were removed only from the beginning of the AVAIL list, and that new available nodes were inserted only at the beginning of the AVAIL list. In other words, the AVAIL list was implemented as a stack. This implementation of the AVAIL list as a stack is only a matter of convenience rather than an inherent part of the structure. In the following subsection we discuss an important situation where the stack is an essential tool of the processing algorithm itself.

Application of a stack

(i) Conversion of infix to postfix form

(ii) Reversing of a line.

(iii) Removal of recursion

(iv) Evaluating post fix expression

application of STACK :- Postponed Decisions

- Stacks are frequently used to indicate the order of the processing of data when certain steps of the processing must be postponed until other conditions are fulfilled.

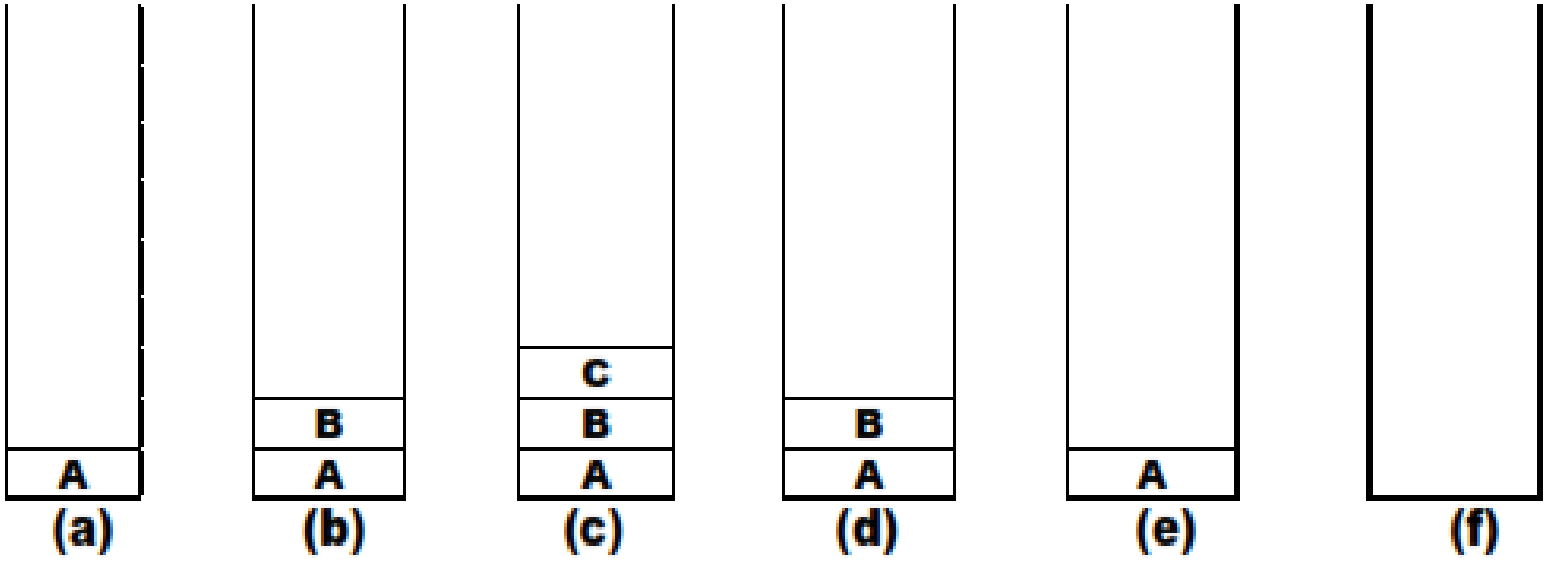


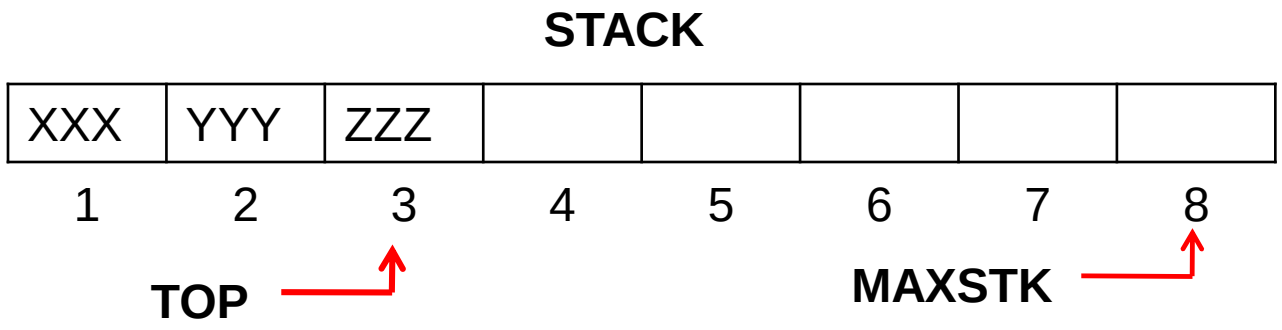
Fig. 6-4

6.3 Array Representation of Stacks

- Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array.
- Our stack is maintained by a linear array STACK: a pointer variable TOP, which contains the location of the top element 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 or TOP = NULL will indicate that the stack is empty.

Array Representation of Stacks

- Figure pictures such an array representation of a stack.
- Since TOP = 3, the stack has three elements, XXX, YYY and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack.



Adding (Pushing) an item onto stack

Procedure 6.1:

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 [Increases TOP by 1]
3. Set STACK[TOP] = ITEM [Inserts ITEM in new TOP position]
4. Return.

Removing (Popping) an item from a stack

Procedure 6.2:

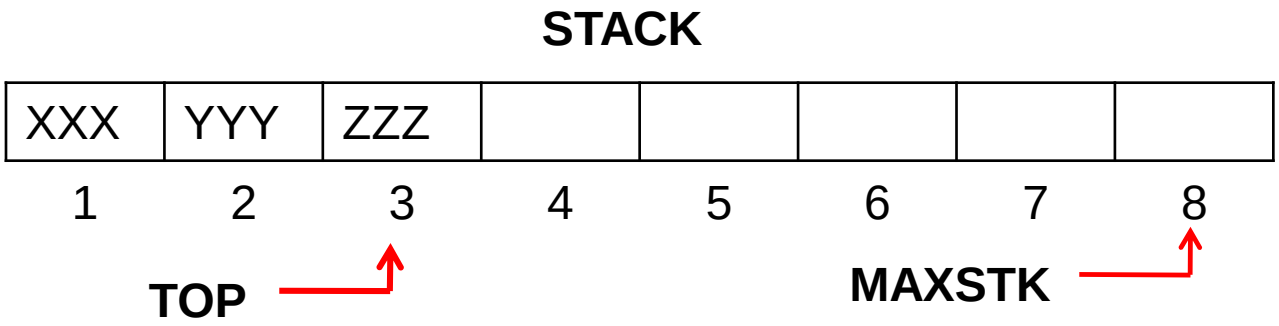
POP(STACK, TOP, ITEM)

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

1. [STACK has no item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM = STACK[TOP] [Assigns TOP element to ITEM]
3. Set TOP = TOP – 1 [Decreases TOP by 1]
4. Return.

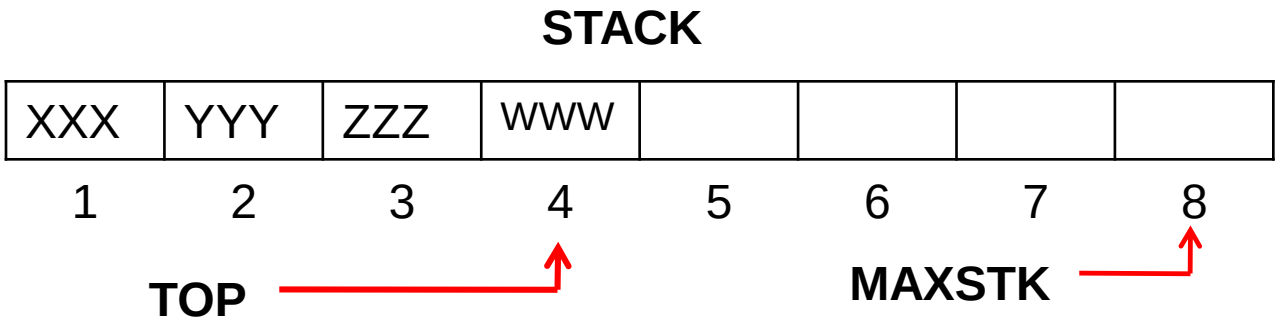
Example 6.2

(a) Consider the following stack. Simulate the operation PUSH(STACK,WWW)



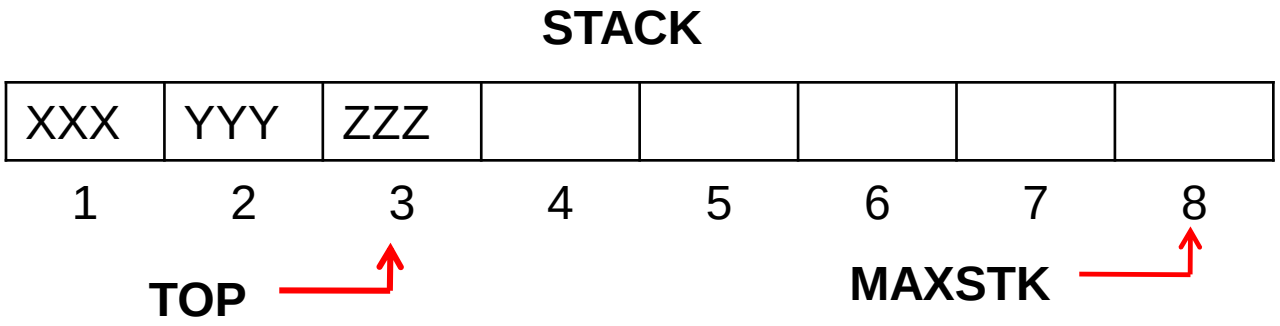
1. Since TOP=3, control is transferred to Step2.
2. TOP=3+1=4.
3. STACK[TOP]=STACK[4]=WWW
4. Return.

Note that WWW is now the top element in the stack.



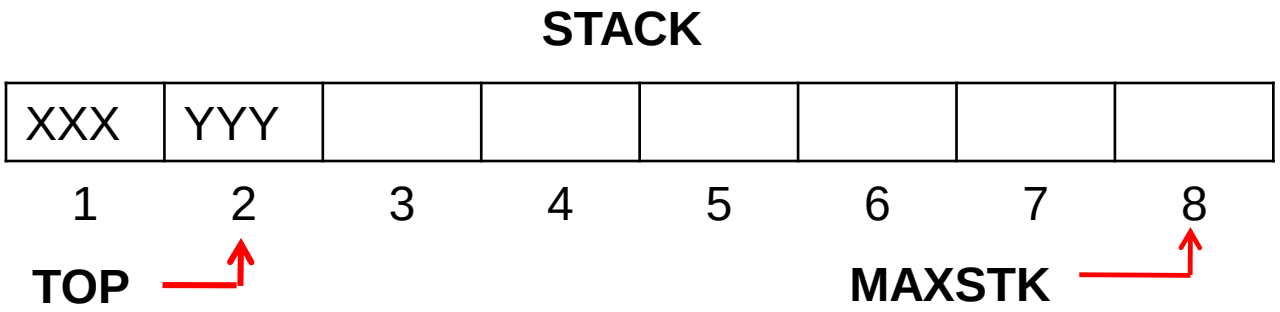
Example 6.2

(b) Consider the following stack. Simulate the operation POP(STACK,ITEM)



1. Since TOP=3, control is transferred to Step2.
2. ITEM=ZZZ
3. TOP=3-1=2.
4. Return.

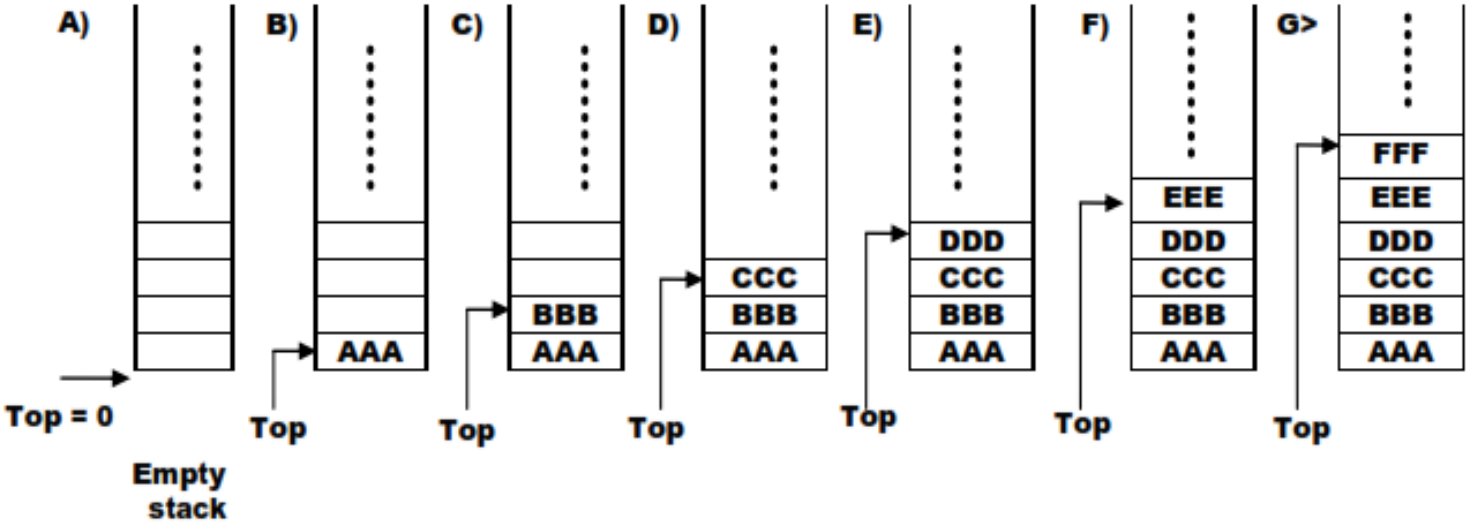
Observe that STACK[TOP]=STACK[2]=YYY is now the top element in the stack.



Example

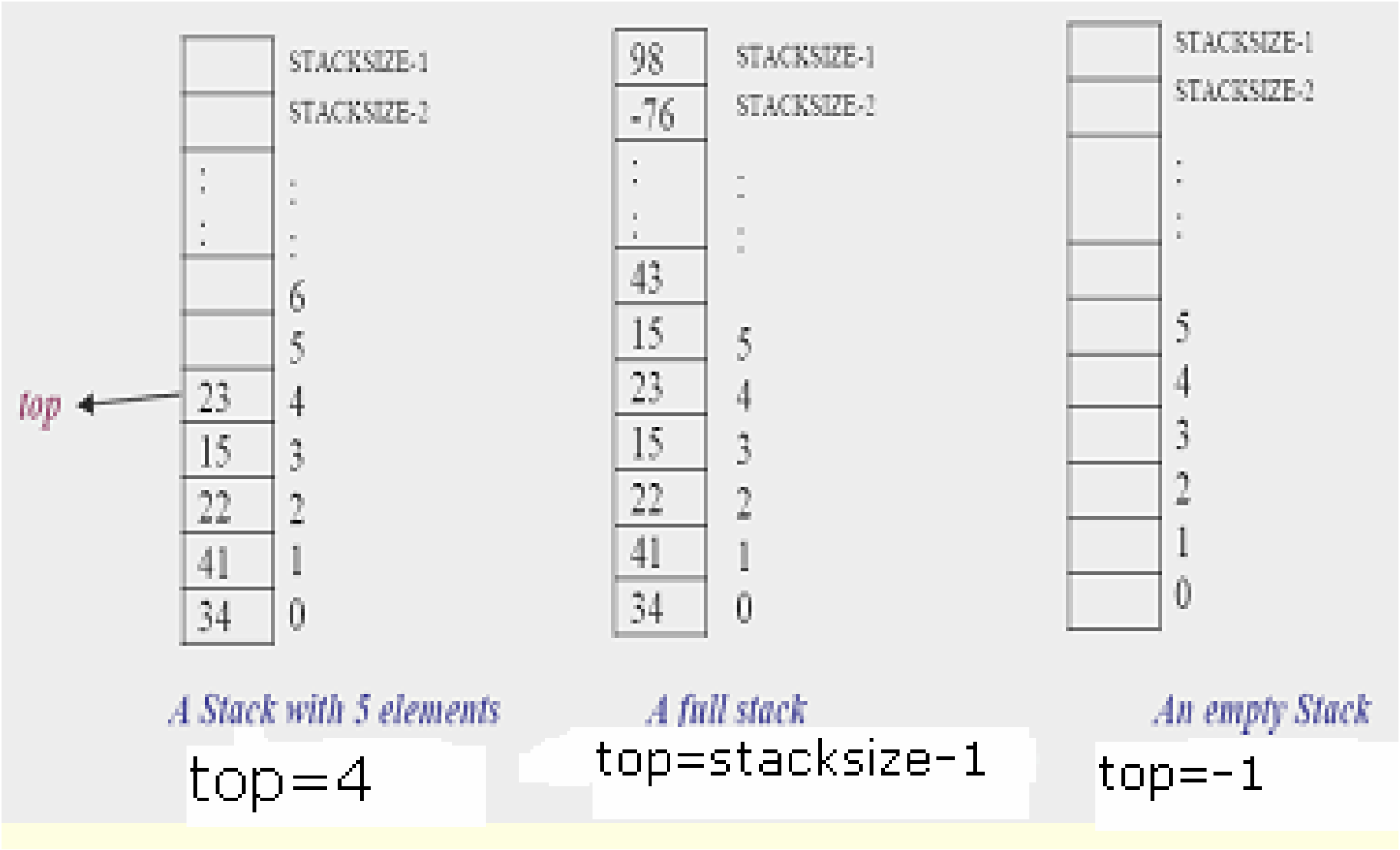
Suppose the following 6 elements are pushed , in order , onto an empty stack :

- **AAA, BBB , CCC ,DDD , EEE , FFF**



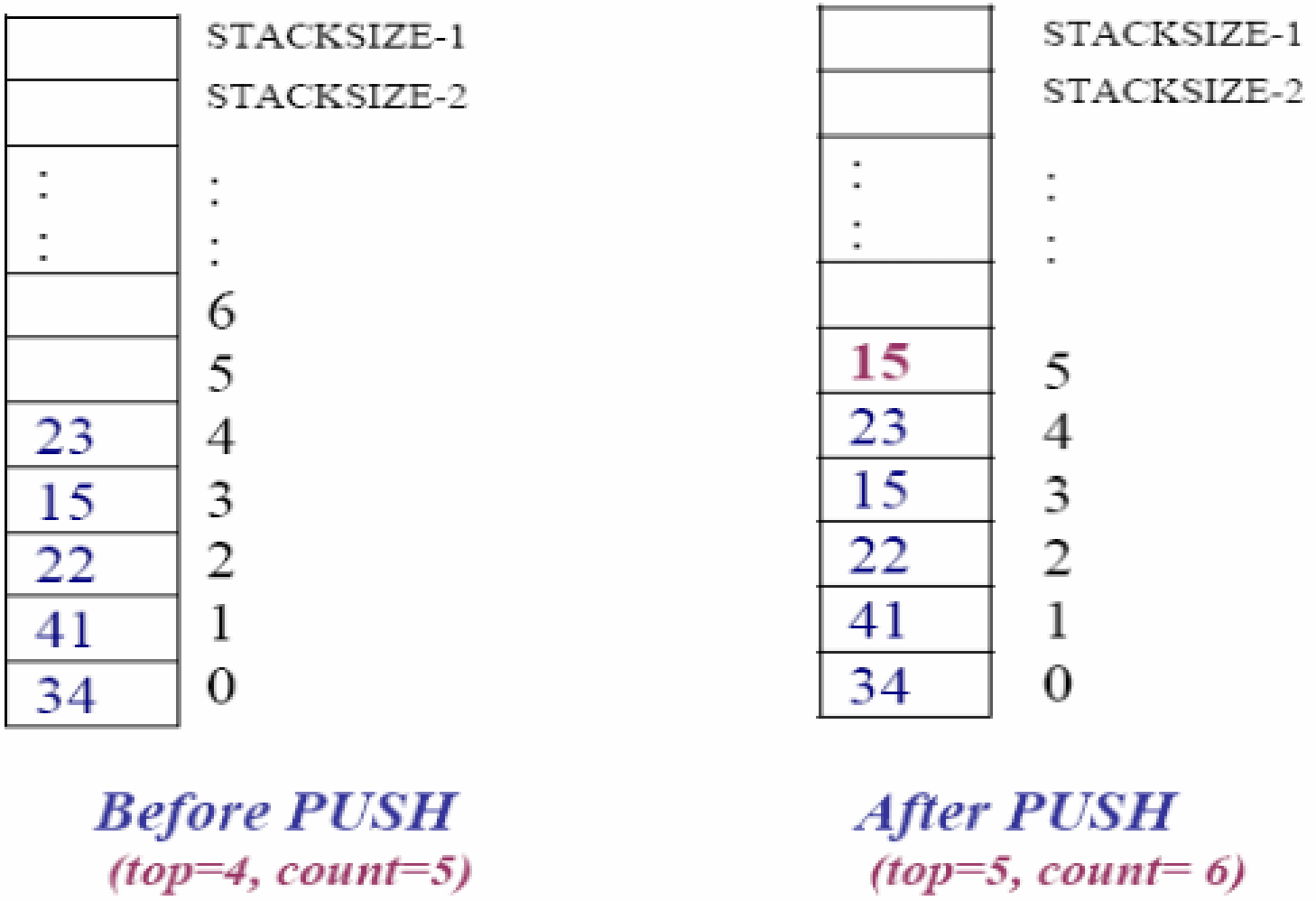
STACK: AAA, BBB , CCC ,DDD , EEE , FFF

Stacks


STACKSIZE-1
STACKSIZE-2
top
23
15
22
41
34
6
5
4
3
2
1
0
A Stack with 5 elements
top=4
98
-76
43
15
23
15
22
41
34
5
4
3
2
1
0
A full stack
top=stacksize-1
An empty Stack
top=-1

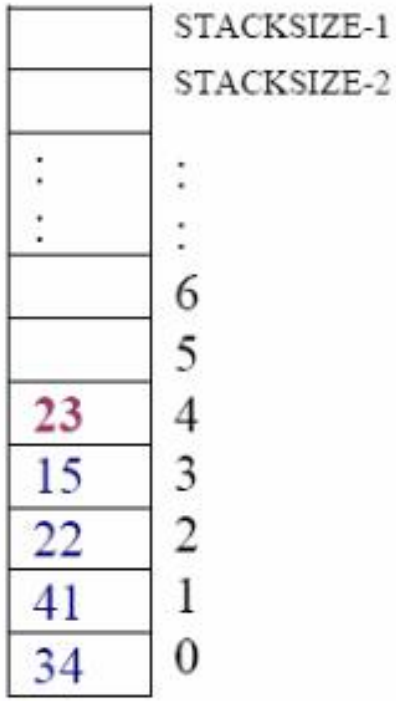
Push Operation

Push an item onto the top of the stack (***insert an item***)

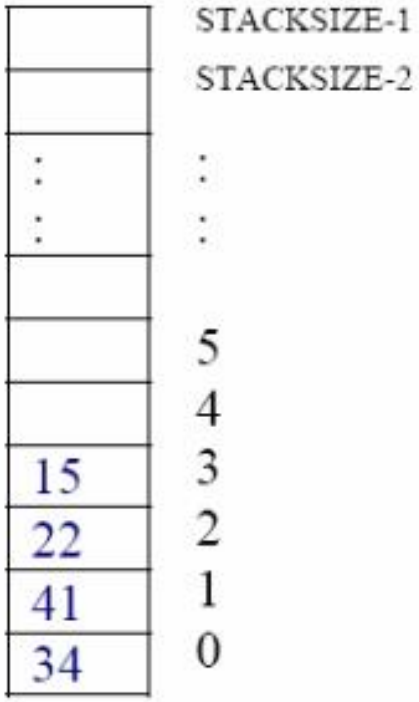


Pop Operation

Pop an item off the top of the stack (***delete an item***)



Before POP
(top=4, count=5)



After POP
(top=3 count=4)

Minimizing Overflow

- Difference between underflow and overflow in dealing with stack
- Underflow depends upon the given algorithm and the given data hence no direct control by the programmer.
- Overflow depends upon the arbitrary choice of the programmer for the amount of memory space reserved for the 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.

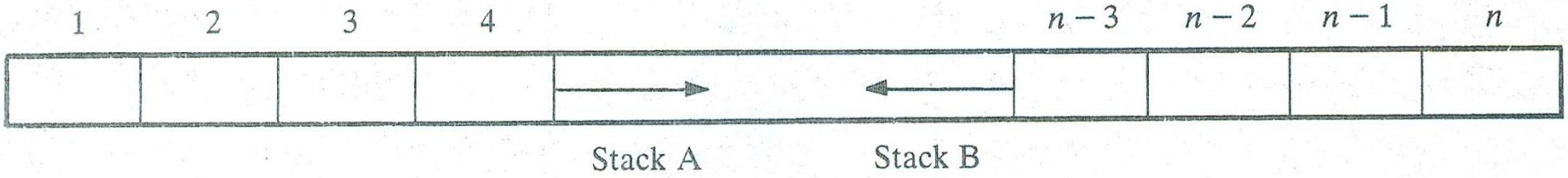


Fig. 6-6

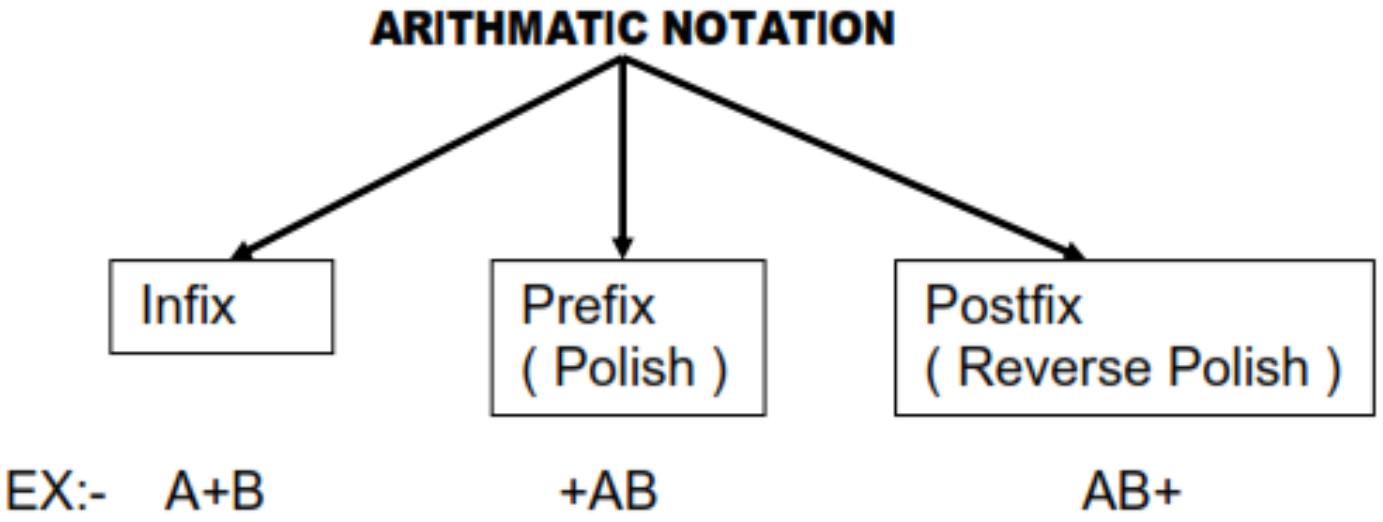
6.4 Arithmetic Expressions; Polish Notation

- Let Q be in arithmetic expression involving constants and operations.
- Discuss an algorithm which finds the value of Q by using reverse polish (postfix) notation.
- Stack is an essential tool in this algorithm.
- Binary operations in Q may have different levels of precedence.
- Specifically, we assume the following three levels of precedence for the usual five binary operations.
- Highest: Exponentiation (↑)
- Next highest: Multiplication (*) & Division (/)
- Lowest: Addition (+) & Subtraction (-)

Example 6.4

- 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.

Infix , postfix and prefix notations



Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Polish Notation

- For most common arithmetic operations, the operator symbol is placed between its two operands. For example,
- A + B, C – D, E * F, G / H
- This is called infix notation. With this notation, we must distinguish between
- (A + B) * C and A + (B * C)

  by using either parentheses or some operator-precedence convention such as the usual precedence levels.
- 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

- **Polish notations named after the polish mathematician Jan Lukasiewiez, refers to the notions in which operators placed before its two operands**
- **For ex: + AB - CD * EF / 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

Reverse Polish Notations:-

- it refers to the notation in which the operator symbol is placed after its two operands.

  For ex: AB + CD - EF * GH /

- One never needs parentheses to determine the order of the operations in any arithmetic expression written in the reverse polish notation.
- The order is defined completely by the relative order of the operands and the operators.
- This notation is frequently called postfix (or suffix) notation, whereas prefix notation is the term used for polish notation.

Evaluation of Expression

- The Computer Usually Evaluates an Arithmetic expression written in infix notation into steps
  1. First converts the expression to postfix notation
  2. Evaluates the postfix expression

- Stack is the Main Tool that is Used to Accomplish given Task.

INFIX	PREFIX
(A + B) * C	[+ A B] * C = * + A B
A + (B * C)	A + [* B C] = + A * B C
(A + B) / (C – D)	[+ AB] / [- CD] = / + AB-CD

Evaluation of a Postfix Expression

Algorithm 6.3

Suppose P is an arithmetic expression written in postfix notation.

The following algorithm, which uses a STACK to hold operands, evaluates P. This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add right parenthesis “)” at the end of P. [This acts as 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 Θ 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 Θ 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 Expression

- Algorithm 6.3
- If **P is an arithmetic expression written in postfix notation. This algorithm** uses STACK to hold operands, and evaluate **P.**

Algorithm: This algorithm finds the VALUE of **P** written in postfix notation.

1. Add a Dollar Sign "$" at the end of **P**. [This acts as sentinel.]
2. Scan **P** from left to right and repeat Steps 3 and 4 for each element of **P** until the sentinel "$" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator © 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** © **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.

Example

Following is an infix arithmetic expression

(5 + 2) * 3 – 8/ 4

And its postfix is:

5 2 + 3 * 8 4 / –

Now add **“)”** **at the end of expression as a sentinel.**

5 2 + 3 * 8 4 / – **)**

Example

5 2 + 3 * 8 4 / –)

	Scanned Elements	Stack	Action to do
(1)	5	5	Pushed on stack
(2)	2	5, 2	Pushed on Stack
(3)	+	7	Remove the two top elements and calculate 5 + 2 and push the result on stack
(4)	3	7, 3	Pushed on Stack
(5)	*	21	Remove the two top elements and calculate 7 * 3 and push the result on stack
(6)	8	21, 8	Pushed on Stack
(7)	4	21, 8, 4	Pushed on Stack
(8)	/	21, 2	Remove the two top elements and calculate 8 / 4 and push the result on stack
(9)	-	19	Remove the two top elements and calculate 21 - 2 and push the result on stack
(10)	)	19	**Sentinel) encounter ,** Result is on top of the STACK

Example 6.5

Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, -

P: 5, 6, 2, +, *, 12, 4, /, -,)

P: (1), (2), (3), (4), (5), (6), (7), (8), (9), (10)

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

Fig. 6-7

Transforming Infix Expressions into Postfix Expressions

Algorithm 6.4 POLISH(Q, P)

Suppose Q is an arithmetic expression written in infix notation.

This algorithm finds the equivalent postfix expression P.

Algorithm: Infix_to_PostFix(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 or higher precedence/priority 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 into Postfix

- The following Algorithm Transforms the Infix Expression Q into its Equivalent Postfix Expression P
- The Algorithm uses a STACK to Temporarily Hold Operators and Left Parentheses.
- The Postfix Expression P will be Constructed from Left to Right using the Operands and Left Parentheses.
- The Postfix Expression P will be Constructed from Left to Right using the Operands from Q and the Operators which are Removed from STACK
- We begin by Pushing a Left Parenthesis onto STACK and Adding Right Parentheses at the End of Q
- The Algorithm is Completed when STACK is Empty.

Transforming Infix Expressions into Postfix Expressions

1. Push “(” onto STACJK, 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 Θ 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 Θ.
   b. Add Θ 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 it to P]

   [End of If structure]

   [End of step 2 loop]
7. Exit.

Example

- Consider the following arithmetic infix expression Q:

Q: A + (B * C – (D / E ↑ F) * G) * H

- Convert **Q: A+(B * C – (D / E ^ F) * G) * H into postfix form showing stack status .**
- Now add **“)” at the end of expression**
- **A+(B * C – (D / E ^ F) * G) * H)**
- **and also Push a “(“ on Stack.**

Example

A+(B * C – (D / E ^ F) * G) * H)

	Symbol Scanned	Stack	Expression Y
		(	
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+(	A
(4)	B	(+(	AB
(5)	*	(+(*	AB
(6)	C	(+(*	ABC
(7)	-	(+(-	ABC*
(8)	(	(+(-(	ABC*
(9)	D	(+(-(	ABC*D
(10)	/	(+(-(/	ABC*D
(11)	E	(+(-(/	ABC*DE
(12)	^	(+(-(/^	ABC*DE
(13)	F	(+(-(/^	ABC*DEF
(14)	)	(+(-	ABC*DEF^/
(15)	*	(+(-*	ABC*DEF^/
(16)	G	(+(-*	ABC*DEF^/G
(17)	)	(+	ABC*DEF^/G**
(18)	(	+*	ABC*DEF^/G*H
(19)	(	+*	ABC*DEF^/G*-H
(20)	)	empty	**ABC*DEF^/G*-H*+**

Example

The elements of Q have now been labeled from left to right for easy reference. Figure 6-8 shows the status of STACK and of the string P as each element of Q is scanned. Observe that

(1) Each operand is simply added to P and does not change STACK.

(2) The subtraction operator (−) in row 7 sends * from STACK to P before it (−) is pushed onto STACK.

(3) The right parenthesis in row 14 sends ↑ and then / from STACK to P, and then removes the left parenthesis from the top of STACK.

(4) The right parenthesis in row 20 sends * and then + from STACK to P, and then removes the left parenthesis from the top of STACK.

After Step 20 is executed, the STACK is empty and

P: A B C * D E F ↑ / G * − H * +

which is the required postfix equivalent of Q.

Quick Sort An Stacks Application

Quick Sort An Stacks Application

- Quick Sort works on Divide and Conquer Rule
- Quick Sort Strategy is to Divide a List or Set into Two Sub-Lists or Sub-Sets.
- Pick an Element, Called a Pivot, from the List.
- Reorder the List so that all Elements which are Less than the Pivot come Before the Pivot and so that All Elements Greater than the Pivot come After it. After this Partitioning, the Pivot is in its Final Position. This is called the *Partition* operation.
- Recursively Sort the Sub-List of Lesser Elements and the Sub-List of Greater Elements.

Quicksort: An application of Stacks

- Let A be a list of n data items. “Sorting A” refers to the operation of rearranging the elements of A so that they are in some logical order, such as numerically ordered when A contains numerical data, or alphabetically ordered when A contains character data.
- Quicksort is an algorithm of the divide and conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.
- Suppose A is the following list of 12 numbers.
- 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
- The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. this is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. interchange 44 and 22 to obtain the list:

Quicksort: An application of Stacks

- 22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66
- (Observe that the numbers 88 and 66 to the right of 44 are each greater than 44). Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. the number is 55. interchange 44 and 55 to obtain the list:
- 22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66
- (Observe that the numbers 22, 33, and 11 to the left of 44 are each less than 44). Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list:
- 22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66
- (Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list:

Quicksort: An application of Stacks

- 22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66
- (Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. we do not meet such a number before meeting 44. this means all the numbers have been scanned and compared with 44. furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44.
- 22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

First Subset **Second Subset**

- Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

Quicksort: An application of Stacks

The above reduction step is repeated with each sublist containing 2 or more elements. Since we can process only one sublist at a time, we must be able to keep track of some sublists for future processing. This is accomplished by using two stacks, called LOWER and UPPER, to temporarily "hold" such

sublists. That is, the addresses of the first and last elements of each sublist, called its *boundary values*, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sublist only after its boundary values are removed from the stacks. The following example illustrates the way the stacks LOWER and UPPER are used.

Example 6.7

Consider the above list A with $n = 12$ elements. The algorithm begins by pushing the boundary values 1 and 12 of A onto the stacks to yield

LOWER: 1 UPPER: 12

In order to apply the reduction step, the algorithm first removes the top values 1 and 12 from the stacks, leaving

LOWER: (empty) UPPER: (empty)

and then applies the reduction step to the corresponding list A[1], A[2], . . . , A[12]. The reduction step, as executed above, finally places the first element, 44, in A[5]. Accordingly, the algorithm pushes the boundary values 1 and 4 of the first sublist and the boundary values 6 and 12 of the second sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 12

In order to apply the reduction step again, the algorithm removes the top values, 6 and 12, from the stacks, leaving

LOWER: 1 UPPER: 4

and then applies the reduction step to the corresponding sublist A[6], A[7], . . . , A[12]. The reduction step changes this list as in Fig. 6-9. Observe that the second sublist has only one element. Accordingly, the algorithm pushes only the boundary values 6 and 10 of the first sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 10

Example 6.7

- Boundary Values :
  - Address of the First and Last values of Sub-List

Lower 1	Upper 12
Lower Empty	Upper Empty
Lower 1,6	Upper 4,12
Lower 1,6	Upper 4,10

Example 6.7

And so on. The algorithm ends when the stack do not contain any sublist to be processed by the reduction step.

A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]
90	77	60	99	55	88	66
66	77	60	99	55	88	90
66	77	60	90	55	88	99
66	77	60	88	55	90	99

First Sub-List Second Sub-List

Procedure 6.5 QUICK (A, N, BEG, END, LOC)

- Here A is an array with N elements parameters BEG & END contain the boundary value of sublist of A to which this procedure applies.
- LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure.
- The local variables LEFT & RIGHT will contain the boundary values of the list of elements that have not been scanned.

Procedure 6.5 QUICK (A, N, BEG, END, LOC)

- A : Name of Array
- N : Number of Elements
- BEG : Beginning Boundary Value
- END : Ending Boundary Value
- LOC : Position of the First Element A[BEG]
- Local Variables : boundary values of the list of elements that have not been scanned
  - LEFT
  - RIGHT

Procedure 6.5 QUICK (A, N, BEG, END, LOC)

Step 1. [Initialize] Set LEFT := BEG, RIGHT := END and LOC := BEG

Step 2. [Scan from Right to Left]

a) Repeat while A[LOC] <= A[RIGHT]

RIGHT := RIGHT – 1

[End of Loop]

b) If LOC = RIGHT, then : Return

c) If A[LOC] > A [RIGHT] ,then:

1) [Interchange A [LOC] and A[RIGHT]]

TEMP := A[LOC],

A[LOC] = A[RIGHT],

A[RIGHT]=TEMP

2) Set LOC := RIGHT

3) Go to Step 3

[End of If Structure]

Procedure 6.5 QUICK (A, N, BEG, END, LOC)

Step 3. [Scan from Left to Right]

a) Repeat while A[LEFT] <= A[LOC]

LEFT := LEFT + 1

[End of Loop]

b) If LOC = LEFT, then : Return

c) If A[LEFT] > A [LOC] ,then:

1) [Interchange A [LEFT] and A[LOC]]

TEMP := A[LOC], A[LOC] = A[LEFT],
A[LEFT]=TEMP

2) Set LOC := LEFT

3) Go to Step 2

[End of If Structure]

Procedure 6.6: Quicksort algorithm

1. [Initialize] TOP := 0
2. [PUSH Boundary values of A onto Stacks when 2 or More Elements]
   If N > 1, then TOP:=TOP+1, LOWER[1]:=1, UPPER[1]:=N
3. Repeat Steps 4 to 7 while TOP != 0
4. [Pop Sub-List from Stacks]
   Set BEG := LOWER[TOP] , END := UPPER[TOP]
   TOP := TOP -1
5. Call QUICK (A, N, BEG, END, LOC)
6. [Push Left Sub-List onto Stacks when 2 or More Elements]
   If BEG < LOC - 1 then TOP := TOP + 1, LOWER[TOP] := BEG,
   UPPER[TOP] := LOC – 1 [End of If Structure]
7. [Push Right Sub-List onto Stacks when 2 or More Elements]
   If LOC + 1 < END then TOP := TOP + 1, LOWER[TOP] := LOC + 1,
   UPPER[TOP] := END [End of If Structure] [End of Step 3 Loop]
8. [Exit]

(Quick Sort)

Complexity of the Quicksort algorithm

The running time of a sorting algorithm is usually measured by the number $f(n)$ of comparisons required to sort n elements. The quicksort algorithm, which has many variations, has been studied extensively. Generally speaking, the algorithm has a worst-case running time of order $n^2/2$, but an average-case running time of order $n \log n$. The reason for this is indicated below.

The worst case

The worst case occurs when the list is already sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have $n - 1$ elements. Accordingly, the second element will require $n - 1$ comparisons to recognize that it remains in the second position. And so on. Consequently, there will be a total of

$$f(n) = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

comparisons. Observe that this is equal to the complexity of the bubble sort algorithm (Sec. 4.6).

Complexity of the Quicksort algorithm

The average case

The complexity $f(n) = O(n \log n)$ of the average case comes from the fact that, on the average, each reduction step of the algorithm produces two sublists. Accordingly:

(1) Reducing the initial list places 1 element and produces two sublists.

(2) Reducing the two sublists places 2 elements and produces four sublists.

(3) Reducing the four sublists places 4 elements and produces eight sublists.

(4) Reducing the eight sublists places 8 elements and produces sixteen sublists.

And so on. Observe that the reduction step in the kth level finds the location of 2^{k-1} elements; hence there will be approximately $\log_2 n$ levels of reductions steps. Furthermore, each level uses at most n comparisons, so $f(n) = O(n \log n)$. In fact, mathematical analysis and empirical evidence have both shown that

$$f(n) \approx 1.4\lceil n \log n \rceil$$

is the expected number of comparisons for the quicksort algorithm.

تم الإنتهاء من المحاضرة