Stacks(unit 2.1)

DEFINITION

A *stack* is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only. The insertion and 'deletion operations in the case of a stack are specially termed PUSH and POP, respectively, and the position of the stack where these operations are performed is known as the TOP of the stack. An element in a stack is termed an ITEM. The maximum number of elements that a stack can accommodate is termed SIZE. Figure shows a typical view of a stack data structure.

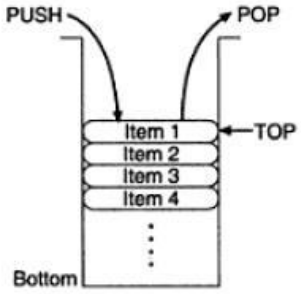


Figure 4.2 Schematic diagram of a stack.

REPRESENTATION OF A STACK

A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list.

Array Representation of Stacks: First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.

In Figure, Itemi denotes the *ith* item in the stack; l and *u* denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack. With this representation, the following two ways can be stated:

EMPTY: TOP < l
FULL: TOP ≥ *u*

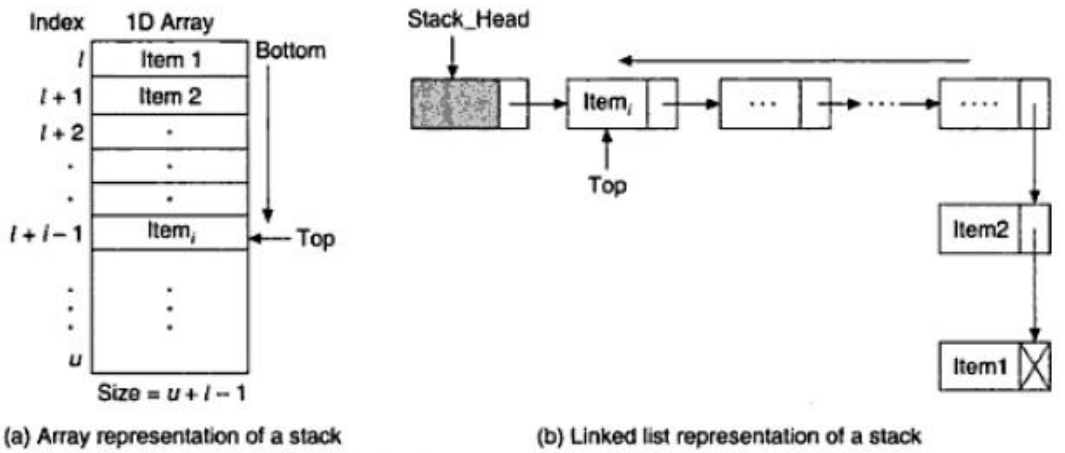


Figure 4.3 Two ways of representing stacks.

Linked List Representation of Stacks: Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications, the size of the stack may vary during program execution. An obvious solution to this problem is to represent a stack using a linked list. A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usual, to point to the next' item. Above Figure b depicts such a stack using a single linked list.

In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item. Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list.

OPERATIONS ON STACKS

The basic operations required to manipulate a stack are:

PUSH:To insert an item into a stack, POP:To remove an item from a stack,
STATUS: To know the present state of a stack

Algorithm Push_Array
Input: The new item ITEM to be pushed onto it.
Output: A stack with a newly pushed ITEM at the TOP position.
Data structure: An array *A* with TOP as the pointer.

Steps:

```
1. If TOP ≥ SIZE then
2.     Print "Stack is full"
3. Else
4.     TOP = TOP + 1
5.     A[TOP] = ITEM
6. EndIf
7. Stop
```

Here, we have assumed that the array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack. The following algorithm *Pop_Array* defines the POP of an item from a stack which is represented using an array A.

Algorithm Pop_Array
Input: A stack with elements.
Output: Removes an ITEM from the top of the stack if it is not empty.
Data structure: An array *A* with TOP as the pointer.

Steps:

```
1. If TOP < 1 then
2.     Print "Stack is empty"
3. Else
4.     ITEM = A[TOP]
5.     TOP = TOP – 1
6. EndIf
7. Stop
```

In the following algorithm *Status_Array*, we test the various states of a stack such as whether it is full or empty, how many items are right now in it, and read the current element at the top without removing it, etc.

Algorithm Status_Array
Input: A stack with elements.
Output: States whether it is empty or full, available free space and item at TOP.
Data structure: An array *A* with TOP as the pointer.

Steps:

```
1. If TOP < 1 then
2.     Print "Stack is empty"
3. Else
4.     If (TOP ≥ SIZE) then
5.         Print "Stack is full"
6.     Else
7.         Print "The element at TOP is", A[TOP]
8.         free = (SIZE – TOP)/SIZE * 100
9.         Print "Percentage of free stack is", free
10.    EndIf
11. EndIf
12. Stop
```

Now let us see how the same operations can be defined for a stack represented with a single linked list.

Algorithm Push_LL

Input: ITEM is the item to be inserted.

Output: A single linked list with a newly inserted node with data content ITEM.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

```
1. new = GetNode(NODE)
   /* Insert at front */
2. new→DATA = ITEM
3. new→LINK = TOP
4. TOP = new
5. STACK_HEAD→LINK = TOP
6. Stop
```

Algorithm Pop_LL

Input: A stack with elements.

Output: The removed item is stored in ITEM.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

```
1. If TOP = NULL
2.     Print "Stack is empty"
3.     Exit
4. Else
5.     ptr = TOP→LINK
6.     ITEM = TOP→DATA
7.     STACK_HEAD→LINK = ptr
8.     TOP = ptr
9. EndIf
10. Stop
```

Algorithm Status_LL( )

Input: A stack with elements.

Output: Status information such as its state (empty or full), number of items, item at the TOP.

Data structure: A single linked list structure whose pointer to the header is known from STACK_HEAD and TOP is the pointer to the first node.

Steps:

```
1. ptr = STACK_HEAD→LINK
2. If (ptr = NULL) then
3.     Print "Stack is empty"
4. Else
5.     nodeCount = 0
6.     While (ptr ≠ NULL) do
7.         nodeCount = nodeCount + 1
8.         ptr = ptr→LINK
9.     EndWhile
10.    Print "The item at the front is", TOP→DATA, "Stack contains", nodeCount,
       "Number of items"
11. EndIf
12. Stop
```

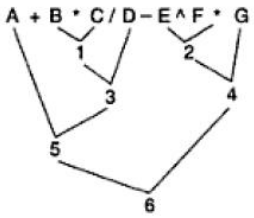
APPLICATIONS. OF STACKS

1. Evaluation of Arithmetic Expressions: An arithmetic expression consists of operands and operators. Operands are variables or constants and operators are of various types such as arithmetic unary and binary operators and Boolean operators. In addition to these, parentheses such as '(' and ')' are also used. A simple arithmetic expression is cited below: *A+B*C/D-E^F*G*

Table 4.1 Precedence and associativity of operators

Operators	*Precedence*	*Associativity*
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right

Thus, with the above rules of precedence and associativity of operators, the evaluation will take place for the above-mentioned expression in the sequence (sequence is according to the number 1, 2, 3, ... , etc.) stated below:



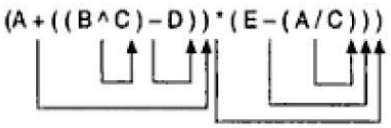
Notations for arithmetic expressions

There are three notations to represent an arithmetic expression, viz. infix, prefix and postfix (or suffix). The conventional way of writing an expression is called infix. For example,

A + B, C - D, E * F, G/H, etc.

Here, the notation is: <operand> <operator> <operand>.

Input: ((A + ((B ^ C) – D)) * (E – (A/C)))
(A fully parenthesized expression)



(Arrows point from operators to their corresponding right parenthesis.)

((A ((B C ^ D – + (E (AC / – *

(Operators are moved to their respective right parentheses.)

Output: A B C ^ D – + E A C / – *

(All parentheses are removed yielding the postfix expression.)

A similar technique can be applied to obtain the prefix notation for a given infix notation but moving the operators corresponds to the left parenthesis.

Three notations for the given arithmetic expression are listed below:

Infix: ((A + ((B ^ C) – D)) * (E – (A/C)))
Prefix: * + A – ^ BCD – E/AC
Postfix: ABC ^ D – + EAC / – *

This is called *infix* because the operator comes in between the operands. The *prefix* notation, on the other hand, uses the convention. <operator> <operand> <operand>
Here, the operator come before the operands. The following are simple expressions in prefix notation: +AB, -CD, *EF, *IGH*, etc.

The last notation is called the *postfix* (or suffix) notation where the operator is suffixed by operands: <operand><operand><operator>
The following expressions are in postfix notation: AB+, CD-, EF*, GH/, etc.
The following example illustrates this conversion. For simplicity, let us consider a fully parenthesized expression.

Conversion* of *an infix expression to postfix expression
First, we have to append the symbol ')' as the delimiter at the end of a given infix expression and initialize the stack with '('. These symbols ensure that either the input or the stack is exhausted.

ReadSymbol(): From a given infix expression, this will read the next symbol.
ISP(X): Returns the in-stack priority value for a symbol X.
ICP(X): This function returns the in-coming priority value for a symbol X.
Output(X): Append the symbol X into the resultant expression.

Algorithm InfixToPostfix
Input: E, simple arithmetic expression in infix notation delimited at the end by the right parenthesis ')', incoming and in-stack priority values for all possible symbols in an arithmetic expression.
Output: An arithmetic expression in postfix notation.
Data structure: Array representation of a stack with TOP as the pointer to the top-most element.

Steps:

```
1.  TOP = 0, PUSH('(')                          // Initialize the stack
2.  While (TOP > 0) do
3.      item = E.ReadSymbol()                   // Scan the next symbol in infix expression
4.      x = POP()                               // Get the next item from the stack
5.      Case: item = operand                    // If the symbol is an operand
6.          PUSH(x)                             // The stack will remain same
7.          Output(item)                        // Add the symbol into the output expression
8.      Case: item = ')',                       // Scan reaches to its end
9.          While x ≠ '(' do                    // Till the left match is not found
10.             Output(x)
11.             x = POP()
12.         EndWhile
13.     Case: ISP(x) ≥ ICP(item)
14.         While (ISP(x) ≥ ICP(item)) do
15.             Output(x)
16.             x = POP()
17.         EndWhile
18.         PUSH(x)
19.         PUSH (item)
20.     Case: ISP(x) < ICP(item)
21.         PUSH(x)
22.         PUSH (item)
23.     Otherwise:
            Print "Invalid expression"
24. EndWhile
25. Stop
```

EXAMPLE: Let us illustrate the procedure *InfixToPostfix* with the following arithmetic expression:

Input: (A + B)^ C - (D * E) / F) (infix form)

Read symbol	*Stack*	*Output*
Initial	(	
1	((	
2	((	A
3	((+	A
4	((+	AB
5	(	AB+
6	(^	AB+
7	(^	AB + C
8	(–	AB + C ^
9	(– (	AB + C ^
10	(– (	AB + C ^ D
11	(– (*	AB + C ^ D
12	(– (*	AB + C ^ DE
13	(–	AB + C ^ DE *
14	(– /	AB + C ^ DE *
15	(– /	AB + C ^ DE * F
16		AB + C ^ DE * F / –

Output: A B + C ^ DE * F / – (postfix form)

Evaluation of* a *postfix expression

Algorithm EvaluatePostfix

Input: E, an expression in postfix notation, with values of the operands appearing in the expression.

Output: Value of the expression.

Data structure: Array representation of a stack with TOP as the pointer to the top-most element.

Steps:

```
1.  Append a special delimiter '#' at the end of the expression
2.  item = E.ReadSymbol( )                          // Read the first symbol from E
3.  While (item ≠ '#') do
4.      If (item = operand) then
5.          PUSH(item)                              // Operand is the first push into the stack
6.      Else
7.          op = item                               // The item is an operator
8.          y = POP( )                              // The right-most operand of the current operator
9.          x = POP( )                              // The left-most operand of the current operator
10.         t = x op y                              // Perform the operation with operator 'op' and operands x, y
11.         PUSH(t)                                 // Push the result into stack
12.     EndIf
13.     item = E.ReadSymbol( )                      // Read the next item from E
14. EndWhile
15. value = POP( )                                  // Get the value of the expression
16. Return(value)
17. Stop
```

EXAMPLE 4.2

To illustrate the algorithm *EvaluatePostfix*, let us consider the following expression:

Infix: A + (B * C) / D
Postfix: A B C * D / +
Input: A B C * D / + # with A = 2, B = 3, C = 4, and D = 6

Read symbol	*Stack*	
A	2	PUSH(A = 2)
B	2 3	PUSH(B = 3)
C	2 3 4	PUSH(C = 4)
*	2 12	POP(4), POP(3), PUSH(T = 12)
D	2 12 6	PUSH(D = 6)
/	2 2	POP(6), POP(12), PUSH(T = 2)
+	4	POP(2), POP(2), PUSH(T = 4)
#		value = POP( )

Conversion of a *postfix expression to* a *code*

Using a stack, we can easily generate an assembly code for an expression given in reverse Polish notation (postfix). In order to simplify our example, we will assume the arithmetic expressions with four arithmetic operations- + (addition), - (subtraction), * (multiplication) and / (division) only-and the assembly codes are in single address form. The following assembly code mnemonics are assumed:

LDA *A* — To load the accumulator with the memory content of *A* and the content of *A* will remain unchanged.

STA *B* — To store the content of the accumulator in memory location *B*.

ADD *A* — To add the value of memory content *A* with the value of the accumulator and the result will be stored in the accumulator; the value of memory content *A* will remain unchanged.

SUB *B* — To subtract the value of memory content B from the value of the accumulator and the result will be stored in the accumulator; the memory content *B* will remain unchanged.

MUL *C* — To multiply the value of memory content *C* with the value of the accumulator and the result will be stored in the accumulator; the memory content *C* will remain unchanged.

DIV *D* — To divide the value of the accumulator by the value of the memory content *D* and the result of the division will be stored in the accumulator; the value of the memory content *D* will remain unchanged.

EXAMPLE: For example, let the infix expression be *A + B* and its postfix form be *AB+*. The assembly code for this expression will be

LDA A; ADD B; STA T

For writing such codes, let us assume one procedure *ProduceCode(A, B, op, Temp)* with four arguments. For instance, with *AB +*, *op* is ADD and *Temp* is *T*. With these, the algorithm for converting a postfix expression to its equivalent assembly code, *PostfixToCode,* is framed as follows:

Algorithm Postfix To Code
Input: An arithmetic expression E in postfix notation.
Output: Assembly code.
Data structure: A stack with TOP as the pointer to the top-most element.

```
Steps:
1.  Append a delimiter '#' at the and of the expression
2.  item = E.ReadSymbol()                    // Read the first symbol from the expression
3.  i = 1, TOP = 0                           // Stack is initialized; an integer will be used as index
4.  While (item ≠ '#') do
5.      Case: item = operand
6.          PUSH(item)                       // Push the item into the stack
7.      Case: item = '+'
8.          x = POP()                        // Pop two operands from the stack
9.          y = POP()
10.         ProduceCode(y, x, 'ADD', Ti)     // Ti is the ith temporary
11.         PUSH (Ti)
12.     Case: item = '–'
13.         x = POP()
14.         y = POP()
15.         ProduceCode(y, x, 'SUB', Ti)
16.         PUSH(Ti)
17.     Case: item = '*'
18.         x = POP()
19.         y = POP()
20.         ProduceCode(y, x, 'MUL', Ti)
21.         PUSH(Ti)
22.     Case: item = '/'
23.         x = POP()
24.         y = POP()
25.         ProduceCode (y, x, 'DIV', Ti)
26.         PUSH(Ti)
27.     Otherwise:
28.         Print "Error in input"
29.         Exit
30.     item = E.ReadSymbol()                // Read for the next symbol from E
31.     i = i + 1                            // The index is incremented
32. EndWhile
33. Stop
```

EXAMPLE:

The above algorithm is illustrated with the following example:

Infix: (A + B) * C / D

Postfix: AB + C * D /

Input: AB + C * D /

The production of codes according to the algorithm *PostfixToCode* is given below:

Scanned symbol	*Content of stack*	*Action*	*Code generated*
A	A	PUSH(A)	
B	AB	PUSH(B)	
+	T1	x = B, y = A PRODUCE_CODE(A, B, 'ADD',T1) PUSH(T1)	LDA A ADD B STA T1
C	T1 C	PUSH(C)	
*	T2	x = C, y = T1 PRODUCE_CODE(T1, C, 'MUL',T2) PUSH(T2)	LDA T1 MUL C STA T2
D	T2 D	PUSH(D)	
/	T3	x = D, y = T2 PRODUCE_CODE(T2, D, 'DIV', T3) PUSH(T3)	LDA T2 DIV D STA T3
#	T3	Stop	

2. Code Generation for stack Machines

let us consider the following arithmetic expressions:

A=B*C-A

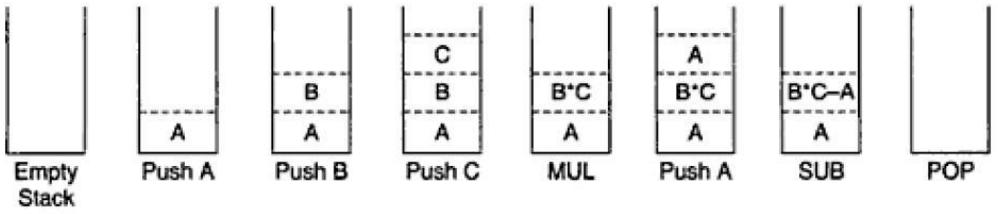
The corresponding postfix notation can be obtained as follows:

ABC*A-=

The instruction code according to the stack machine is as given below:

```
PUSH A      // Load operand A into the stack
PUSH B      // Load operand B into the stack
PUSH C      // Load operand C into the stack
MUL         // Multiply B * C
PUSH A      // Load operand A into the stack
SUB         // Subtract B * C – A
POP A       // Store the result in the memory location for A
```

The various states of the stack are depicted in Figure 4.5.



To generate machine codes for a stack machine when an arithmetic expression is given in postfix notation, an algorithm *PostfixToCodeForStackMachine* is described below.

Algorithm PostfixToCodeForStackMachine
Input: An arithmetic expression *E* in postfix notation.
Output: Equivalent codes for stack machine
Data structure: Array representation of a stack with TOP as the pointer to the top-most element.

Steps:

```
1.  Add a delimiter '#' at the end of the expression
2.  item = E.ReadSymbol()                    // To read an element from the expression E
3.  While (item ≠ '#') do
4.      If (item = anOperand)                // For the operand only
5.          ProduceCode('PUSH', item)
6.      Else                                 // Item is the operator
7.      Case: item = '+'
8.          ProduceCode ('ADD')
9.      Case: item = '–'
10.         ProduceCode ('SUB')
11.     Case: item = '*'
12.         ProduceCode ('MUL')
13.     Case: item = '/'
14.         ProduceCode ('DIV')
15.     EndIf
16.     item = E.ReadSymbol()                // Read the next symbol from E
17. EndWhile
18. Stop
```

3. Implementation of Recursion

$n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$

As a simple example, let us consider the case of calculation of the factorial value for an integer n.

Or

$n! = n \times (n - 1)!$

The last expression is the recursive description of the factorial whereas the first is the iterative definition. Using a pseudo code, the above two types of definitions are expressed as follows:

Factorial_I
Input: An integer number *N*.
Output: The factoral value of *N*, that is *N*!.
Remark: Code using the iterative definition of factorial.

Steps:

1. fact =1
2. **For** (i = 1 to N) **do**
3. fact = i * fact
4. **EndFor**
5. **Return**(fact) // Return the result
6. **Stop**

Here, Step 2 defines the iterative definition for the calculation of a factorial. Now, let us see the recursive definition of the same.

Factorial_R //Code using the recursive definition of factorial
Input: An integer number *N*.
Output: The factoral value of *N*, that is *N*!.
Remark: Code using the recursive definition of factorial.

Steps:

1. **If** (N = 0) **then** // Termination condition of repetition
2. fact = 1
3. **Else**
4. fact = N * **Factorial_R**(N – 1)
5. **EndIf**
6. **Return**(fact) // Return the result
7. **Stop**

We will illustrate the implementation of three popular recursive computations:
1. Calculation of factorial value
2. Quick sort
3. Tower of Hanoi problem

For each problem, we will describe the recursive description, then the translation of the recursive description to a non-recursive version using stacks.

4. Factorial Calculation

Factorial(*N*)

Steps:

1. **If** (N = 0) **then**
2. fact = 1
3. **Else**
4. fact = N * **Factorial**(N – 1)
5. **EndIf**
6. **Return** (fact)
7. **Stop**

To implement the above, we require two stacks: one for storing the parameter *N* and another to hold the return address. No stack is necessary to store local variables, as the procedure does not possess any local variable. Let these two stacks be PARAM (for parameter) and ADDR (for return address).

We assume PUSH(X, Y) operation to push the items X and Y into the stack PARAM and ADDR, respectively.

Algorithm FactorialWithStack

Input: An integer N, and MAIN, the address of the main routine, say.
Output: Factorial value of N (that is $N!$).
Data structure: Array representation of stack.

Steps:

```
1.  val = N, top = 0, addr = Step 15
2.  PUSH (val, addr)                                   // Initialize the stack
3.  val = val – 1, addr = Step 11                      // Next value and return address
4.  If (val = 0 ) then
5.      fact = 1
6.      Go to Step 12
7.  Else
8.      PUSH(val, addr)        // Val pushed into PARAM and addr pushed into ADDR
9.      Go to Step 3
10. EndIf
11. fact = val * fact
12. val = POP_PARAM( ), addr = POP_ADDR( )
13. Go to addr
14. Return (fact)
15. Stop
```

The above implementation is illustrated for $N = 5$ in Figure

Action	Execution	Step	Stack content
1.	val=5, TOP=0 addr = Step 10	1	
	PUSH(5, Step 10)	2	PARAM 5
	val=4, addr = Step7	3	ADDR 10
2.	val <> 0	4	5 4 / 10 7
	PUSH(4, Step 7)	6	
3.	val=3, addr=Step7	3	5 4 3 / 10 7 7
	val <> 0	4	
	PUSH(3, Step7)	6	
4.	val=2, addr=Step7	3	5 4 3 2 / 10 7 7 7
	val <> 0	4	
	PUSH(2, Step7)	6	
5.	val=1, addr=Step7	3	5 4 3 2 1 / 10 7 7 7 7
	val <> 0	4	
	PUSH(1, Step 7)	6	
6.	val=0, addr=Step7	3	5 4 3 2 1 / 10 7 7 7 7
	val = 0	4	
	fact = 1	5	

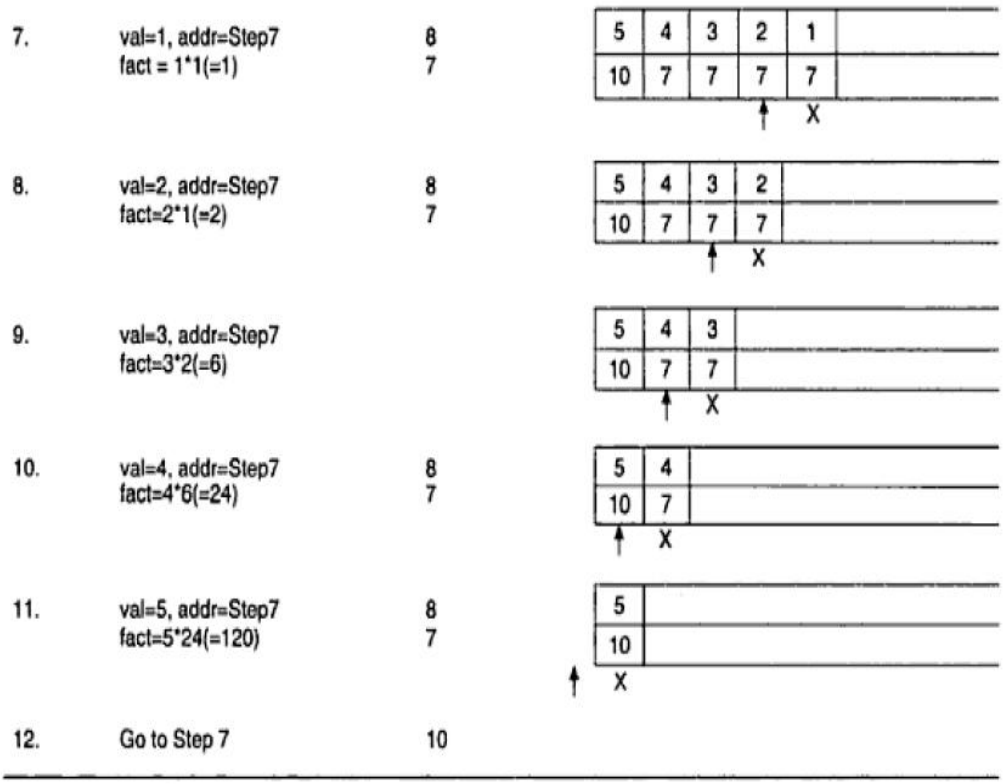


Figure 4.6 Computation of a factorial (recursively) using a stack.

5. Quick Sort

The *quick sort* algorithm is based on the *divide and conquer* technique. The principle behind the divide and conquer technique is to divide a problem into a number of sub-problems. Again each sub-problem is divided into a number of smaller sub-problems and so on till a sub-problem is not decomposable. Solving a problem means solving all the sub-problems.

In the case of quick sort, the list to be sorted is partitioned into two sub-lists so that sorting these two sub-lists is sorting the main list; sorting the sub-list again follows the same procedure recursively. Note that partition is to be done in such a way that sorting of the sub-lists is the sorting of the original list. The question is how can this be done. One simple idea is to select any element in the list (let it be the first element and be termed *pivot* element). Place the pivot element on the list so that all the elements before the pivot element are smaller and all the elements after the pivot element are larger than it. As an illustration, let us consider the following list of numbers to be sorted:

41	79	65	35	21	48	59	87	52	**28** ↑

Here, 41 is selected as the pivot element, which is encircled. In order to place 41 in its right position, first compare this element with the element at the extreme right end (shown with an upright arrow below it, we call this the pointer), swap the elements if they are not in order (that is, if the *element at the extreme right end is smaller than the pivot element);* otherwise move the pointer to left one step and repeat the comparison. In the given list, we see that the pivot element is greater than the element at the extreme right end and hence they are swapped. The list after the swap operation is shown below:

28	**79** ↑	65	35	21	48	59	87	52	**41**

We see that this swap places the pivot element at the extreme right end. We now compare the pivot element with the element which is next to the element just swapped, see the pointer). In this case, a swap will occur if the comparison tells that they are not in order (that is, the *pivot element is smaller than the element on the left),* otherwise move the pointer to right one step and repeat the comparison. In the above list, the comparison leads to a swap operation and the list after the swap appears as shown below.

28	**41**	65	35	21	48	59	87	**52** ↑	79

Repeatedly applying the above steps one can get the following observations. Comparisons with the pivot will shift the pointer to the left side from 52 to 87, 87 to 59, 59 to 48 and 48 to 21, when we find that the pivot element is larger than 21. So a swap will take place:

28	21	**65** ↑	35		48	59	87	52	79
				41					

Comparing from left, we get the following change:

28	21		**35** ↑	65	48	59	87	52	79
		41							

Another comparison from right yields

28	21	35	**41**	65	48	59	87	52	79

we observe that 41 is now placed in its final position, and we get two sub-lists, one containing all the elements to the left of 41 which are smaller than 41 and another list to the right of 41 containing elements which are larger than 41.

28	21	35	**41**	65	48	59	87	52	79

After getting the two sub-lists, we have to apply the same procedure on each sub-list. Then a sub-list is divided into two other sub-lists and the repetition continues until there is a sub-list containing no elements or a single element (this is the terminal stage of the repetition over a sub-list). Basically, the repetition is in a recursive manner. Now, here is the application of a stack. Whenever a sub-list is divided into two sub-lists, one sub-list has to be pushed onto the stack before sorting the other list. When taking care of this list, pop the next list to be considered and the procedure will continue till the stack is empty.
The quick sort algorithm now can be defined recursively as follows: Let L be the original list and FL, EL be the locations of the front and end elements of L, respectively.

Algorithm QuickSort
Input: L is the list of elements under sorting with FL and EL being the two pointers at the two extremes.
Output: The list *L* is sorted in ascending order.
Remarks: List is in the form of an array and the algorithm is defined recursively.

```
1.  If (EL–FL ≤ 1) then      // Termination check: List is empty or contains a single element
2.      Exit                 // Terminate the QuickSort for the list
3.  EndIf
4.  loc = Divide(FL, EL)     // Partition the list into two sub-lists
5.  If (loc – FL) > 1        // If the leftmost list L contains more than one element
6.      QuickSort(FL, loc–1) // Apply quick sort on the leftmost list
7.  EndIf
8.  If (EL–loc) > 1          // If the rightmost sub-list contains more than one element
9.      QuickSort(loc+1, EL) // Apply quick sort on the rightmost list
10. EndIf
11. Stop
```

Note that a sub-list can be identified by the location of its two extreme elements. Here *SizeOf(L)* is assumed to determine the number of elements in *L*. Before going to describe quick sort using a stack, let us assume the following:

(i) The list of elements is stored in an array *A*.
(ii) In order, to push a list onto the stack we actually push the locations of the first element and the last element; hence two stacks are required. Let these two stacks be represented using two arrays, namely FRONT and END.

Algorithm Divide

Input: FL and EL are boundaries, that is, the locations of the front and end elements of the list to be divided.

Output: LOC is the location of the pivot element which is between the two sub-lists after the divide.

Data structure: Array representation of a stack with TOP as the pointer to the top-most element.

Steps:

```
1.  left = FL, right = EL          // Initialization: left and right are two pointers at the extremes
2.  loc = FL                       // loc denotes the location of the pivot
3.  While (loc ≠ right) and (A[loc] ≤ A[right]) do  // Compare from right, pivot is being at left
4.      right = right–1            // Move to the left
5.  EndWhile
6.  If (loc = right) then          // List is scanned fully or list contains a single element
7.      Return (loc)               // Element is placed in its final position
8.  Else                           // Elements are not in order and hence swap
9.      Swap (A[loc], A[right])    // Interchange the pivot and the element on the right of it
10.     left = loc+1               // Set the left marker
11.     loc = right                // New position of the pivot element
12. EndIf
```

```
13. While (loc ≠ left) and (A[loc] ≥ A[left]) do  // Compare from left pivot is being at right
14.     left = left + 1            // Move to the right
15. EndWhile
16. If (loc = left) then           // List is fully scanned as it contains a single element
17.     Return(loc)                // Element is placed in its final position
18. Else                           // Elements are not in order and swap
19.     Swap (A[loc], A[left])     // Interchange the pivot and the element on the left of it
20.     right = loc – 1            // Set the right marker
21.     loc = left                 // New position of the pivot element
22. EndIf
23. Go to Step 3                   // Repeat the steps of scanning
24. Stop
```

Now, we shall define the quick sort algorithm.

Algorithm QuickSort

Input: An array *A* with *N* elements

Output: Sorted list of elements in *A* in ascending order

Data structure: Array representation of stack with TOP as the pointer to the top-most element.

Steps:

```
1.  fl = 1, el = N                 // Boundaries of the list
2.  top = NULL,                    // Stacks are empty initially
3.  If (N > 1)                     // If the list is not empty or has more than a single element
4.      PUSH(fl, el)               // Push the values into their respective stacks
5.  EndIf
6.  While (top ≠ NULL) do          // Till the stack is not empty
7.      POP(fl, el)                // Pop a sub-list from stacks
8.      Divide (fl, el, loc)       // Divide the list into two sub-lists and get the position of pivot
9.      If (fl < loc – 1) then     // Test for left sub-list whether it has more than one element
10.         PUSH(fl, loc – 1)      // The left sub-list is large enough and will be considered later
11.     EndIf
12.     If (el > loc + 1) then     // Test for right sub-list whether it has more than one element
13.         PUSH(loc + 1, el)      // The right sub-list is large enough and will be considered later
14.     EndIf
15. EndWhile
16. Stop
```

In the above algorithm *QuickSort( )*, we assume *PUSH(FL,* EL) to push FL and EL into the FRONT and END of stacks, respectively. Similarly, *POP(FL,* EL) is to *POP* the items from two stacks FRONT and END and they are stored as FL and EL, respectively. The details of the quick sort method are illustrated through an example as shown in below figure. Note that an element with a dotted circle indicates that the element is placed at the final position. The circled element is the present pivot element. A 'X' in the stack pointer position indicates the deletion of the entry, that is a POP.

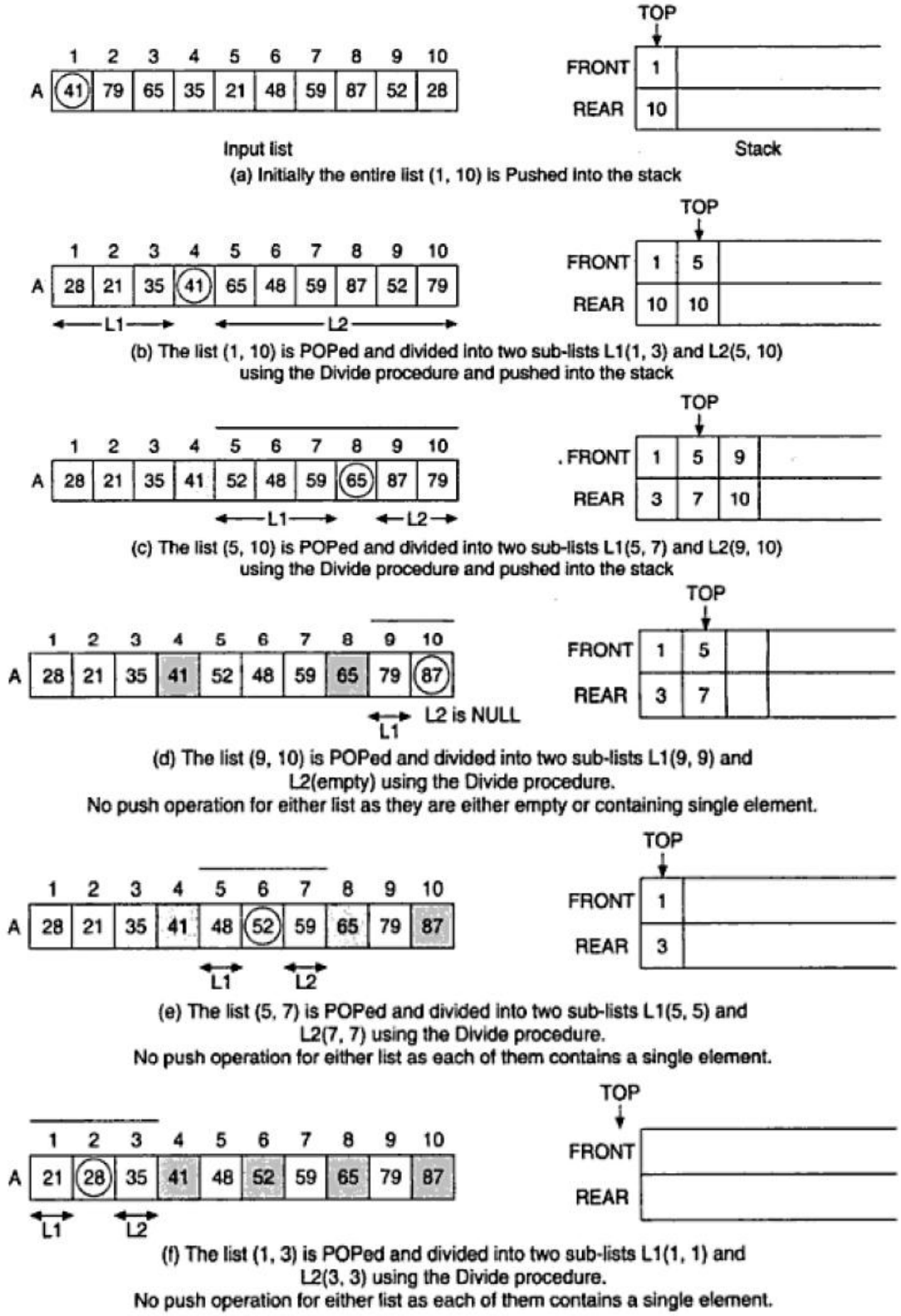


Figure 4.7 Illustration of recursive execution of quick sort using stacks.

6. Tower of Hanoi Problem

Suppose there are three pillars *A, Band* C. There are *N* discs of decreasing size so that no two discs are of the same size. Initially, all the discs are stacked on one pillar in their decreasing order of size. Let this pillar be *A.* The other two pillars are empty. The problem is to move all the discs from one pillar to another using the third pillar as an auxiliary so that

- Only one disc may be moved at a time.
- A disc may be moved from any pillar to another pillar.
- At no time can a larger disc be placed on a smaller disc.

Figure represents the initial and final stages of the tower of Hanoi problem for $N = 5$ discs.

Figure 4.8 represents the initial and final stages of the tower of Hanoi problem for $N = 5$ discs.

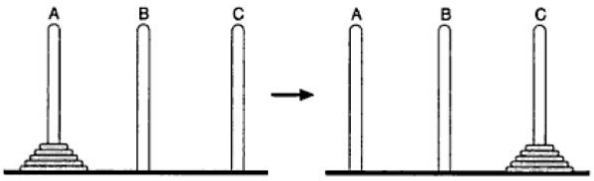


Figure 4.8 Tower of Hanoi problem with 5 discs.

The solution of this problem can be stated recursively as follows:
Move *N* discs from pillar *A* to *C* via the pillar *B* means

Moving the first $(N - 1)$ discs from pillar *A* to *B*.
Moving the disc from pillar *A* to *C*.
Moving all $(N - 1)$ discs from pillar *B* to *C*.

The solution of this problem can be stated recursively as follows:
Move *N* discs from pillar *A* to C via the pillar *B* means

Moving the first *(N* - 1) discs from pillar *A* to *B.*
Moving the disc from pillar *A* to C.
Moving all *(N* - 1) discs from pillar *B* to C.

The above solution can be described by writing a function, say *Move(N,* ORG, INT, DES), where *N* is the number of discs, ORG, INT and DES are origin (from pillar), intermediate (via pillar) and destination (to pillar), respectively. Thus, with this notation, *Move(5,* X, Z, Y) means moving 5 discs from pillar X to pillar *Y* taking the intermediate pillar as Z. With this definition in mind, the problem can be solved with recursion as follows:

Algorithm Move

Input: Number of discs in the tower of Hanoi *N*, specification of ORG as from the pillar and DES as to the pillar, and INT as the intermediate pillar.
Output: Steps of moves of *N* discs from pillar ORG to DES pillar.

Steps:

```
1. If N > 0 then                          // N = 0 is the termination condition
2.     Move(N – 1, ORG, DES, INT)
3.     ORG → DES (Move from ORG to DES)
4.     Move (N – 1, INT, ORG, DES)
5. EndIf
6. Stop
```

For $N = 3$, how will this recursion solve the problem as shown in Figure 4.9.

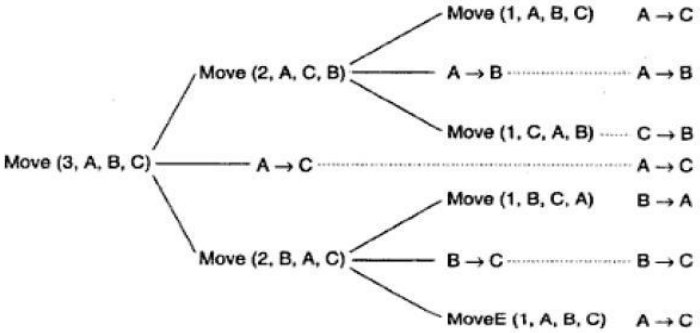


Figure 4.9 Tower of Hanoi (with $N = 3$) solution with recursion.

Now, let us implement this recursion using stacks. For this purpose, we have to assume the following stacks.

STN	is to store the number of discs
STA	is to store the pillar of origin
STB	is to store the intermediate pillar
STC	is to store the destination pillar
STADD	for the return address

PUSH (N, X, Y, Z, R) and POP (N, X, Y, Z, R) are the two stack operations over these stacks and are expressed as follows:

PUSH(N, X, Y, Z, R)	*POP(N, X, Y, Z, R)*
TOP = TOP + 1	N = STN[TOP]
STN[TOP] = N	X = STA[TOP]
STA[TOP] = X	Y = STB[TOP]
STB[TOP] = Y	Z = STC[TOP]
STC[TOP] = Z	R = STADD[TOP]
STADD[TOP] = R	TOP = TOP – 1

Algorithm HanoiTower

Input: N = number of discs, A = origin, B = intermediate and C = destination pillar.

Output: Steps of movements.

Data structure: Array representation of a stack.

Steps:

```
1.  top = NULL                                   // Initially all the stacks are empty
2.  org = 'A' , int = 'B', des = 'C', n = N      // Initialization of the parameters
3.  addr = Step 26                               // Return to the end step
4.  PUSH(n, org, int, des, add)                  // Push the initial value to the stacks
5.  If (STN[top] = 0) then                       // Terminal condition reached
6.      Go to STADD[top]
7.  Else                                         // Translation of move (N – 1, A, C, B)
8.      n = STN[top] – 1
9.      org = STA[top]
10.     int = STC[top]
11.     des = STB[top]
12.     addr = Step 15                           // After completing these moves return to Step 6
13.     Go to Step 4
14. EndIf
15. POP(n, org, ini, des, r)
16. Print "Move disc from:" org → des            // Move the nth disc from A to C
17. Do the following:
18.     n = STN[top] – 1                         // Translation of move (N – 1, B, A, C)
19.     org = STB[top]
20.     int = STA[top]
21.     des = STC[top]
22.     addr = Step 24
23.     Go to Step 4
24. POP(n, org, int, des, r)
25. Go to r                                      // Return address
26. Stop
```

To verify the algorithm *HanoiTower*, the reader can trace down the steps for N = 1, N = 2, N = 3, and N = 4 discs. It may be observed that the minimum number of moves required with N discs is 2^N -1

7. Activation Record Management

The *block structured* (also called *procedural)* programming language allows a user to define a number of variables having the same name or different names in various blocks. The scope of a variable is defined as the regions (that is the blocks) over which the variable is accessible. For example, consider the block structured program depicted in Figure.

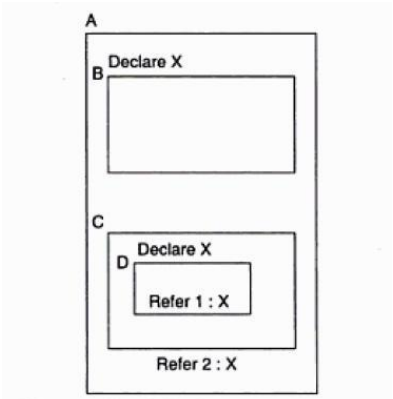


Figure 4.10 A block structured program.

Here, the variable X is declared in block *A* as well as in block C. Now references of the variable name at locations say, Refer 1 and Refer 2, are corresponding to which declaration? This can be decided by a *scope* rule. There are two scope rules: the *static scope rule* and the *dynamic scope rule*. The static scope rule defines the scope of a name in terms of the syntactic structure of a program. This rule is called 'static' because one can determine a variable's definition by looking at the program text alone. The static scope rule can be defined as below:

- The scope of a variable declared in a particular block consists of that block, exclusive of any block nested within it that declares the same identifier.
- If a variable is not declared within a block, then it obtains the declaration from the next outer block, if not there then the next outer block, and so on until a declaration is found. Such a rule is known as the 'most closely nested rule'.

Thus, with this rule, reference of a variable at Refer 2 (see Figure) will be resolved from the declaration in block *A,* whereas reference at Refer 1 will be resolved from the declaration in block C.

In the dynamic scope rule, on the other hand, reference of an identifier is resolved during the execution of the program and the same variable name may be defined at several points within the same program, that is, the variable name may change its definition as the execution proceeds. This is why the dynamic scope rule is also termed 'fluid binding'. This rule is stated as follows.

- The declaration of a variable is referred from the most recently occurring and still active definition of the name during the execution of the program.

For example, consider the program structure shown in below Figure. *P* is the main program. During its execution, at a certain point, the procedure 'call *A'* occurs. Here, procedure *A* is in the currently active block *P* . So, for any reference, X in *A* will be obtained from the declaration in *P.* For the other procedure 'ca1l2 C', as it itself has declaration for X (Declare 3) so any reference of X will be resolved from that only. Now, suppose *B* is on execution. *B* in turn calls Procedure *A* (as 'ca1l4 *A'):* Here, reference of X will be obtained from Declare 2 as *B* is the most recently occurring block. Thus, for a given reference of X in *A,* its declaration is once resolved from Declarel and another from Declare 2 in the same program.

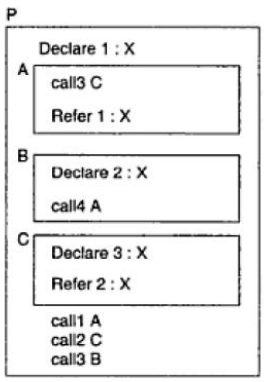


Figure 4.11 Dynamic scope rule.

Implementation of scope rules using stack

The static storage allocation is easy to implement and efficient from the execution point of view. Here, all variables which are required for a program are allocated during compile time. This is why static storage allocation is known as a compile time phenomenon. In this scheme, each subprogram/subroutine of a program is compiled separately and the space required for them is reserved till the completion of execution of the program. The space required for a program is, thus, just the sum of the space needed for the program and the subprograms-the space never changes as the program is running.

On the other hand, in dynamic storage allocation, the space for memory variables is allocated dynamically, that is, as per the current demand during the execution. When a subprogram is invoked, space for it is allocated and the space is returned when the subprogram completes its execution. Thus, the space required to run a program is not fixed as in static allocation; rather it varies as the program is executed

When a subprogram is invoked, a block of memory required for it is allotted and as soon as the execution is completed it is freed. A single chunk of storage, called an *activation record,* is used for this purpose. An activation record typically contains the following information:

- Storage for variables local to the subprograms.
- Declaration of the procedures and pointers (address of the starting location) to the definitions of procedures in the subprogram.
- The return address (after the end of subprogram, where the control should return).
- A pointer to the activation record of the location (the location of the block to which the subprogram belongs).

Thus, for the above-mentioned information, the structure of an activation record can be represented as shown in Figure.

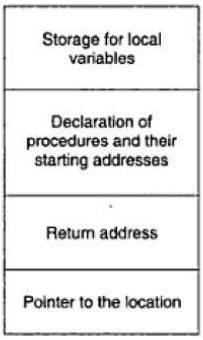
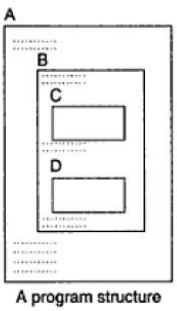


Figure 4.12 Structure of an activation record.

A stack for storing the "activation records needs to be maintained during the execution of a program (note that this stack should be with list structure because of the dynamic nature of the programs).

When the program control enters a new subprogram, its activation record is pushed onto the stack and when the subprogram finishes its execution, the control returns to an address which can be obtained from the field 'Return Address' of the activation record and this activation record is removed from the stack by updating the stack pointer. For example, for a program as shown in Figure 4.13, where *A* is the main program, it invokes *B, B* in turn invokes C and *D.* When a subprogram finishes its execution, then the next subprogram to be resumed can be decided by maintaining a stack. Next, we will see, how the scope of a memory variable can be resolved. To do this, let us first consider the pseudo code of a program, as listed in Figure.



A program structure

Stack	Action
A	Activation record of A is pushed
A B	A invokes B
A B C	B calls C
A B	C completes execution
A B D	B calls D
A B	D completes execution
A	Return to A

Run-time stack

Figure 4.13 Execution of program and its run-time stack.

```
01  Program MAIN
02    A, B, C : integer
03      Procedure Q
04      Begin
05        A = A+2
06        C = C+2
07      End
08      Procedure R
09        C : integer
10      Begin
11        C = 2;
12        call Q;
13        B = A + B
14      End
15      Procedure S
16        B, C : integer;
17        Procedure Q
18        Begin
19          A = A+1
20          C = C+1
21        End
22        Begin
23          B = 3
24          C = 1
25          call Q
26          call R
27        End
28        Begin
29          A = 1
30          B = 2
31          C = 3
32          call R
33          call S
34        End
```

Figure 4.14 A block structured program.

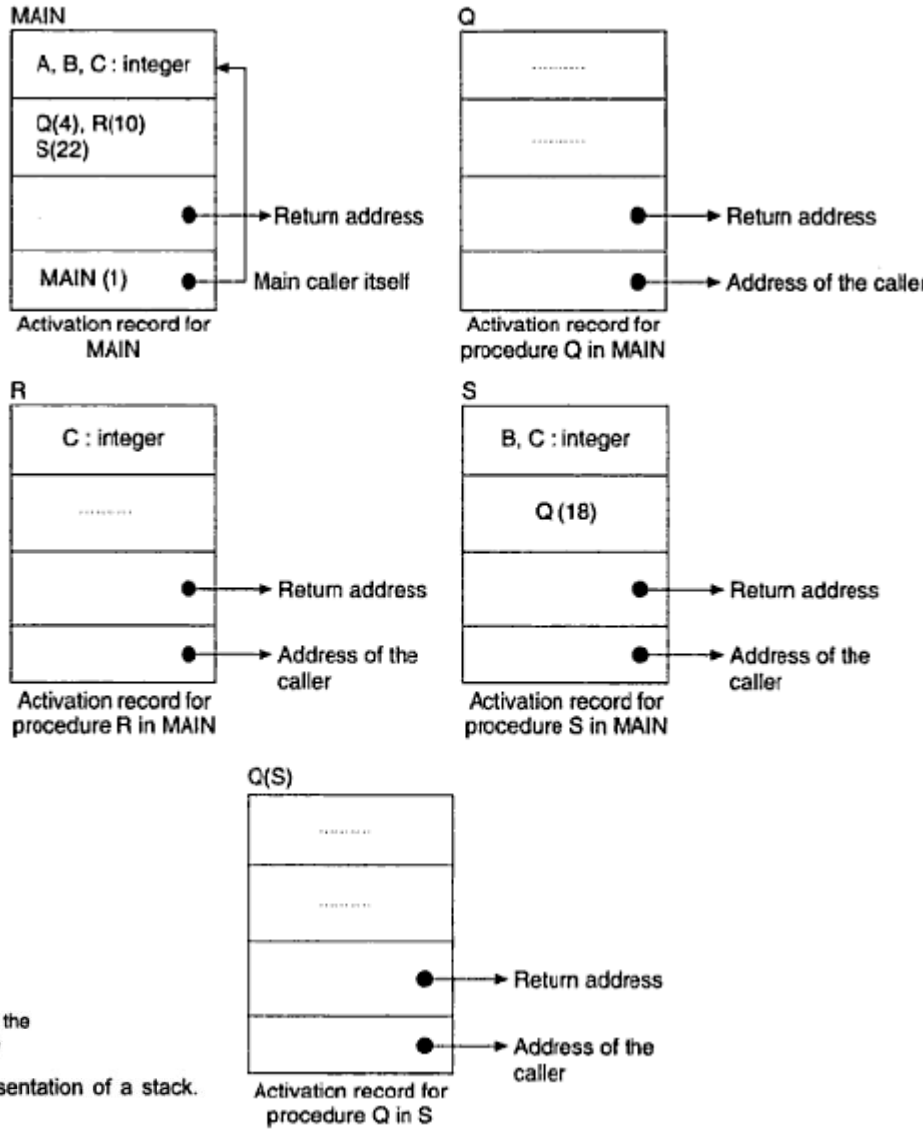


Figure 4.15 Activation records of various procedures.

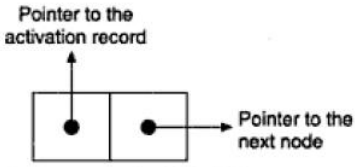
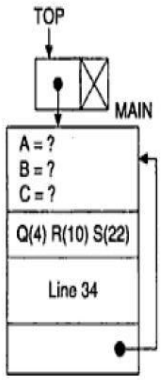


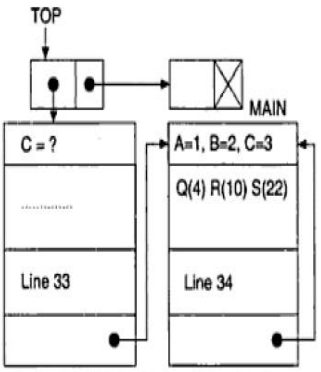
Figure 4.16 Node structure for linked list representation of a stack.

Implementation* of *static scope rule

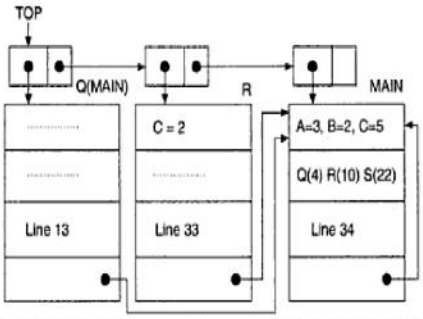
The reference of a variable will be resolved by consulting the current activation record, if it is not resolved here then it will be resolved from the activation records of its caller, and so on. Here the *caller* means a program/subprogram which calls the subprogram under discussion. The run-time stack view during the execution of the program MAIN (code) is illustrated in below Figure.



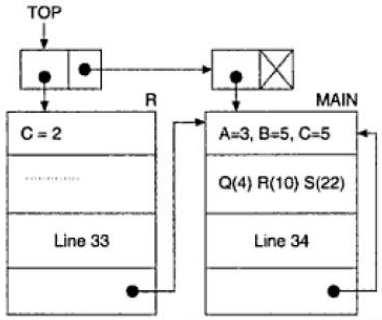
(a) MAIN begins its execution at line 28. Its activation record is PUSHed into the stack



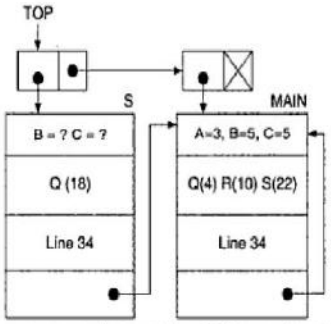
(b) Program control reaches the line 32, R is invoked and its activation record is PUSHed into the stack



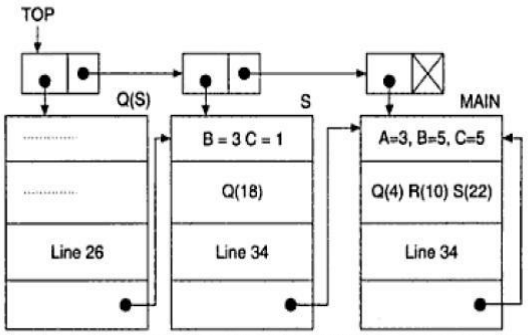
(c) Procedure R begins its execution at line 10 and invokes procedure Q at line 12. As Q is not in R, so from its pointer to the caller it is resolved and the corresponding activation record of Q is pushed. Execution of Q begins at line 4. References of A and C (at line 5 and 6) are resolved from MAIN, the outer block of Q.



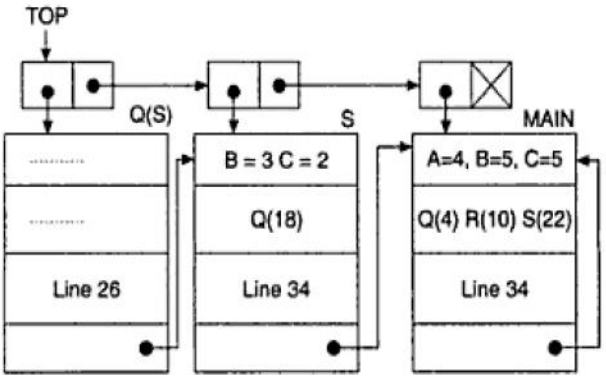
(d) When Q finishes its execution, control gets the return address from its activation record which is line 13. Activation record of Q is removed. Now, references to B and A at line 13 are obtained from their declaration in MAIN.



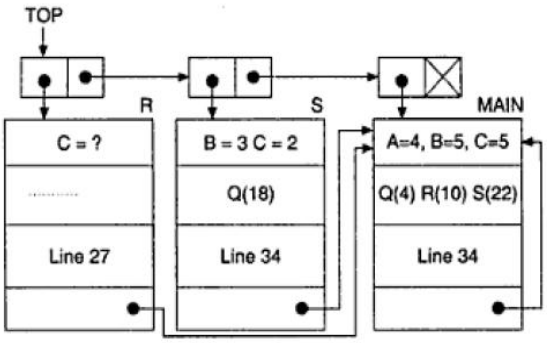
(e) When R completes its execution, control returns to line 33; the procedure S is invoked whose reference is resolved by the current activation record namely, MAIN and record of S is PUSHed into the stack



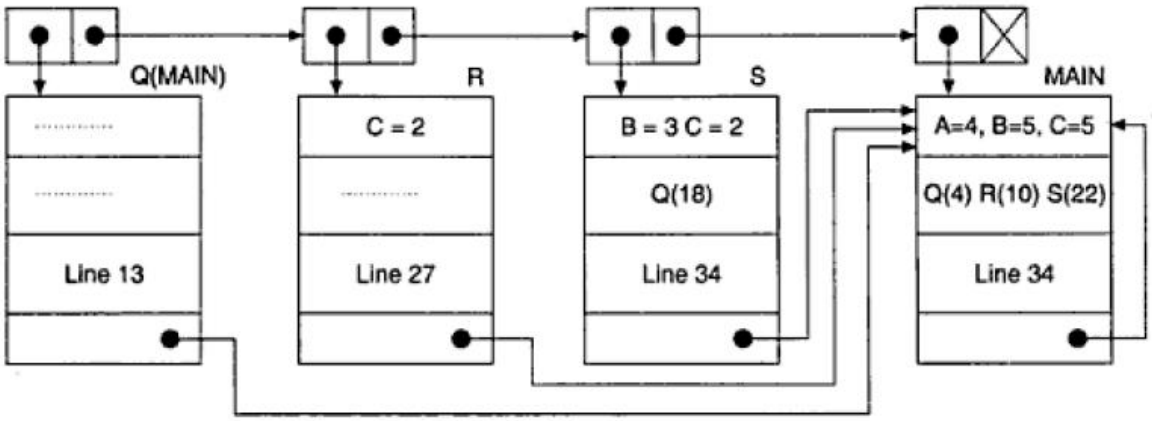
(f) Procedure S begins its execution at line 22 and when control reaches the line 25 it involves the procedure Q. The reference of Q is resolved from the current activation record, that is, of S and then the activation record of Q is then PUSHed into the stack.



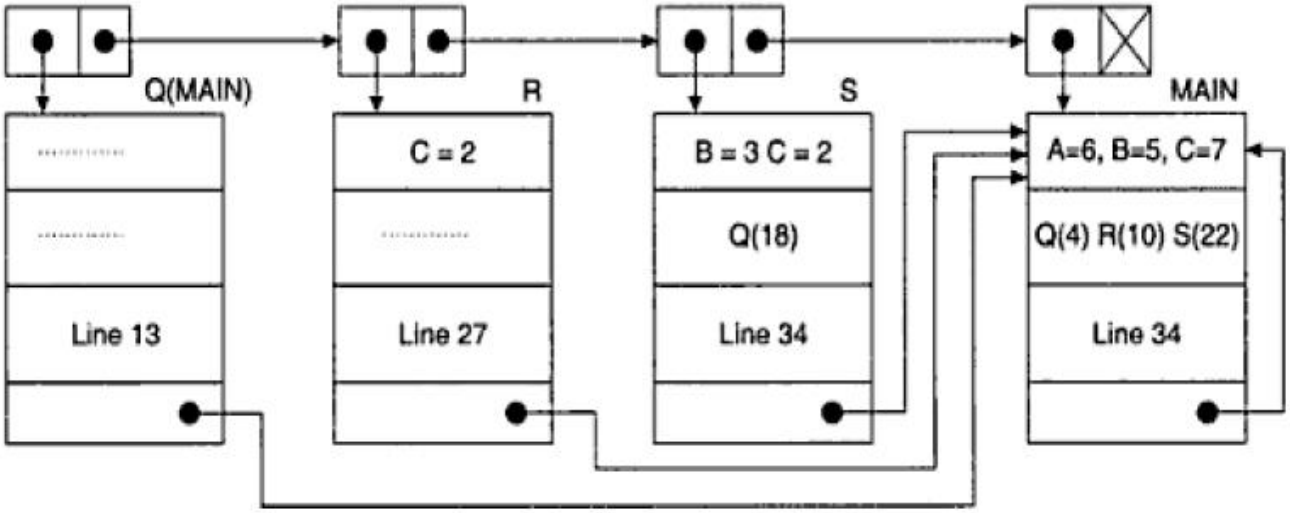
(g) Execution of Q is started and references of A and C at lines 19 and 20 are resolved from S and MAIN, respectively.



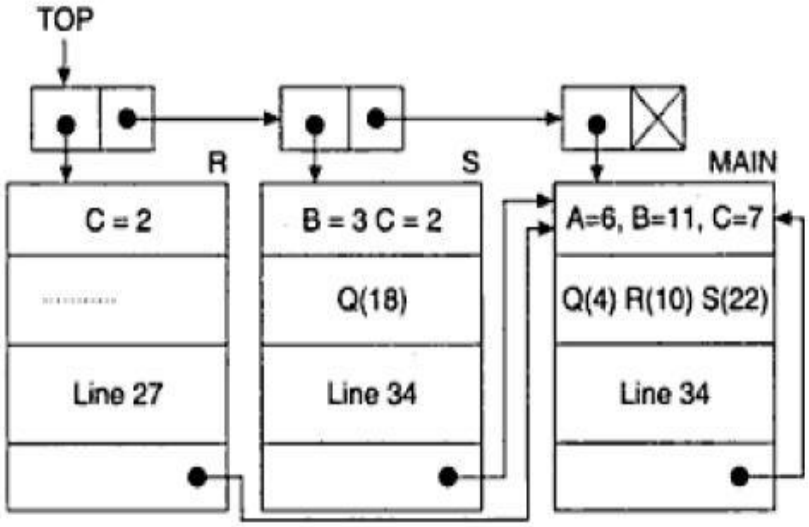
(h) When Q finishes its execution, control returns to line 26, its activation is then POPed and procedure R is invoked. This R is resolved from the activation record of MAIN.



(i) R begins its execution at line 10. Reference of C is resolved from the activation record of R. It again invokes Q for its activation record, R is searched, which in turn searches the activation record of MAIN; so the reference of Q is resolved from MAIN.



(j) Q starts execution at line 5; references of A and C at lines 5 and 6, respectively, are resolved from MAIN.

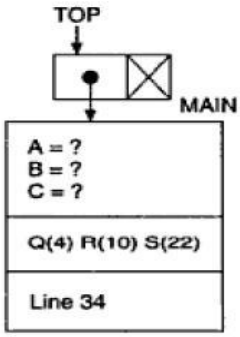


(k) When Q finishes its execution, control returns to line 13, Q's activation record is returned; current activation record is R. References of A and B (at line 13) are resolved from MAIN. When R finishes its execution at line 14, control gets the return address the line 27, R is removed; control next returns to line 27. Line 27 is the end of S, so S is completed; control returns to line 34. Line 34 is the end of the program MAIN. The execution of the program reaches its end.

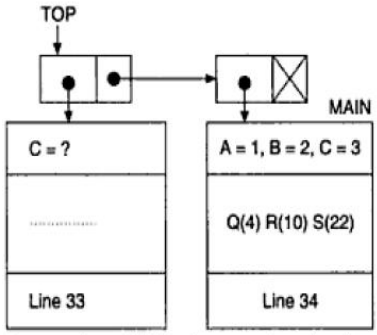
Implementation of dynamic scope rule

Implementation of the dynamic scope-rule is much easier than the implementation of the static scope rule. For the dynamic scope rule, the structure of an activation record is the same as for the static scope rule except that here it is not required to maintain a pointer field to store the address of the locator.

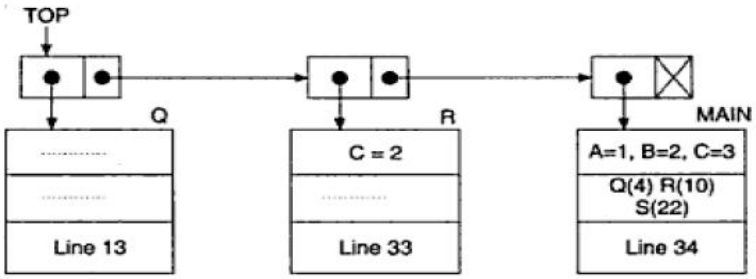
On reference of a variable, its declaration will be searched first from the current activation record; if not found then the next activation record on the stack and so on till the declaration is found or all the records on the stack are searched. As in the static scope rule, here also the execution of a subprogram starts with pushing its activation record onto the stack and, when the execution is finished, the activation record is simply wiped out (that is popped). For the program(code) structure as mentioned, its execution using the dynamic scope rule is illustrated in Figure.



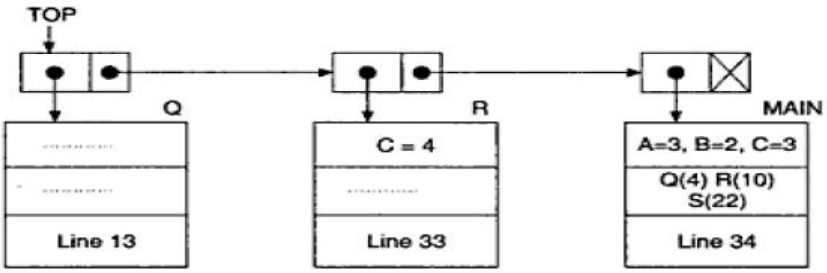
(a) Main begins its execution at line 28. Its activation record is PUSHed into the stack.



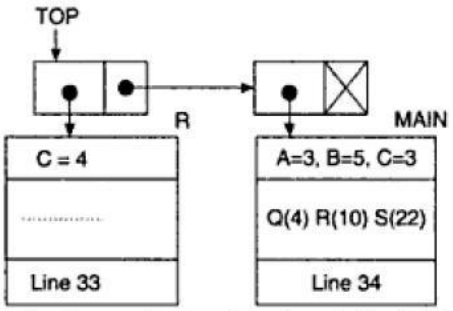
(b) During the execution of MAIN when control reaches line 32 and call of R occurs, the activation record of R is PUSHed into the stack.



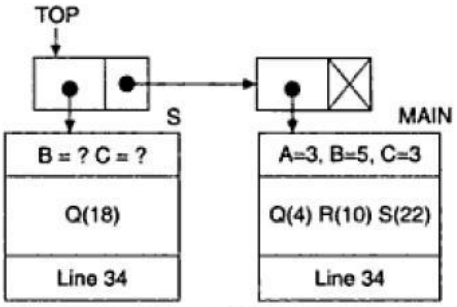
(c) Control reaches line 12, the execution of Q is initiated. This Q will be referred from the first activation record present in the stack, that is, from MAIN.



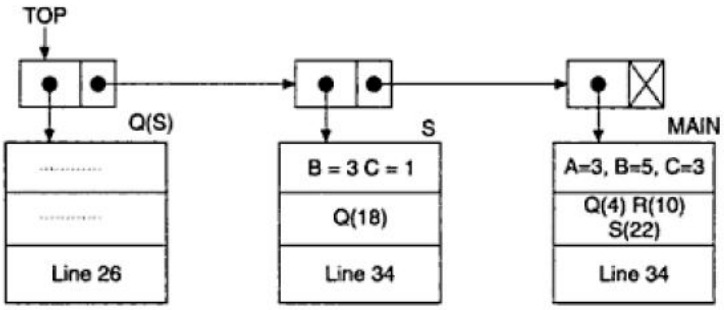
(d) When Q begins its execution at line 4 the reference of A is from MAIN and that of C is from R.



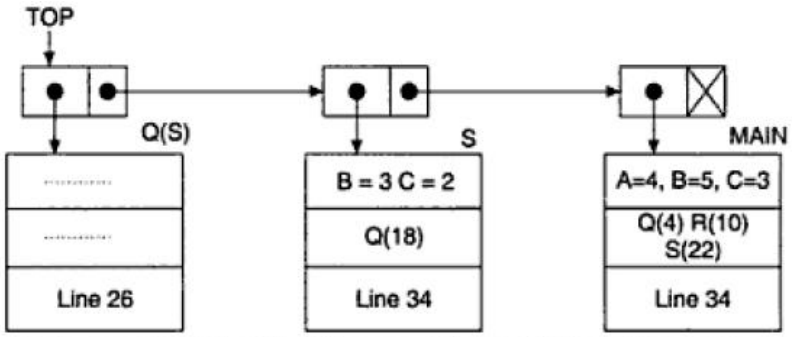
(e) Q completes its execution, control returns to line 13; A and B are referred from the activation record of MAIN. B is updated.



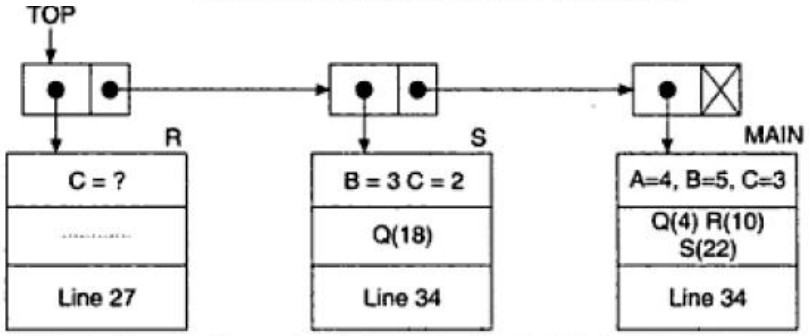
(f) When R finishes its execution, control returns to line 33; call of S occurs and the activation record of S is PUSHed into the stack.



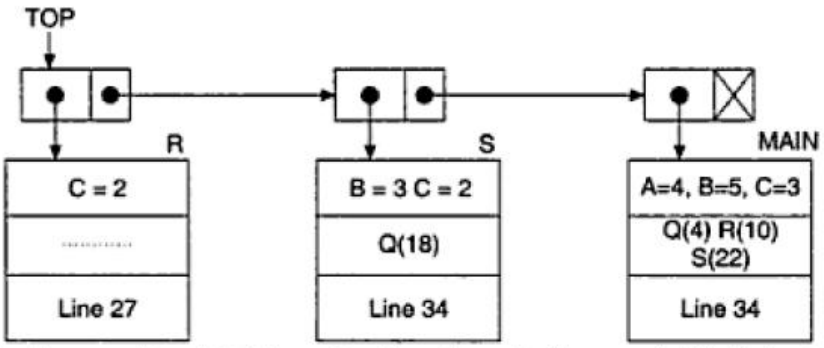
(g) Execution of S begins at line 22 (referred from MAIN) and references of B and C are from the activation record of S. When control reaches line 25, invocation of Q occurs. Reference of Q is resolved from the first occurrence, that is, from the activation of S in stack.



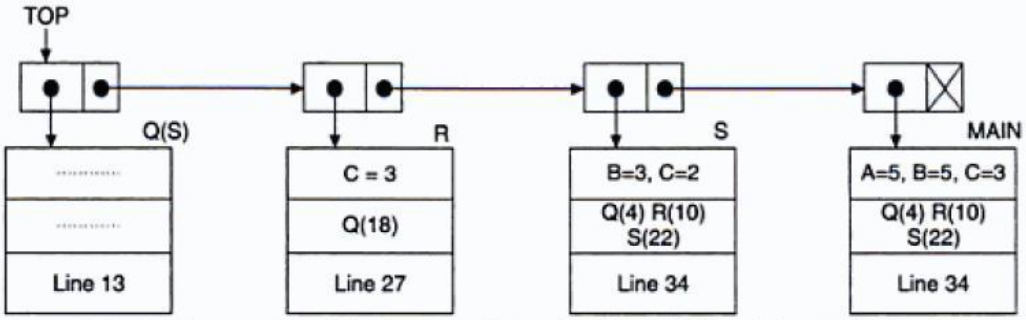
(h) Q begins its execution at line 18. References of A and C (at lines 19 and 20, respectively) will be resolved from MAIN and S, respectively.



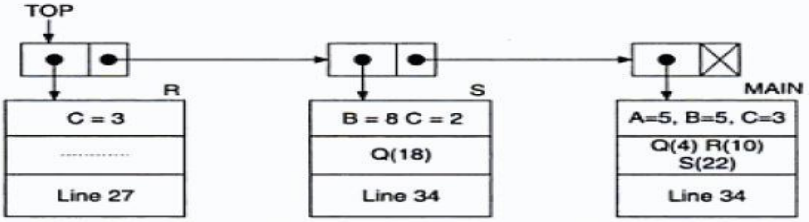
(i) Q finishes its execution and control reaches line 26, R is initiated. Its activation record is PUSHed into the stack, control jumps to line 10.



(j) During the execution of R, C is resolved from the activation record of R. Next, when Q is invoked, reference of Q is resolved from the first occurrence, that is, from S. The Q is at line 18.



(k) During the execution of Q, reference of A is referred from MAIN and that of C is from R.



(l) After Q finishes its execution, control returns to line 13. For the reference of B and A at line 13, B is resolved for S and A is from MAIN. Later, when the execution of R is completed, control returns to line 27, which indicates that the execution of S is finished, then control returns to line 34 which is the end of the program MAIN.

Figure 4.18 (a)–(l) Execution of MAIN using the dynamic scope rule.