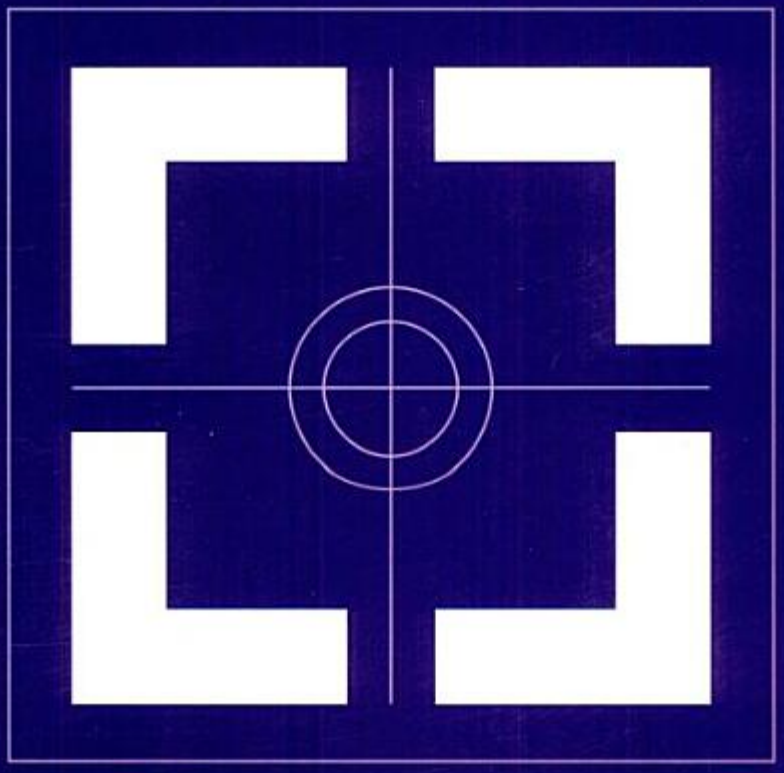
Eastern Economy Edition

SECOND EDITION

Classic Data Structures



Debasis Samanta

Rs. 395.00

CLASSIC DATA STRUCTURES, 2nd ed. (With CD-ROM)
Debasis Samanta

Warning and Disclaimer
The author and the publisher are not responsible for any loss or damage arising from the use of CD-ROM included with this book. The CD-ROM included with this book has no commercial value and cannot be sold separately.

ISBN-978-81-203-3731-2

Seventeenth Printing (Second Edition) ... ... July, 2009

Published by Asoke K. Ghosh, PHI Learning Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Rajkamal Electric Press, Plot No. 2, Phase-IV, Kundli, Haryana.

Contents

With the built-in data types, programming languages provide users with a lot of advantages of processing of various types of data. For example, if a user declares a variable of type Real (say), then several things are automatically implied, such as how to store a value for that variable, what are the different operations possible on that type of data, what amount of memory is required to store, etc. All these things are taken care of by the compiler or the run-time system manager.

Abstract data type

When an application requires a special kind of data which is not available as a built-in data type, then it is the programmer's responsibility to implement his own kind of data. Here, the programmer has to specify how to store a value for that data, what are the operations that can meaningfully manipulate variables of that kind of data, amount of memory required to store a variable. The programmer has to decide all these things and accordingly implement them. Programmers' own data type is termed *abstract data type*. The abstract data type is also alternatively termed *user-defined data type*. For example, suppose we want to process dates of the form dd/mm/yy. For this, no built-in data type is known in C, FORTRAN, and Pascal. If a programmer wants to process dates, then an abstract data type, say Date, has to be devised and various operations such as adding a few days to a date to obtain another date, finding the days between two dates, obtaining a day for a given date, etc. have to be defined accordingly. Besides these, programmers should decide how to store the data, what amount of memory will be needed to represent a value of Date, etc. An abstract data type, in fact, can be built with the help of built-in data types and other abstract data type(s) already built by the programmer. Some programming languages provide a facility to build abstract data types easily. For example, using struct/class in C/C++, and using record in Pascal, programmers can define their own data types.

1.2 CONCEPT OF DATA STRUCTURES

A digital computer can manipulate only primitive data, that is, data in terms of 0's and 1's. Manipulation of primitive data is inherent within the computer and does not require any extra effort on the part of the user. But in our real-life applications, various kinds of data other than the primitive data are involved. Manipulation of real-life data (also termed user data) requires the following essential tasks:

1. *Storage representation of user data:* User data should be stored in such a way that the computer can understand it.
2. *Retrieval of stored data:* Data stored in a computer should be retrieved in such a way that the user can understand it.
3. *Transformation of user data:* Various operations which require to be performed on user data so that it can be transformed from one form to another.

The basic theory of computer science deals with the manipulation of various kinds of data, wherefrom the concept of data structures comes in. In fact, data structures constitute the fundamentals of computer science. For a given kind of user data, its structure implies the following:

1. Domain ($\mathcal{D}$): This is the range of values that the data may have. This domain is also termed data object.
2. Function ($\mathcal{F}$): This is the set of operations which may legally be applied to elements of the data object. This implies that for a data structure, we must specify the set of operations.
3. Axioms ($\mathcal{A}$): This is the set of rules with which the different operations belonging to $\mathcal{F}$ can actually be implemented.

Now we can define the term data structure.

A *data structure* D is a triplet, that is, D = ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) where $\mathcal{D}$ is a set of data object, $\mathcal{F}$ is a set of functions and $\mathcal{A}$ is a set of rules to implement the functions. Let us consider an example.

We know that for the integer data type (int) in the C programming language the structure includes the following types:

$\mathcal{D}$ = (0, ±1, ±2, ±3, ...)
$\mathcal{F}$ = (+, –, *, /, %)
$\mathcal{A}$ = (A set of binary arithmetics to perform addition, subtraction, division, multiplication, and modulo operations.)

It can be easily recognized that the triplet ($\mathcal{D}$, $\mathcal{F}$, $\mathcal{A}$) is nothing but an abstract data type. Also, the elements in set $\mathcal{D}$ are not necessarily from primitive data; it may contain elements from some other abstract data types. Alternatively, an implementation of a data structure D is a mapping from $\mathcal{D}$ to a set of other data structures D_i, i = 1, 2, ..., n, for some n. More precisely, this mapping specifies how every object of $\mathcal{D}$ is to be represented by the objects of D_i, i = 1, 2, ..., n. Every function of D must be written using the function of the implementing data structures D_i, i = 1, 2, ..., n. The fact is that each of the implementing data structures is either a primitive data type or an abstract data type. We can conclude the discussion with another example.

Suppose, we want to implement a data type, namely `Complex` as an abstract data type. Any variable of the complex data type has two parts: a real part and an imaginary part. In our usual notation, if z is a complex number, then $z = x + iy$, where x and y are the real and imaginary parts, respectively. Both x and y are of the `Real` data type which is another abstract data type (available as a built-in data type). So, the abstract data type `Complex` can be defined using the data structure `Real` as

```
Complex z {
  x : Real
  y : Real
}
```

Now the set $\mathcal{D}$ of `Complex` can be realized from the domain of x and y which is `Real` in this case. Let us specify the set of operations for the `Complex` data type, which are stated as $\mathcal{F}$:

$$\mathcal{F} = (\oplus, -, \otimes, \div, \nabla, \|)$$

Assume that $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ are two data of the `Complex` data type. Then we can define the rules for implementing the operations in $\mathcal{F}$, thus giving axioms $\mathcal{A}$. In the current example, for the `Complex` data type

$\mathcal{A}$ = {

$z = z_1 \oplus z_2 = (x_1 + x_2) + i(y_1 + y_2)$ (complex addition)

$z = z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$ (complex subtraction)

$z = z_1 \otimes z_2 = (x_1 \times x_2 - y_1 \times y_2) + i(x_1 \times y_2 + x_2 \times y_1)$ (complex multiplication)

$z = z_1 \div z_2$

$$= \frac{x_1 \times x_2 + y_1 \times y_2}{x_2^{\,2} + y_2^{\,2}} + i\frac{x_2 \times y_1 - x_1 \times y_2}{x_2^{\,2} + y_2^{\,2}}$$ (complex division)

$$z = \nabla z_1 = \frac{x_1}{x_1^{\,2} + y_1^{\,2}} - i\frac{y_1}{x_1^{\,2} + y_1^{\,2}}$$ (complex conjugate)

$$z = |z_1| = \sqrt{x_1^{\,2} + y_1^{\,2}}$$ (complex magnitude)

}

Note that how different operations of the `Complex` data type can be implemented using the operations +, –, ×, /, of the implementing data structure, namely `Real`.

Assignment 1.1

Implement *Date* as an abstract data type which consists of *dd/mm/yy*, where *dd* varies from 1 to 31, *mm* varies from 1 to 12 and *yy* is any integer with four digits.

Specify $\mathcal{F}$ consisting of all possible operations on variables of type *Date* and then define $\mathcal{A}$ to implement all the operations in $\mathcal{F}$. Assume the implementing data structure(s) which is/are necessary.

1.3 OVERVIEW OF DATA STRUCTURES

In computer science, several data structures are known depending on the areas of application. Out of them, a few data structures are frequently used in almost all application areas and with the help of which almost all complex data structures can be constructed. These data structures are known as *fundamental data structures* or *classic data structures*. Figure 1.2 gives a classification of all classic data structures.

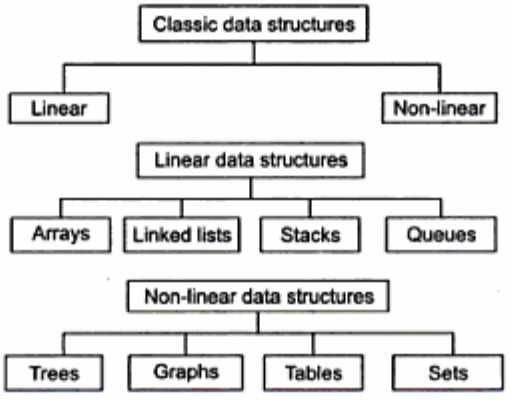


Figure 1.2 Classification of classic data structures.

4	15	14	1
9	6	7	12
5	10	11	8
16	3	2	13

Figure 2.15 A magic square with SUM = 34.

(a) Write a program to read a set of integers for a square matrix and then decide whether the matrix represents a magic square or not.

(b) Write a game program as follows:

(i) Read the size of the square matrix, $N \times N$.

(ii) Display a square matrix (now it is blank) of $N \times N$.

(iii) Allow the player to insert data into the matrix as displayed (you should give a chance to the user to confirm the entry and to alter the previous entries, if desired).

(iv) After the completion of all the entries from the player, count the score as follows:
Score = 0 (zero) if it is not a magic square. Otherwise score = $T + P*100$, where T is the time required in seconds and P is the number of alteration of entries.

The player's performance will be judged by the minimum score achieved other than zero.

REFERENCES

Gotlieb, C.C. and L.R. Gotlieb, *Data Types and Structures,* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

Horowitz, Ellis, and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland, 1985.

Kruse, Robert L., Bruce P. Leung and L. Clovis Tondo, *Data Structures and Program Design in C,* Prentice Hall of India, New Delhi, 1995.

Tremblay, Jean Paul, and Paul G. Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1987.

3

Linked Lists

An array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, a block of memory that is required for the array should be allocated before hand. Once the memory is allocated, it cannot be extended any more. This is why the array is known as a *static data structure*. In contrast to this, the linked list is called a *dynamic data structure* where the amount of memory required can be varied during its use. In the linked list, the adjacency between the elements is maintained by means of *links* or *pointers*. A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained. An element in a linked list is a specially termed *node*, which can be viewed as shown in Figure 3.1. A node consists of two fields: DATA (to store the actual information) and LINK (to point to the next node).

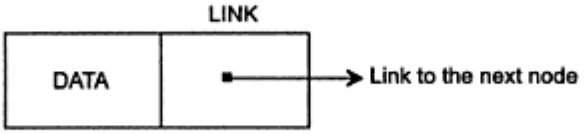


Figure 3.1 Node: an element in a linked list.

3.1 DEFINITION

A *linked list* is an ordered collection of finite, homogeneous data elements called *nodes* where the linear order is maintained by means of links or pointers.

Depending on the requirements the pointers are maintained, and accordingly the linked list can be classified into three major groups: single linked list, circular linked list, and double linked list.

3.2 SINGLE LINKED LIST

In a single linked list each node contains only one link which points to the subsequent node in the list. Figure 3.2 shows a linked list with six nodes.

Here, N1, N2, . . ., N6 are the constituent nodes in the list. HEADER is an empty node (having data content NULL) and only used to store a pointer to the first node N1. Thus, if one knows the address of the HEADER node from the link field of this node, the next node can be traced, and so on. This means that starting from the first node one can reach to the last node whose link field does not contain any address but has a null value. Note that in a single linked list one can move from left to right only; this is why a single linked list is also called *one way* list.

3.2.1 Representation of a Linked List in Memory

There are two ways to represent a linked list in memory:

1. Static representation using array
2. Dynamic representation using free pool of storage

Static representation

In static representation of a single linked list, two arrays are maintained: one array for data and the other for links. The static representation of the linked list in Figure 3.2 is shown in Figure 3.3.

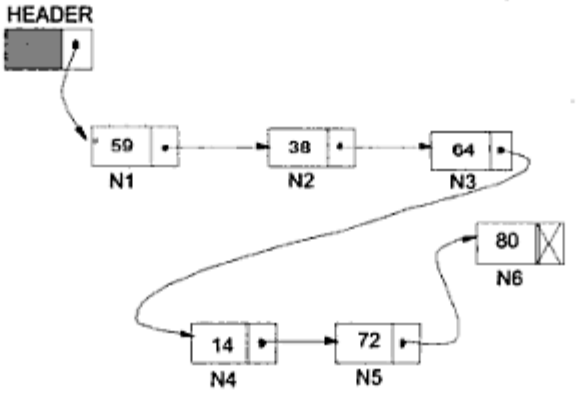


Figure 3.2 A single linked list with six nodes.

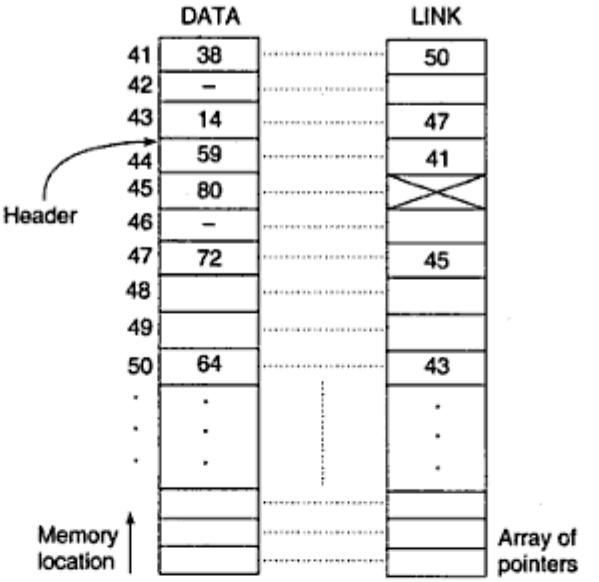


Figure 3.3 Static representation using arrays of the single linked list of Figure 3.20.

Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless this contradicts the idea of the linked list (that is non-contagious location of elements). But in some programming languages, for example, ALGOL, FORTRAN, BASIC, etc. such a representation is the only representation to manage a linked list.

Dynamic representation

The efficient way of representing a linked list is using the free pool of storage. In this method, there is a *memory bank* (which is nothing but a collection of free memory spaces) and a *memory manager* (a program, in fact). During the creation of a linked list, whenever a node is required the request is placed to the memory manager; the memory manager will then search the memory bank for the block requested and, if found, grants the desired block to the caller. Again, there is also another program called the *garbage collector*; it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is basically a list of memory spaces which is available to a programmer. Such a memory management is known as *dynamic* memory management. The dynamic representation of linked list uses the dynamic memory management policy.

The mechanism of dynamic representation of single linked list is illustrated in Figures 3.4(a) and 3.4(b). A list of available memory spaces is there whose pointer is stored in AVAIL. For a request of a node, the list AVAIL is searched for the block of right size. If AVAIL is null or if the block of desired size is not found, the memory manager will return a message accordingly. Suppose the block is found and let it be XY. Then the memory manager will return the pointer of XY to the caller in a temporary buffer, say NEW. The newly availed node XY then can be inserted at any position in the linked list by changing the pointers of the concerned nodes. In Figure 3.4(a), the node XY is inserted at the end and change of pointers is shown by the dotted arrows. Figure 3.4(b) explains the mechanism of how a node can be returned from a linked list to the memory bank.

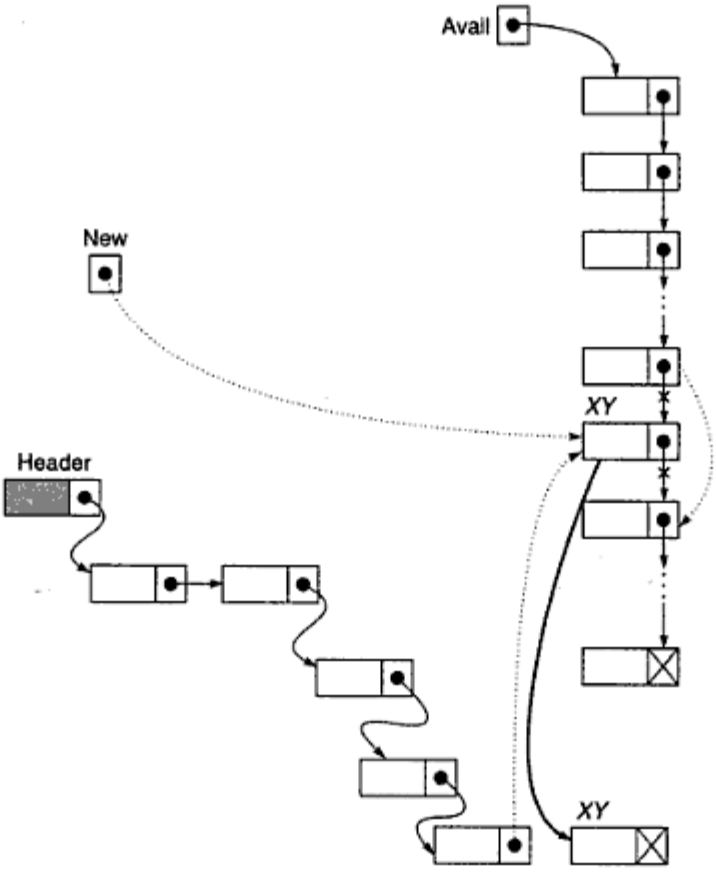


Figure 3.4(a) Allocation of a node from memory bank to a linked list.

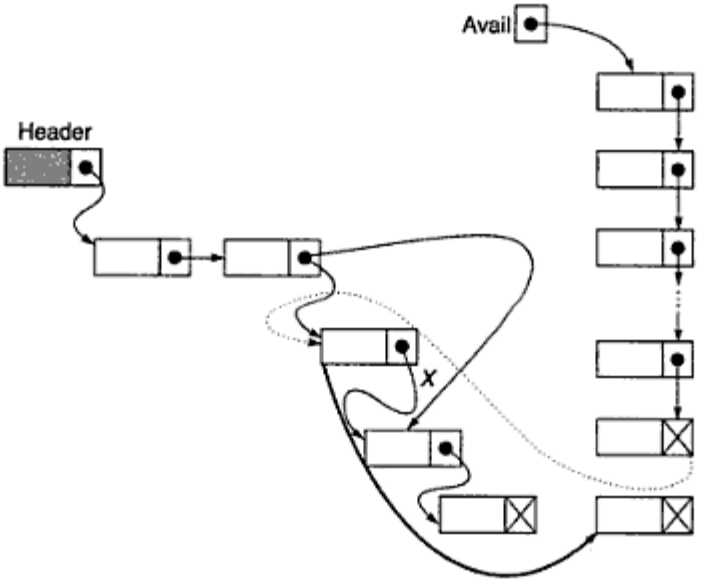


Figure 3.4(b) Returning a node from a linked list to memory bank.

the free list is saved following an allocation and is used to begin search for the subsequent request. The idea of this strategy is to reduce the search by avoiding examination of smaller blocks that, in the long run, tend to be created at the beginning of the free list as it happens in the case of first-fit allocation.

Comparison among the strategies

All these strategies are for variable sized blocks. With these strategies, we encounter the problem of fragmentation, a problem which never occurs in the case of fixed size requests. There are two types of memory fragmentation: internal fragmentation and external fragmentation.

Internal fragmentation

External fragmentation makes the memory management system inefficient as it results in a large number of small blocks and hence a long list for searching. In order to get rid of this problem, we can refuse to split a block into small pieces, one of which might be very small; instead what we can do is that allocate the entire block which is larger than the requested size. This excess space is termed 'hole', which is 'allocated but unused'. This is wasted in the sense that this hole cannot be allocated any more. This phenomenon of partitioning the total unused storage into available blocks and allocating these blocks with some portion of these blocks remaining unused but not available, is called *internal fragmentation.*

External fragmentation

If a large number of storage blocks are requested and allocated, the linked list of available storages can be reasonably lengthy and the average block size becomes small and there remain probably very few blocks which are large. If a request for a large block is received, it may have to be denied because there is no single block on the free list that is big enough, even though the total amount of free storage (in the small blocks) may be much greater than the requested amount. This phenomenon of decomposing the total available storage into a large number of relatively small blocks is called *external fragmentation.*

Fragmentation is a major problem in any dynamic memory management system and should be handled carefully. All the above-mentioned strategies suffer from the fragmentation problem.

First-fit and best-fit are among the most popular strategies in a dynamic memory management system. As indicated earlier, first-fit is generally faster because it terminates as soon as a free block, large enough to house a new partition, is found. The best-fit method, on the other hand, does not use the first suitable block found, but instead continues the search until the smallest suitable block has been found. This, although, tends to save larger blocks at times, which may be needed later to service large requests. Thus, the best-fit method requires a search of the entire list, while the average length of search for the first-fit would be half of the best-fit or even less. Furthermore, the best-fit method has the unfortunate tendency to produce a large number of very small free blocks, and these are often unusable by almost all requests. In principle, the first-fit is faster, but it does not minimize wastage of memory for a given allocation. Best-fit is slower and it tends to produce small leftover free blocks that may be too small for subsequent allocation. However, when processing a series of requests starting with an initially free memory, neither strategy has been shown to be superior to the other in terms of wasted memory.

Worst-fit prevents what the best-fit does; it reduces the rate of production of small blocks. However, some simulation studies indicate that the worst-fit allocation is not very effective in reducing wasted memory in processing a series of requests.

Next-fit as discussed, is a modification of the first-fit strategy. In general, the next-fit does not outperform the first-fit in reducing the amount of wasted memory.

So far as processing speed is concerned, a rough comparison can be made as per their order of superiority, which is specified below:

Next-fit > First-fit > Best-fit, Worst-fit

The boundary tag system uses a slight variation of the above-mentioned strategies. It uses the next-fit storage allocation strategy and does not consider a block for allocation which if allocated leaves a small block of size < ε. That is, our pool of free storage will not contain any free block of size < ε, where ε is chosen by a statistic based on the nature of requests. As per the next-fit strategy, after allocating a block, search for the subsequent request will continue from the next block of these allocated blocks, and let AVAIL store the address of such a next block.

Algorithm GetNodeNextFit_BTS

Input: N, the size of the block requested, e the minimum size of a block for fragmentation, and AVAIL being the pointer to the block on the list from where the search for the desired block is to be continued.

Output: ptr, the pointer of the block of required size if available else message.

Data structure: Linked structure for the boundary tag system.

Steps:

```
 1. ptr = AVAIL                          // AVAIL is the current pointer to a block in the list
 2. flag = 0                             // This flag is used to indicate the continuity of search
 3. While (flag = 0) do
 4.     If (ptr→SIZE > N) then                          // If large enough block is found
 5.         diff = ptr→SIZE – N          // Difference between block size and requirement
 6.         If (diff < e) then           // Fragmentation test suggests for whole block allocation
 7.             ptr1 = ptr→LLINK
 8.             ptr2 = ptr→RLINK                        // Delete the block from the list
 9.             ptr1→RLINK = ptr2
10.             ptr2→LLINK = ptr1
11.             ptr→TAG = 1
12.             (ptr+ptr→SIZE – 1) →TAG=1                // Block is allocated
13.             AVAIL = ptr2                            // Block for next search
14.             flag = 1                                // Block is found
15.             Return(ptr)                             // Return ptr to the caller
16.         Else                                // Allocate the lower words in the block
17.             ptr→SIZE = diff
```

```
18.            (ptr + diff – 1) →UPLINK = ptr      // Make the upper words in the block free
19.            (ptr + diff – 1) →TAG = 0
20.            AVAIL = ptr
21.             ptr = ptr + diff                   // Allocate the last words of the block and
22.            ptr→SIZE = N                                  // set its fields as allocated
23.            ptr→TAG = 1
24.            (ptr + N – 1) →TAG = 1
25.            flag = 1                                        // Block is allocated
26.         EndIf
28.      Else
29.         If (ptr→RLINK = AVAIL) then
30.            Print "Block size is insufficient or memory underflow"
31.            flag = 1                                                      // Exit
32.         Else
33.            ptr = ptr→RLINK                              // Move to the next block
34.         EndIf
35.      EndIf
36.   EndWhile
37.   Stop
```

It may be noticed that whenever a block is allocated, it is only required to change the field TAG and SIZE; the other fields like LLINK, RLINK and UPLINK are not at all required to be updated. This is because for an allocated block these fields are useless information.

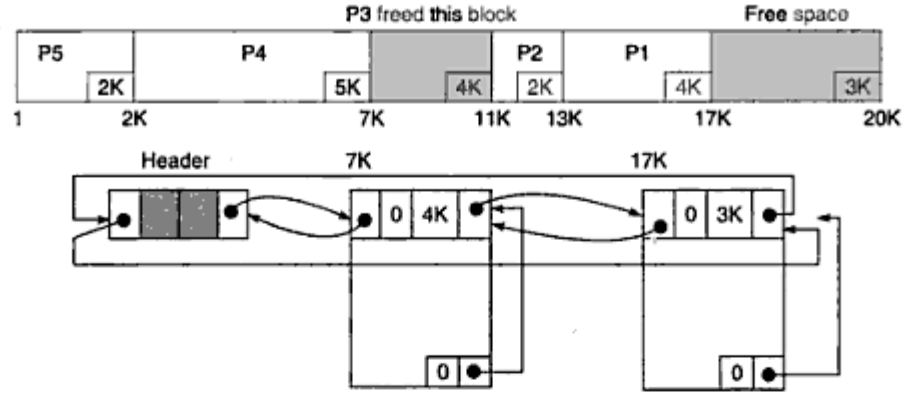
3.9 DEALLOCATION STRATEGY

When a block is no more in use, it is required to be returned to the pool of free storages so that other programs can use this storage space. The boundary tag system not only inserts this block to the list of free blocks, but also combines this newly inserted block with its adjacent block(s). The various cases of this type of combining during the deallocation are illustrated in Figure 3.26.

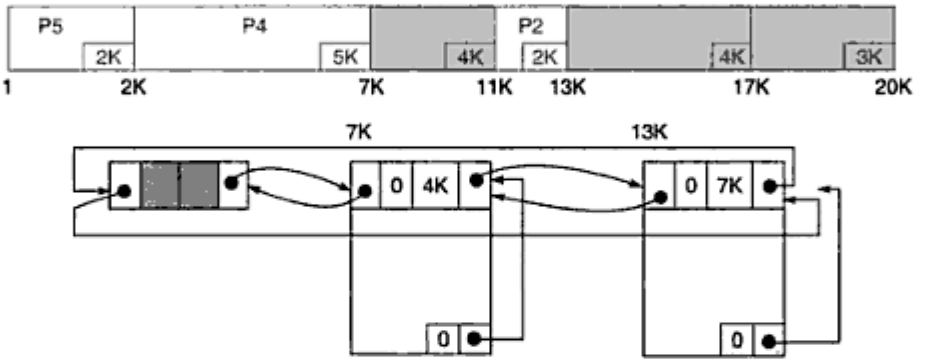
Here, we assume a total 20K memory space and 5 programs, viz. P1, P2, P3, P4 and P5 which are using it, occupying memories of sizes 4K, 2K, 4K, 5K, and 2K, respectively. Suppose, P3 releases its space of 4K and it is inserted into the list of free storage. In this situation, the available list of storage contains two non-adjacent free blocks of size 4K and 3K, as shown in Figure 3.26(a).

Now consider the case when P1 releases the block allocated to it. This block has a right adjacent block and hence two blocks will be combined into one. Figure 3.26(b) depicts it.

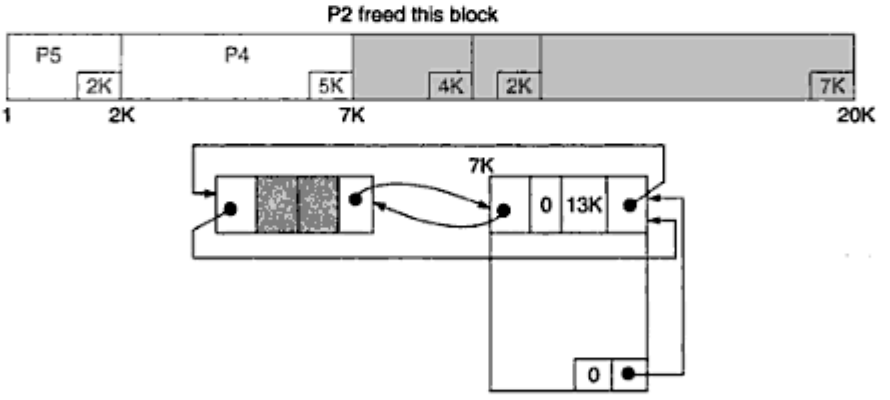
Next, let us consider the case, when P2 releases the block of size 2K. As there are two adjacent free blocks of this block, it is possible to fuse these three blocks into one. Figure 3.26(c) illustrates this case.



(a) Map for total 20K memory space and its corresponding list of free blocks



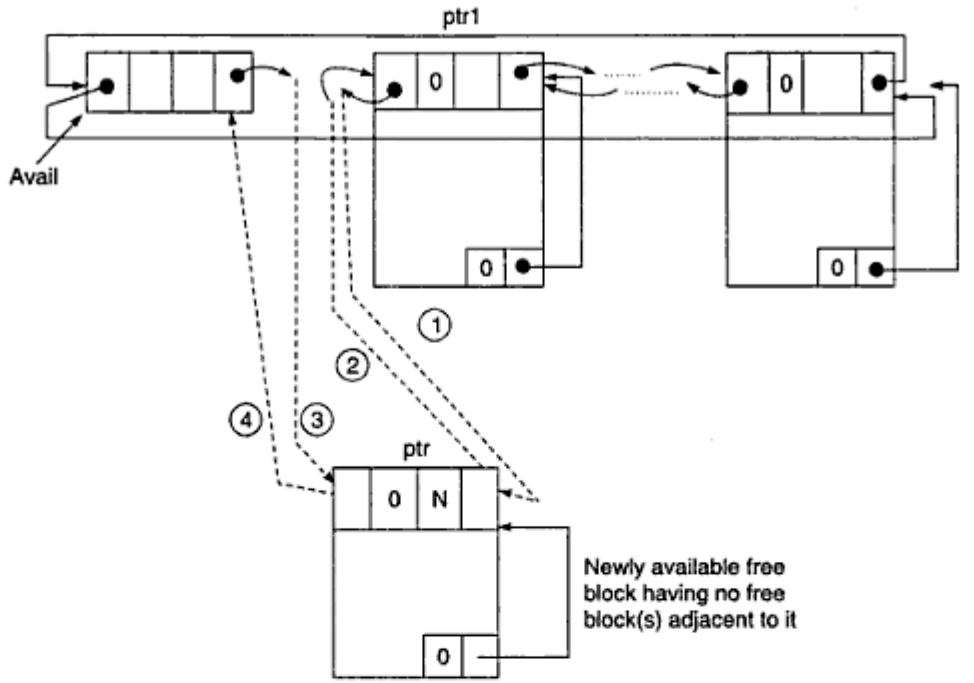
(b) Free storage list when P1 releases the block of 4K



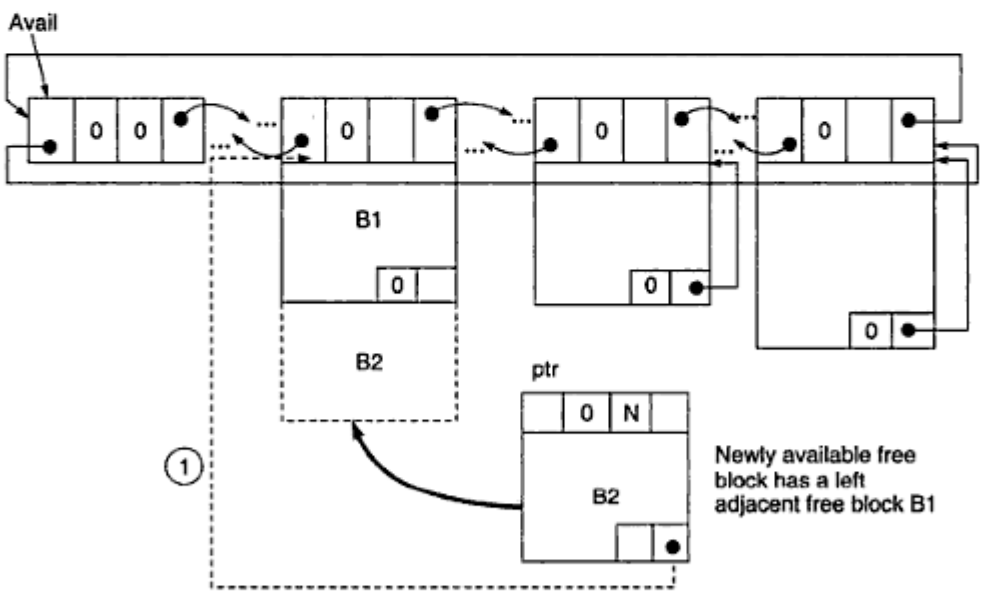
(c) Free storage space when P2 releases a block of size 2K

Figure 3.26 Deallocation of blocks in a boundary tag system.

The process of combining two adjacent blocks is very straightforward in the boundary tag system. Here is the use of two TAG fields in each block. Whenever a block is returned, the system will search the TAG fields of its adjacent blocks. If the TAG field(s) is/are found as zero, then combination can be carried out by changing a few pointers. Four cases of combination may arise. Changes in pointers for four cases are illustrated in Figure 3.27.

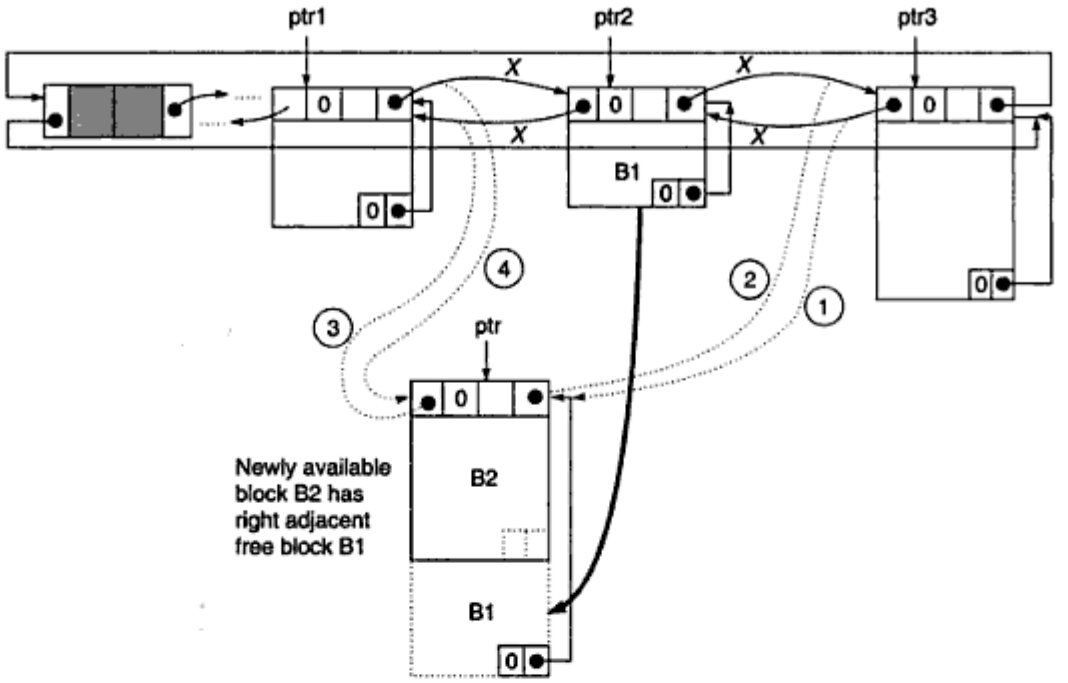


(a) Case 1: Newly freed block does not have any adjacent blocks on both sides

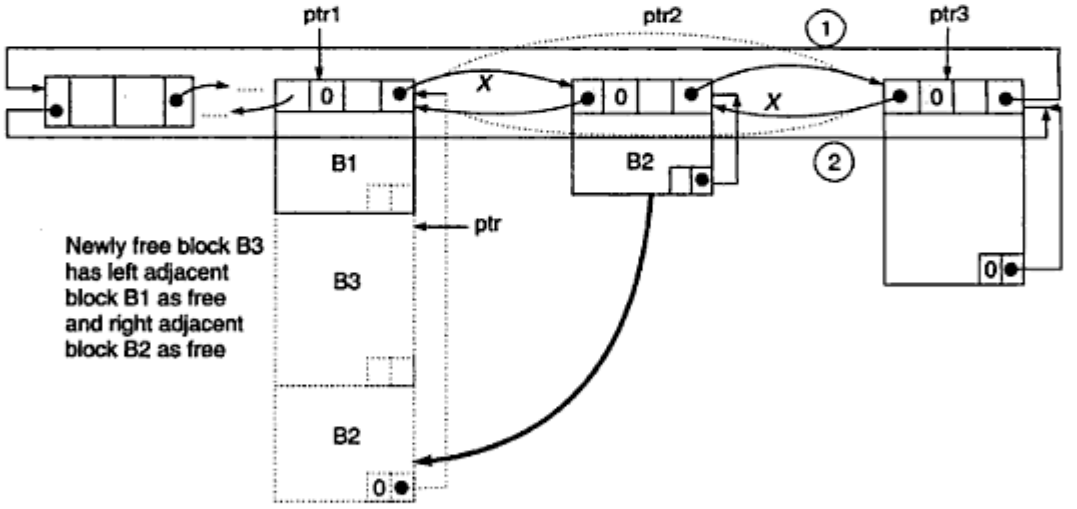


(b) Case 2: Newly freed block has only a left adjacent block as free

Figure 3.27 Continued.



(c) Case 3: Newly freed block has only a right adjacent block as free



(d) Case 4: Newly freed block has a left as well as right adjacent free blocks

Figure 3.27 Deallocation strategy in a boundary tag system.

From Figure 3.27, it can be understood how a very recent freed block can be combined with its adjacent free block(s) to build a larger block in the list of storages. The following is the algorithm *ReturnNode_BTS* describing the above illustrated cases of combining the free blocks when a free block is returned to the memory system.

Algorithm ReturnNode_BTS(PTR)

Input: A list of free storage having AVAIL as the pointer to the header of the lists; *PTR* is the pointer of the block to be deallocated.

Output: A list of free storage with newly free block inserted into it.

Data structure: Linked link structure for boundary tag system.

Steps:

1. n = PTR→SIZE
   / Both adjacent blocks of the newly freed block are not free */*
2. **Case 1:** ((PTR – 1) →TAG = 1) and ((PTR + n) →TAG = 1)
   */*** Insert at front ***/*
3. PTR→TAG = 0 // Set the TAG fields of the new block as free block
4. (PTR + n – 1) →TAG = 0
5. (PTR + n – 1) →UPLINK = PTR // Set the UPLINK
6. ptr1 = AVAIL→RLINK // Pointer to the first block in the list
7. ptr1→LLINK = PTR // Change the pointer as shown (1) in Figure 3.27(a)
8. PTR→RLINK = ptr1 // Change the pointer as shown (2) in Figure 3.27(a)
9. AVAIL→RLINK = PTR // Change the pointer as shown (3) in Figure 3.27(a)
10. PTR→LLINK = AVAIL // Change the pointer as shown (4) in Figure 3.27(a)
    / Only left adjacent block to the newly freed block is free */*
11. **Case 2:** ((PTR – 1).TAG = 0) and ((PTR + n).TAG = 1)
    */*** Append after the left adjacent block ***/*
12. ptr1 = (PTR – 1) →UPLINK // Obtain the pointer to the left free block
13. ptr1→SIZE = ptr1→SIZE + n // Update the size of left block which //includes new free block
14. (ptr1 + n – 1) →TAG = 0 // Set the TAG of the newly combined block
15. (ptr1 + n – 1) →UPLINK = ptr1 // Set the UPLINK as shown (1) in Figure 3.27(b)
    / Only right adjacent block to the newly freed block is free */*
16. **Case 3:** ((PTR – 1).TAG = 1) and (PTR + n).TAG = 0
    */*** Append right adjacent block after the newly freed block ***/*
18. ptr1 = (PTR – 1) →UPLINK
19. ptr2 = ptr1→RLINK
20. ptr3 = ptr2→RLINK
21. ptr3→LLINK = ptr1 // Change of pointer as shown (1) in Figure 3.27(c)
22. PTR→RLINK = ptr3 // Change of pointer as shown (2) in Figure 3.27(c)
23. PTR→LLINK = ptr1 // Change of pointer as shown (3) in Figure 3.27(c)
24. ptr1→RLINK = PTR // Change of pointer as shown (4) in Figure 3.27(c)
25. PTR→SIZE = n + ptr2→SIZE // Change the size of the newly combined block
26. (PTR + n – 1) →UPLINK = PTR // Set UPLINK fields of the newly // combined block
27. (ptr1 + n – 1) →TAG = 0 // Set TAG field of the newly combined block
    / Both the left and right adjacent blocks to the newly freed block are free */*
28. **Case 4:** ((PTR – 1) →TAG = 0) and ((PTR + n) →TAG = 0)
    */** Append new free block after left adjacent block and right adjacent block **/*

(*Contd.*)

```
29.   ptr1 = (PTR - 1) →UPLINK                    // Pointer to the left adjacent block
30.   ptr2 = ptr1→RLINK                           // Pointer to the right adjacent block
31.   ptr3 = ptr2→RLINK
32.   ptr1→RLINK = ptr3          // Change of pointer as shown (1) in Figure 3.27(d)
33.   ptr3→LLINK = ptr1          // Change of pointer as shown (2) in Figure 3.27(d)
34.   ptr1→SIZE = ptr1→SIZE + n + ptr2→SIZE       // Change the size of newly
                                                  // combined block
35.   (ptr1 + ptr1→SIZE - 1) →TAG = 0             // Set the TAG field of the
                                                  // newly combined block
36.   (ptr1 + ptr1→SIZE - 1) →UPLINK = ptr1       // Set UPLINK field as shown (3)
                                                  // in Fig. 3.27(d)
37.   Stop
```

Assignment 3.9

(a) What modification would you suggest in order to make the algorithm *GetNodeNextFit_BTS* as *GetNodeFirstFit_BTS* based on the first-fit allocation strategy?

(b) Write an algorithm for availing a node based on the best-fit storage allocation strategy.

(c) Procedure *ReturnNode_BTS* maintains a list of free storages. For a given request, this list is to be searched from header node whose pointer is AVAIL. One modification is suggested as "Recently freed block is for the consideration of next service". What necessary change you should incorporate in the procedure *ReturnNode_BTS* and *GetNodeNextFit_BTS* to incorporate the suggested modification?

3.10 BUDDY SYSTEM

So far we have discussed that the block sizes are either fixed or completely arbitrary. The *buddy system* is another storage management system which restricts the sizes of blocks to some fixed set of sizes. These blocks of restricted sizes are maintained in a linked list. Whenever a request for a block of size N comes, the number M, the smallest of the fixed sizes but equal to or larger than N, is determined, and a block of size M is allocated, if available on the list. If a block of size M is not available, then a larger block, if available, is split into two sub-blocks (known as *buddies*), each of them are also of fixed sizes, and this process is repeated until a block of size M is produced.

A buddy system specifies the restricted sizes as $F_0, F_1, \ldots, F_{MAX}$ for blocks according to some pattern. One principle for the specification of block sizes is the use of the recurrence relation

$$F_n = F_{n-1} + F_{n-k}, \qquad k \le n \le \text{MAX}$$

for a given k and MAX

$$F_0 = C_0, F_1 = C_1, \ldots, F_{k-1} = C_{k-1}$$

with initial conditions.

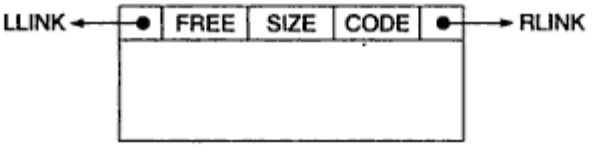
For example, if $k = 1$ and $F_0 = 8$, then the block sizes are 8, 16, 32, 64, 128, ... That is, the block sizes are successive powers of 2; and the buddy system based on such fixed sizes is called the *binary buddy system*. Another buddy system which restricts the fixed sizes to the Fibonacci sequence is called the *Fibonacci buddy system*. This system is for $k = 2$ and $F_0 = 8$, $F_1 = 13$; the sizes of the blocks then are 8, 13, 21, 34, 55, 89, ..., etc. Note that, F_0, F_1, F_2, etc. are specified based on the applications, that is, how much small sizes of the blocks it may require.

When a request for a memory space comes to the system, it searches for a block whose size is nearer to but not smaller than the size of the requested block; if such a block is found then it will be allocated; otherwise the next larger block or the largest block in the list (depending on the allocation strategy) will be split successively till the block having the size nearest to the required size is found. On the other hand, whenever a block is returned to the system, it will be combined with another free block into a larger free block if these two blocks were the buddies that were formed by some previous splitting. If this larger free block is again qualified as a buddy, then it will be recombined with its partner buddy, if it is free and this process of reformation will continue till no buddies fit for coalescing are present.

To illustrate the above allocation and deallocation techniques, let us consider the Fibonacci buddy system which restricts the block sizes as 8, 13, 21, 34, 55, 89 and 144; assume that 144 is the largest block possible, that is, maximum memory available. Figure 3.28 illustrates a few cases of allocation and deallocation.

From Figure 3.28, the following facts are evident: initially the system maintains a free block of size 144, say. When a request for a block of size 30 comes, it splits into two buddies, namely 89 and 55 and 55 is again split into two buddies, namely 34 and 21. As 34 is the nearest block size so it is allocated. Figure 3.28(a) illustrates this. Now for a subsequent request for a block of size 50, it searches and finds that a block of size 89 is available. As 89 is too large for the request, a split occurs into 55 and 34 and then the block of size 55 is allocated for the request. This is illustrated by Figure 3.28(b). Figure 3.28(c) illustrates the case when a block of size 55 is returned to the system. As 55 and 34 are free buddies, so they are combined into a block of size 89. Block 89 has no free buddy, so here merging comes to a halt. On the other hand, deallocation for a block of size 34 [Figure 3.28(d)] first combines 34 and 21 as they are free buddies into a large block of size 55. Next, 55 and 89 are free buddies, so another combination yields the largest block of size 144.

Our next discussion is to define the structure of a node which will be suitable to manage the above-mentioned operations. For this we can decide the following structure:

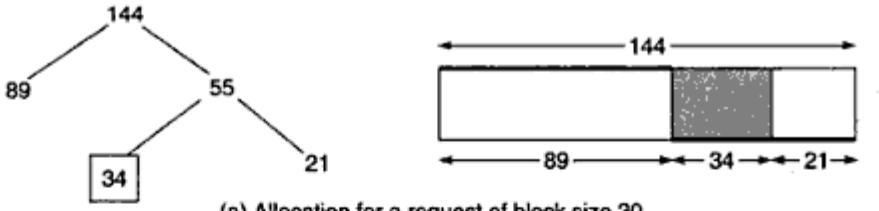


Two link fields, LLINK and RLINK, store the addresses of the predecessor and the successor of the block. The SIZE field stores the size of the block. Instead of storing the absolute size of the block we will store the index value for the size in the recurrence relation which the buddy system assumes. For example, in the case of Fibonacci buddy system, $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 8$, $F_1 = 13$, we will assume the value for SIZE field as

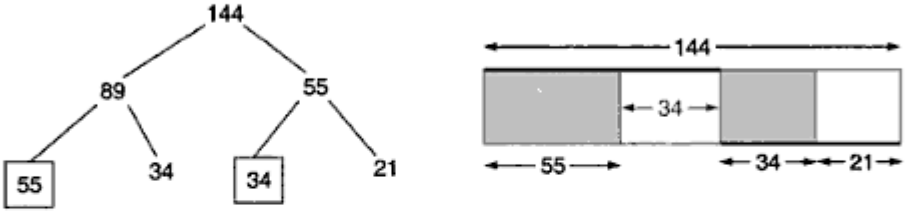
Absolute size (F_n)	SIZE (n)
$F_0 = 8$	0
$F_1 = 13$	1
$F_2 = 21$	2
$F_3 = 34$	3
$F_4 = 55$	4
$F_5 = 89$	5
$F_6 = 144$	6

and so on. Another field CODE has enough significance in this system. This field will store the information whether two blocks are buddies or not. A clever solution has been suggested by Hinds (in Proceedings of IEEE Symposium on Foundations of Computer Science, Vol. 19, 123–130, 1978) in order to decide the buddy property of blocks in a Buddy system; the solution is mentioned below:

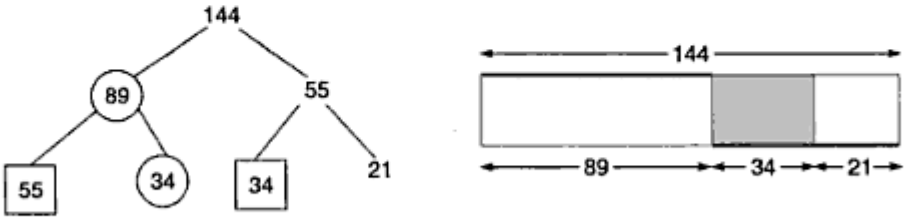
Initially:	CODE = 0	//This is for the largest block in the system
Splitting:	$CODE_{LEFT} = CODE_{PARENT} + 1$	//On splitting a parent block, two blocks
	$CODE_{RIGHT} = 0$	//namely LEFT and RIGHT will result
Merging:	$CODE_{PARENT} = CODE_{LEFT-1}$	//On merging of two buddies the resultant
		//block has CODE which is one less than its
		//left buddy



(a) Allocation for a request of block size 30



(b) Another allocation in response to a request for a block of size 50



(c) Deallocation of block size 55

Figure 3.28 Continued.

3.2 Swap two adjacent elements

(a) by interchanging the elements

(b) by adjusting only the pointers (and not the data).

In both cases, assume that the elements are stored in (i) a single linked list (ii) a double linked list. Which method out of (i) and (ii) might appear better? Why?

3.3 Given a single circular linked list containing a set of data, obtain the following from this data structure:

(a) Reverse the direction of links.

(b) For the given two elements in the list, find the distance (that is, the number of nodes) between them.

Write an algorithm and implement it with C++ in each case.

3.4 Given a single linked list containing any type of data, obtain the following:

(a) Reverse the ordering of the data.

(b) Suppose X and Y are two nodes in the list. Add all the nodes between X and Y (both are inclusive) so that

(i) X is the first node in the list.

(ii) Y is the last node in the list.

3.5 Let A and B be two lists representing two polynomials. Obtain the operation to find

(a) $C = A + B$, where C is a list representing a polynomial and is obtained by adding two polynomials A and B.

(b) $C = A - B$, where C is a list representing a polynomial and is obtained by subtracting two polynomials A and B.

(c) $C = A * B$, where C is a list representing a polynomial and is obtained by multiplicating two polynomials A and B.

3.6 The following are the sorting algorithms which can be performed on the set of data which are stored in a single linked list.

(a) Bubble sort

(b) Selection sort

(c) Insertion sort

(d) Radix sort

(e) Quick sort

Write the generic code for the above-mentioned sorting methods using the linked list data structure. Obtain a comparative study of the above sorting methods.

3.7 Suppose two lists INPUT and SPLITTER containing data of the same type are given, as shown in Figure 3.34. All the elements are to be split into three lists, LIST1, LIST2 and LIST3, based on the SPLITTER so that LIST1 contains all the data less than any data in SPLITTER, LIST2 contains data equal to any data in SPLITTER and LIST3 contains all the data greater than any data in SPLITTER. Write an algorithm for this. Implement the algorithm with C++.

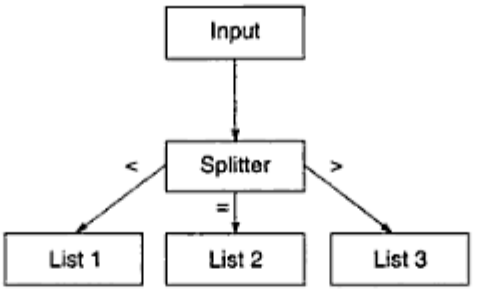


Figure 3.34 Splitting of data.

3.8 In programming languages such as BASIC and FORTRAN, we cannot maintain the list structure as there is no pointer concept. But in some cases lists are very much useful. Give an idea, as to how using two arrays one for DATA and the other for LINK, the concept of (a) single linked lists, (b) double linked list, and (c) single circular linked list can be obtained.

3.9 An alternative to the standard deletion strategy, a *lazy deletion*, is known. Here, the deletion of an element takes place logically but not physically. This is done by marking the node as deleted (using an extra bit field). The nodes containing a deleted tag remain in the list. If there are as many marked nodes as non-marked elements (50:50), the entire lists should be traversed to eliminate the marked nodes using standard deletion.

(a) Write an algorithm for lazy deletion.
(b) Implement your algorithm with C++.
(*Hint:* Use the principle of inheritance for the list traversal and usual deletion.)

3.10 Suppose, a university has to maintain a list of all students, a list of all subjects and a record of which student has registered for which course. Give an idea as to how using linked list structures, the above information can be maintained.

(*Hint:* Use multi-lists implementation of sparse matrix where the header column for course, header row for subjects and an entry corresponds to a student who registered in the subject of the course.)

3.11 It is required to maintain a library database using a number of lists as mentioned below:

The list BOOKS contains the information like title, accession number and tag field (to indicate whether a book is issued or not) for all the books in the library. Note that each book can be held in multiple copies, but their accession numbers are different.

Another list SUBSCRIBERS will contain the name, the borrower number and the list of books (with the date of issues) issued. Assume that a subscriber can be issued up to five books at the most and no more than two copies of the same book.

(a) Design a suitable data structure using a single linked list.
(b) Write a menu driven program using C++ for the following:
  (i) To issue a book
  (ii) Return a book
  (iii) Show the list of books issued to a subscriber
  (iv) Given a title, find out to whom it has been issued.

REFERENCES

Donal E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Reading, Massachusetts, 1984.

Jean Paul Tremblay and G. Paul, Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1987.

John Welsh, John Elder and David Bustard, *Sequential Program Structures*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

R. Hind, Efficient Dynamic Storage Management with Buddy System, *Proceedings of IEEE Symposium on Foundations of Computer Science*, Vol. 19, 123–130, 1978).

Thomas L. Naps, *Introduction to Data Structures with C*, West Publishing Company, West Virginia, 1986.

4

Stacks

4.1 INTRODUCTION

A *stack* is a linear data structure and very much useful in various applications of computer science. The implementation of the majority of systems programs is simplified using this data structure. Before discussing this data structure, let us first consider a few examples of the stack phenomenon.

Shunting of trains in a railway yard

Suppose there is a railway yard with a single track. Trains enter into the railway yard for placement, and when they exit, it is just in opposite order to that they had entered, i.e. the last train comes out first (see Figure 4.1).

Shipment in a cargo

For the shipment of goods, they are loaded into a cargo compartment. At the destination, they are unloaded exactly in the opposite sequence to that in which they were loaded. That is, the goods loaded last get unloaded first.

Plates on a tray

Suppose a chef placed the dishes on a tray one above the other. The waiter served the dishes to the customers in the opposite order that the chef placed them, that is, the dish at the top which

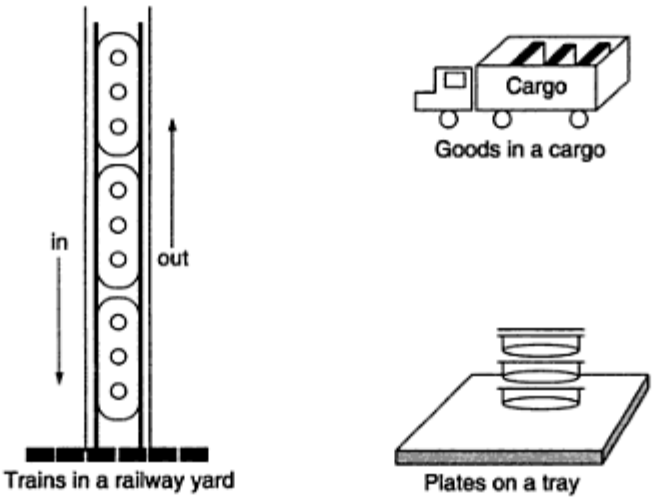


Figure 4.1 Some examples of stacks.

was placed last by the chef is serviced first. The first dish placed by the chef on the tray is serviced last by the waiter.

From the above examples, it is clear that a *stack* is something which follows the last-in first-out strategy. This is why a stack is alternatively termed LIFO (Last-In-First-Out).

4.2 DEFINITION

A *stack* is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only.

Like an array and a linked list, a stack is also a linear data structure but the only difference is that in the case of the former two, insertion and a deletion operations can take place at any position. 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 4.2 shows a typical view of a stack data structure.

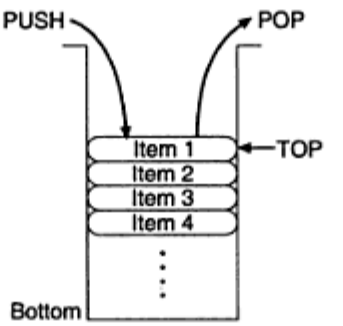


Figure 4.2 Schematic diagram of a stack.

4.3 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. Representations of stacks in a memory are discussed in the following two sections.

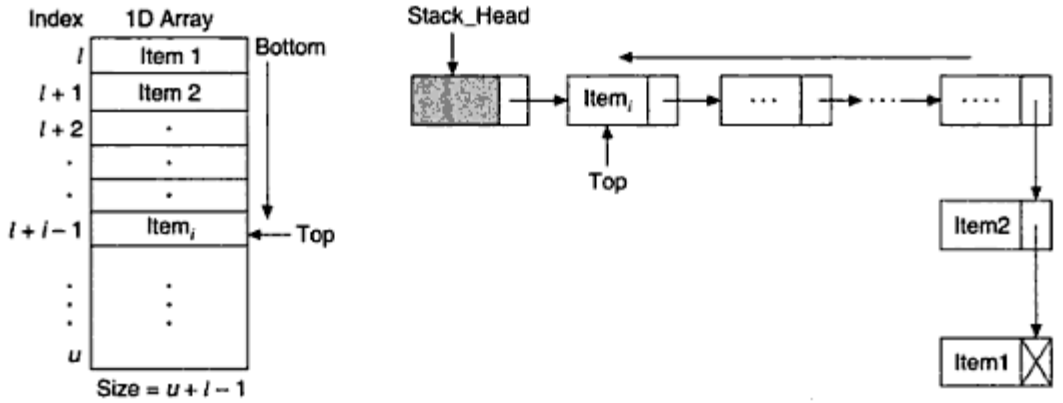
4.3.1 Array Representation of Stacks

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 4.3(a), $ITEM_i$ 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



(a) Array representation of a stack

(b) Linked list representation of a stack

Figure 4.3 Two ways of representing stacks.

4.3.2 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. Figure 4.3(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. The SIZE of the stack is not important here because this representation allows dynamic stacks instead of static stacks, as with arrays.

In the linked list representation of a stack, whether a stack is empty or not can be ascertained by testing the LINK field of the STACK_HEAD node. Note that a test for overflow is not applicable in this case.

4.4 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

Let us define all these operations of a stack. First, we will consider the above-mentioned operations for a stack represented by an array.

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
```

The operation *Status* now can be defined as follows:

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
```

Assignment 4.1

Using only PUSH, POP and STATUS as the basic operations defined in Section 4.4, write the procedures to solve the following problems.

(a) Read the top-most element in the stack. (Note that read will not remove the element.)
(b) Visit all elements in a stack. (Preferably without using another stack).
(c) Search for an item in the stack.

(Note: You should consider both types of memory representations of the stack.)

Assignment 4.2 (Multiple stack)

In several applications, more than one stack may be required together. Some stacks overflow whereas others are nearly empty. Suppose an application requires two stacks X and Y (Figure 4.4). One can define an array A with N_x elements for stack X and another array B with N_y elements for stack Y. Now instead of defining two separate arrays A and B, we can define a single array, say AB, with $N = N_x + N_y$ elements for X and Y together. Let us define the starting locations of items for stack X and stack Y as $AB[1]$ and $AB[N]$ respectively and X 'grows' to the right whereas Y 'grows' to the left.

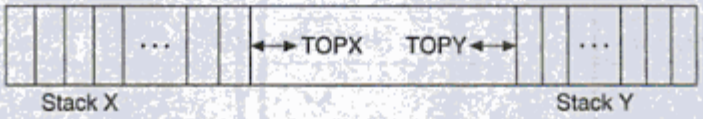


Figure 4.4 A multiple stack.

With this scheme, overflow will occur only when X and Y together have more than N elements. This technique will usually decrease the number of situations of occurrence of overflow even though we have not increased the total amount of space reserved for the two stacks.

Using this scheme, we need the modified versions of PUSH_A and POP_A operations as PUSH_X, PUSH_Y, POP_X, and POP_Y. Similarly, the operation STATUS_AB has to be defined to test the state of empty or full, percentage of space occupied by X and Y, etc. Write the basic operations for the implementation of such a multi-stack.

4.5 APPLICATIONS OF STACKS

Various applications of stacks are known. A classical application in a compiler design is the evaluation of arithmetic expressions; here the compiler uses a stack to translate an input arithmetic expression into its corresponding object code. Some machines are also known which use built-in stack hardware called 'stack machine'. Another important application of a stack is during the execution of recursive programs; some programming languages use stacks to run recursive programs. One important feature of any programming language is the binding of memory variables. Such binding is determined by the scope rules. There are two scope rules known: the *static* scope rule and the *dynamic* scope rule. Implementation of such scope rules is possible using a stack known as a *run time stack*.

The following subsections highlight the above-mentioned applications of stacks.

4.5.1 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 [for example, – (unary), + (addition), – (subtraction), * (multiplication), / (division), ^ (exponentiation), % (remainder modulo), etc.], relational operators (for example, <, >, <=,

< >, >=, etc.), and Boolean operators (such as, AND, OR, NOT, XOR, etc.). In addition to these, parentheses such as '(' and ')' are also used. A simple arithmetic expression is cited below:

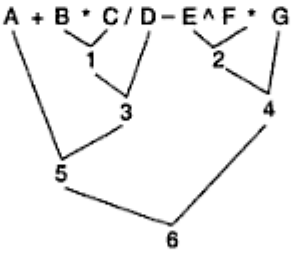
A + B * C / D – E ^ F * G

The problem to evaluate this expression is the order of evaluation. There are two ways to fix it. First, we can assign to each operator a precedence and associativity. For example, a set of usual operators with their precedence and associativity is given in Table 4.1.

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:

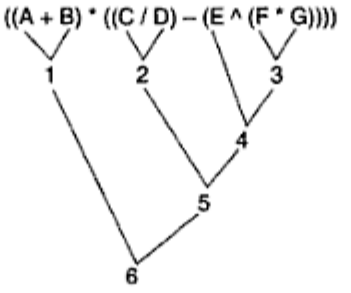


It should be noted that the above rules for precedence and associativity vary from one programming language to another.

Another way of fixing the order of evaluation is parenthesizing the expression fully; this allows one to override the rules for precedence and associativity. The following is the parenthesized version of the same expression:

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

With this parenthesization, the innermost parenthesis part (called sub expression) will be evaluated first, then the next innermost, and so on; such a sequence of evaluations is shown below:



Whatever way we may specify the order of evaluations, the problem is that we must scan the expression from left to right repeatedly. Hence, the above-mentioned processes are inefficient because of the repeated scanning required. Another problem is the ambiguity about how the compiler can resolve to generate a correct code for a given expression. The last problem mainly occurs for a partially parenthesized expression. These problems can be solved in the following two steps:

1. Conversion of a given expression into a special notation
2. Evaluation/production of an object code using a stack.

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>.

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, /GH, etc.

The prefix notation was introduced by the Polish mathematician Jan Lukasiewicz and hence also termed *Polish notation.*

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 postfix notation is just reverse of the Polish notation, hence it is also termed *reversed Polish notation.*

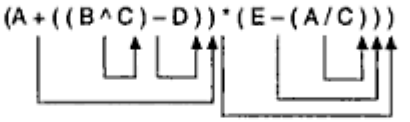
It may be noted that in all of the above notations, a unary operator precedes its operand.

An expression given in infix notation can easily be converted into its equivalent prefix or postfix notation. The following rule is applied to convert an infix expression into a postfix form.

- Assume the fully parenthesized version of the infix expression.
- Move all operators so that they replace their corresponding right part of parentheses.
- Remove all parentheses.

The following example illustrates this conversion. For simplicity, let us consider a fully parenthesized expression.

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 / – *

The following points may be observed from the above three notations:

1. In both prefix and postfix equivalents of an infix expression, the variables are in the same relative positions.
2. The expressions in prefix or postfix form are completely parenthesis free.
3. The operators are rearranged according to the rules of precedence of operators.

Out of these three notations, the postfix notation has certain advantages over the other notations from the computational point of view. The main advantage of postfix is its evaluation. During the evaluation of an expression in postfix notation it is no longer required to scan the expression from left to right several times, but scanning is required exactly once. This is possible using a stack and will be discussed shortly.

Thus, evaluation of an expression is a two-step process. First, we have to convert the expression into its postfix notation, and then evaluate this expression in postfix notation. In each step, the stack is the main data structure that is used to accomplish these tasks.

The uses of the stack for the purpose and the above-mentioned procedures are discussed below. We will assume the array representation of a stack in our discussions.

Conversion of an infix expression to postfix expression

To formalize the conversion method, we will assume simple arithmetic expressions containing the +, –, *, /, and ^ (exponentiation) operators only (i.e. without unary operators, Boolean operators and relational operators). The expression may be parenthesized or unparenthesized.

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.

Our next step is iterative: read one input symbol at a time and decide whether it has to be pushed onto the stack or not. This decision will be governed by Table 4.2.

Table 4.2 In-stack and in-coming priorities of symbols

Symbol	*In-stack priority value*	*In-coming priority value*
+ –	2	1
* /	4	3
^	5	6
operand	8	7
(	0	9
)	–	0

From the table, it can be noted that for a symbol we have considered two priority values, viz. *in-stack* priority and *in-coming* priority values. A symbol will be pushed onto the stack if its in-coming priority value is greater than the in-stack priority value of the top-most element. Similarly, a symbol will be popped from the stack if its in-stack priority value is greater than or equal to the in-coming priority value of the in-coming element. In order to define the algorithm, we will assume the following functions:

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.

Let as assume that a stack of capacity SIZE is known and TOP is the current pointer in it. PUSH and POP are usual operations of the stack.

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
```

Note: This is a procedure irrespective of the type of memory representation of the stack to convert an infix expression to its postfix form using the basic operations of the stack, namely PUSH and POP. These operations can be replaced with their respective versions and hence implementable to stack with any type of memory representation.

EXAMPLE 4.1

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

Input: (A + B) ^ C – (D * E) / F) (infix form)
Symbol reading: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

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)

The above procedure assumes that the input infix expressions are according to the right syntax. So, if the input expression is not correct, its postfix form will not be correct. Extending the same idea, we can incorporate relational, Boolean and unary operators in the above procedure.

Assignment 4.3

(a) Verify the algorithm *InfixToPostfix* for producing the postfix notations of the following from the infix notations.

(i) (A + B) ^ C ^ D * E – F / G
(ii) A + * B – C
(iii) (A + B) * (C – D)
(iv) A * (+ B) – C / D

(b) Modify the algorithm *InfixToPostfix* to convert an infix expression containing Boolean operators and relational operators and unary operators to their equivalent postfix form.

(c) For the algorithm *InfixToPostfix*, we assume that the input expression is correct. That is, there is no error checking performed. But the input expression may not be well formed always: expressions having two operands together or two binary operators together or unmatched parentheses are some common mistakes. Write a procedure, so that it will check for the correct expression at the time of producing the postfix form.

Evaluation of a postfix expression

For a given expression in postfix notation, it can be easily evaluated. The following algorithm *EvaluatePostfix* is used to evaluate an arithmetic expression in postfix notation using a stack.

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 4.3

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 PostfixToCode
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 4.4

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	

Assignment 4.4

(a) Consider an expression which contains relational and Boolean operators. Formulate the precedence values required for these operators to convert such an expression to reversed Polish notation.

(b) Modify the algorithm *PostfoxToCode* so that it will also handle the unary operators.

(c) Modify the algorithm *PostfoxToCode* for the six relational operators <, >, <=, >=, < >, =) and four Boolean operators (AND, OR, NOT, XOR).

4.5.2 Code Generation for Stack Machines

In Section 4.5.1, we discussed the generation of codes for a given arithmetic expression in postfix form. The codes are of the type called *single address* codes. These codes are for those machines which maintain a number of registers; the registers are to store the temporaries. One problem using other machines, which have a very limited number of registers, is how to handle the storage of intermediate results.

Some machine architectures are known which use a stack instead of registers to store temporaries or intermediate results; obviously the stack size in this case should be adequate enough to handle a large expression. Here, we will present the description of a simple hypothetical machine to illustrate the concept of code generation for the postfix form of arithmetic expressions.

Let us assume the following set of instructions (given in mnemonic forms) of the stack machine.

PUSH <name> To load from memory onto stack. This instruction loads an operand from the memory location named <name> and places the content of <name> into the stack.

POP <name> To store the top element of the stack in memory. The content of the top of the stack is removed and stored in the memory location referenced by <name>.

Let us assume that our machine performs the following arithmetic operations only:

ADD, SUB, MUL, DIV

Assume that the operations can be stated with zero-address operation code. Operands are explicitly mentioned as top two elements in the stack. The operations are as stated below:

$$S[TOP - 1] = S[TOP - 1]\ \theta\ S[TOP]$$
$$TOP = TOP - 1$$

where θ stands for an operation. This simply means that, for an operation, the operands are from the top two elements, the result is stored at the second top location and the stack pointer is decremented by one.

To illustrate the hypothetical machine, let us consider the following arithmetic expressions:

$$A = B * C - A$$

The corresponding postfix notation can be obtained as follows:

$$A\ B\ C\ *\ A\ -\ =$$

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.

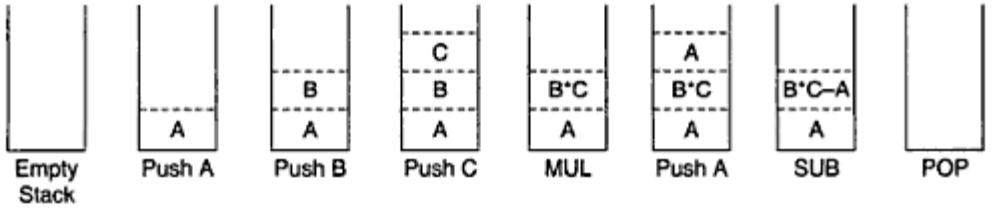


Figure 4.5 Evaluation of an arithmetic expression using a stack machine.

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
```

Here, we assume the procedure ***ProduceCode***( ) which will add the code into the list of codes. It may be observed that the above-mentioned procedure *PostfixToCodeForStackMachine* generates the optimized code. Here, the stack operations PUSH and POP are only involved, which perfectly fit the process of generating the object code from the postfix notation.

4.5.3 Implementation of Recursion

Recursion is an important tool to describe a procedure having several repetitions of the same. A procedure is termed *recursive* if the procedure is defined by itself. As a simple example, let us consider the case of calculation of the factorial value for an integer n.

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

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
```

Here, Step 4 recursively defines the factorial of an integer *N*. This is actually a direct translation of $n! = n * (n - 1)!$ in the form of *Factorial_R* $(n) = n *$ *Factorial_R* $(n - 1)$. This definition implies that $n!$ will be calculated if $(n - 1)!$ is known, which in turn if $(n - 2)!$ is known and so on until $n = 0$ when it returns 1 (Step 1).

The question that arises is how this definition can be implemented. We will see that a data structure stack can be used for this purpose. Some programming languages such as C, Pascal, which have a dynamic memory management mechanism, can directly accept the recursive definition of procedures; compilers of these programming languages are responsible to produce an object code suitable for execution using a stack (called *run time stack*). In other programming languages such as FORTRAN, COBOL, BASIC, which do not have a dynamic memory management mechanism, it is the user's responsibility to define and maintain the stack in order to implement the recursive definition of the procedure.

For an illustration, let us consider the calculation of 5! using the recursive definition. To do this, the following steps are needed:

Steps:

```
 1.  5! = 5*4!
 2.          4! = 4*3!
 3.                  3! = 3*2!
 4.                          2! = 2*1!
 5.                                  1! = 1*0!
 6.                                          0! = 1
 7.                                  1! = 1
 8.                          2! = 2
 9.                  3! = 6
10.          4! = 24
11.  5! = 120
```

Here, it is required to push the intermediate calculations till the terminal condition is reached. In the above calculation for 5!, Steps 1 to 6 are the push operations. Then subsequent pop operations will evaluate the value of intermediate calculations till the stack is exhausted.

To control the recursion, we have to maintain the following stacks:

Stack(s) for parameter(s) // To store the parameter(s) with which the recursion is defined

Stack(s) for local variable(s) // To hold the local variable(s) that are used within the definition

A stack to hold the return address.

From the above list of stacks it is evident that the number of stacks required is as many as there are parameters in the procedure. For some procedure which does not use any local variable, no stack is required for that purpose. Similarly, a stack to hold the return address may not be required for some recursive procedure.

Details about implementing recursive procedures using a stack are illustrated in the following sections with some well-known problems. Let us assume that all stacks are represented using an array structure. Also, assume that the sizes of stacks are adequate to run a procedure.

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.

4.5.4 Factorial Calculation

Recall that the factorial for an integer N can be defined recursively as follows:

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
```

Note that Steps 3–10 control PUSH and Steps 11–13 are POP operations. Here, the stacks are initialized by the calling value and address of the Return statement, that is, Step 14. POP_PARAM( ) and POP_ADDR( ) are the two POP operations assumed on two stacks PARAM and ADDR, respectively.

The above implementation is illustrated for $N = 5$ in Figure 4.6.

Action	*Execution*	*Step*	*Stack content*
1.	val=5, TOP=0 addr = Step 10	1	
	PUSH(5, Step 10)	2	PARAM 5 ADDR 10
	val=4, addr = Step7	3	
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	
7.	val=1, addr=Step7	8	5 4 3 2 1 10 7 7 7 7 X
	fact = 1*1(=1)	7	
8.	val=2, addr=Step7	8	5 4 3 2 10 7 7 7 X
	fact=2*1(=2)	7	
9.	val=3, addr=Step7		5 4 3 10 7 7 X
	fact=3*2(=6)		
10.	val=4, addr=Step7	8	5 4 10 7 X
	fact=4*6(=24)	7	
11.	val=5, addr=Step7	8	5 10 X
	fact=5*24(=120)	7	
12.	Go to Step 7	10	

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

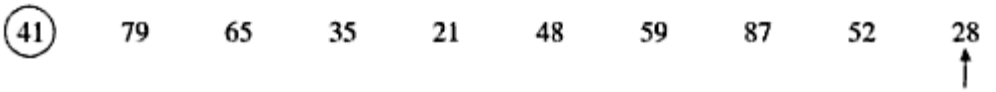
It may be observed that writing an iterative version for the calculation of a factorial is not only easy but its execution is also efficient. The example for factorial calculation using a stack is only of theoretical interest to illustrate the stack mechanism of a recursive procedure execution.

The next two examples highlight the actual advantages of stacks in recursion.

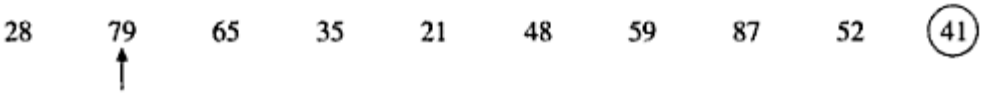
4.5.5 Quick Sort

One useful application of a stack is to sort a number of data using the quick sort algorithm. 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.

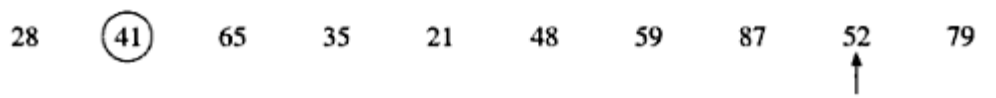
In the case of quick sort, the list to be sorted is partitioned into two sub-lists so that sorting these two sub-lists is sorting the main list; sorting the sub-list again follows the same procedure recursively. Note that partition is to be done in such a way that sorting of the sub-lists is the sorting 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:



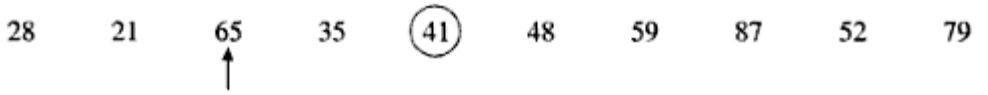
Here, 41 is selected as the pivot element, which is encircled. In order to place 41 in its right position, first compare this element with the element at the extreme right end (shown with an upright arrow below it, we call this the pointer), swap the elements if they are not in order (that is, if the *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:



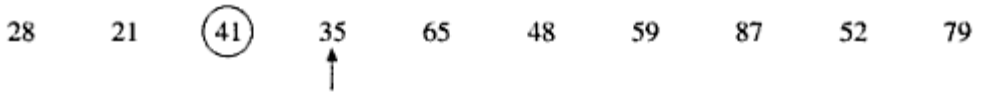
We see that this swap places the pivot element at the extreme right end. We now compare the pivot element with the element which is next to the element just swapped, see the pointer). In this case, a swap will occur if the comparison tells that they are not in order (that is, the *pivot element is smaller 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.



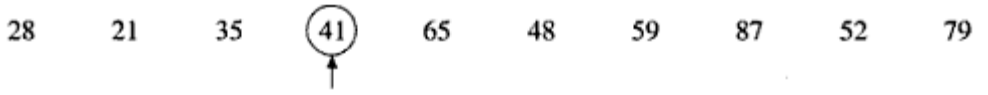
Repeatedly applying the above steps one can get the following observations. Comparisons with the pivot will shift the pointer to the left side from 52 to 87, 87 to 59, 59 to 48 and 48 to 21, when we find that the pivot element is larger than 21. So a swap will take place:



Comparing from left, we get the following change:



Another comparison from right yields



Now, we should stop as we have scanned the entire list (as the pointer has reached the pivot) and 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.

This explains how we can partition a given list into two sub-lists. It may be noted that the two sub-lists are not necessarily of equal length. 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.

The quick sort algorithm uses a procedure *Divide*( ). Let us define it next.

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
```

(*Contd.*)

```
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

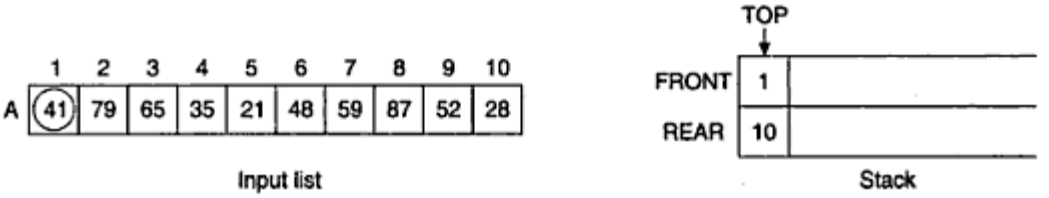
Input: An array A with N elements
Output: Sorted list of elements in A in ascending order
Data structure: Array representation of stack with TOP as the pointer to the top-most element.

Steps:

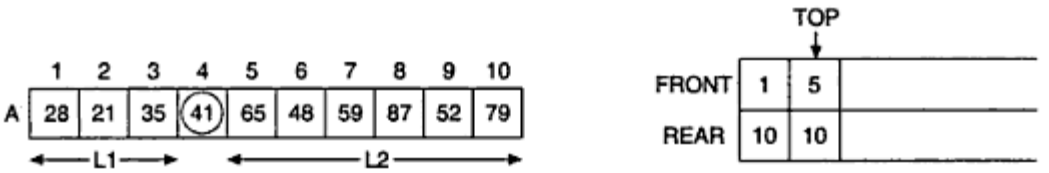
```
1.  fl = 1, el = N                        // Boundaries of the list
2.  top = NULL,                           // Stacks are empty initially
3.  If (N > 1)                  // If the list is not empty or has more than a single element
4.      PUSH(fl, el)                      // Push the values into 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
```

An illustration of recursive execution of quick sort using stacks is shown in Figure 4.7.

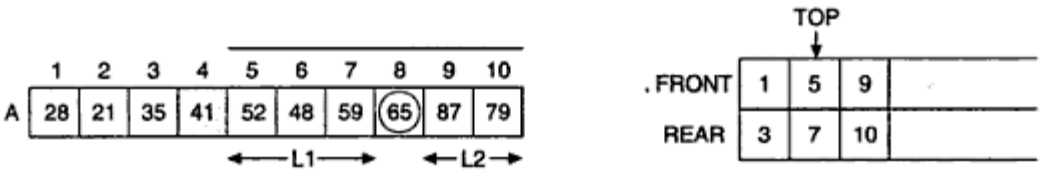
In the above algorithm *QuickSort*( ), we assume *PUSH*(FL, EL) to push FL and EL into the FRONT and END of stacks, respectively. Similarly, *POP*(FL, EL) is to *POP* the items from two stacks FRONT and END and they are stored as FL and EL, respectively. The details of the quick sort method are illustrated through an example as shown in Figure 4.7. Note that an



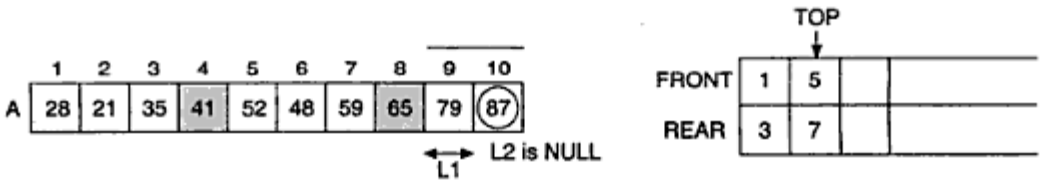
(a) Initially the entire list (1, 10) is Pushed into the stack



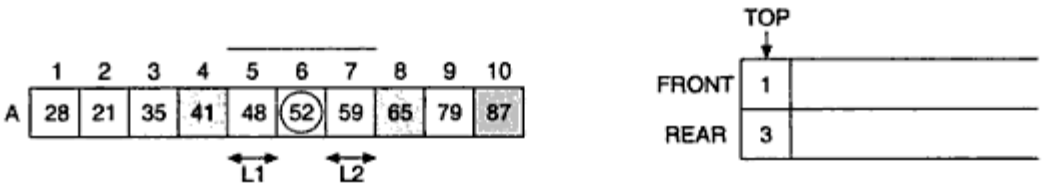
(b) The list (1, 10) is POPed and divided into two sub-lists L1(1, 3) and L2(5, 10) using the Divide procedure and pushed into the stack



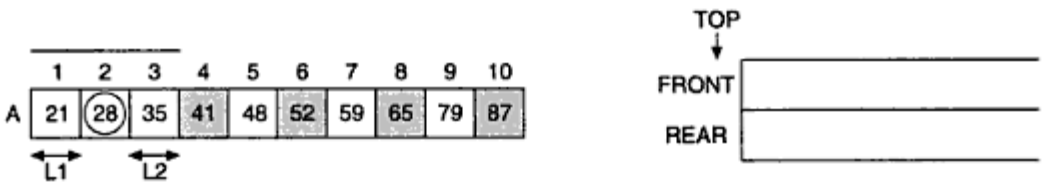
(c) The list (5, 10) is POPed and divided into two sub-lists L1(5, 7) and L2(9, 10) using the Divide procedure and pushed into the stack



(d) The list (9, 10) is POPed and divided into two sub-lists L1(9, 9) and L2(empty) using the Divide procedure.
No push operation for either list as they are either empty or containing single element.



(e) The list (5, 7) is POPed and divided into two sub-lists L1(5, 5) and L2(7, 7) using the Divide procedure.
No push operation for either list as each of them contains a single element.



(f) The list (1, 3) is POPed and divided into two sub-lists L1(1, 1) and L2(3, 3) using the Divide procedure.
No push operation for either list as each of them contains a single element.

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

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.

The above discussion of the quick sort technique is in the context of the application of a stack and hence we emphasize its use in the algorithm. A general version of the quick sort technique, its implementations and variations are discussed in Chapter 10, Section 10.5.4.

4.5.6 Tower of Hanoi Problem

Another complex recursive problem is the tower of Hanoi problem. This problem has a historical basis in the ritual of ancient Vietnam. The problem can be described as follows:

Suppose there are three pillars *A*, *B* and *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 4.8 represents the initial and final stages of the tower of Hanoi problem for $N = 5$ discs.

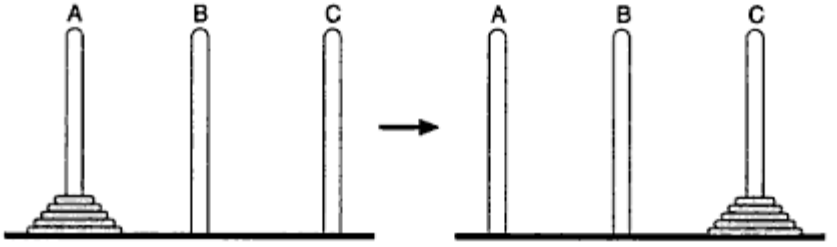


Figure 4.8 Tower of Hanoi problem with 5 discs.

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 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.

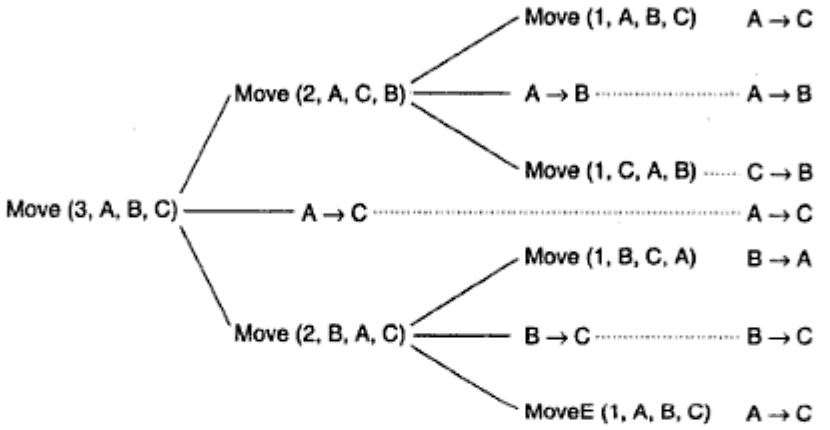


Figure 4.9 Tower of Hanoi (with $N = 3$) solution with recursion.

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

With these assumptions, the recursive implementation of the tower of the Hanoi problem will be defined as follows:

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 $2^N - 1$.

Assignment 4.5

The celebrated Fibonacci sequence (denoted as F_0, F_1, F_2, F_3, ..., F_n) is as below:

1, 1, 2, 3, 5, 8, 13, ...

The above sequence can be defined using recursion as follows:

If $n = 0$ or $n = 1$ then $F_n = 1$

Else $F_n = F_{n-1} + F_{n-2}$

Show how this Fibonacci sequence can be generated by writing a recursive program and show how recursive implementation can be done using a stack

Assignment 4.6

To compute the greatest common divisor (GCD) of two integers M and N, the recursive definition (called Euclid's algorithm) as follows:

GCD(N, M) if $N > M$

GCD(M, N) = M, if $N = 0$

GCD(N, MOD(M, N)), otherwise

Here MOD(M, N) is the module function which returns the remainder when M is divided by N. Write a recursive program for the above implementation using a stack.

4.5.7 Activation Record Management

Before discussing the application of stacks to the management of activation records, let us say a few words about its theory.

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 4.10.

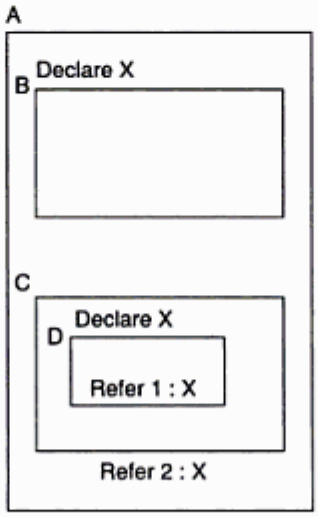


Figure 4.10 A block structured program.

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 4.10) 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 Figure 4.11. *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 'call2 *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 'call4 *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 Declare1 and another from Declare 2 in the same program.

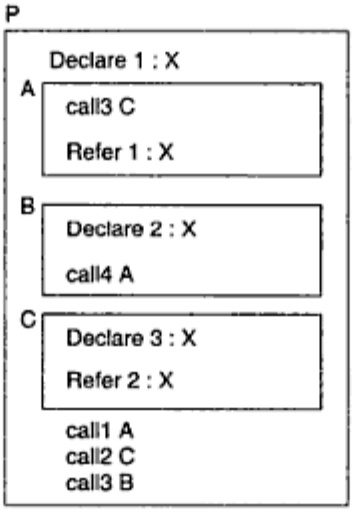


Figure 4.11 Dynamic scope rule.

Assignment 4.7

Let us consider the following program structure:

```
MAIN
    Boolean b = TRUE

    Procedure P
        Begin
            Output (b)
        End

    Procedure Q
        Boolean b = FALSE
        Begin
            Output (b);
            call P;
        End

    BEGIN          //MAIN
        call P;
        call Q;
    END
```

What output will you find (a) if the scope rule is static and (b) if the scope rule is dynamic.

Implementation of scope rules using stack

The next question that arises is how to implement a scope rule. This is completely a burden on the system programmer. Actually, the implementation of scope rules is meant to solve the problem of allocation of memory variables that are declared in different blocks. As we know, there are two different types of storage allocations, namely *static storage allocation* and *dynamic storage allocation*. 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.

We are to discuss the implementation of scope rules using the dynamic memory allocation strategy. Easy implementation is possible using a stack called a *run time stack*. In this

implementation, 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 4.12.

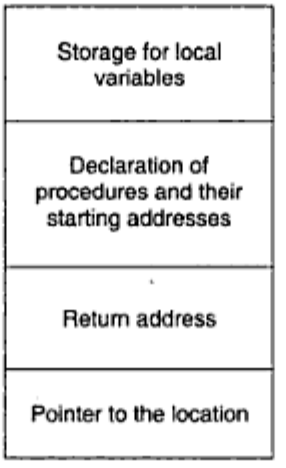


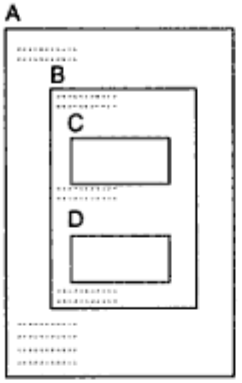
Figure 4.12 Structure of an activation record.

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 4.14.



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.

This program requires a total of five activation records. These are displayed in Figure 4.15.

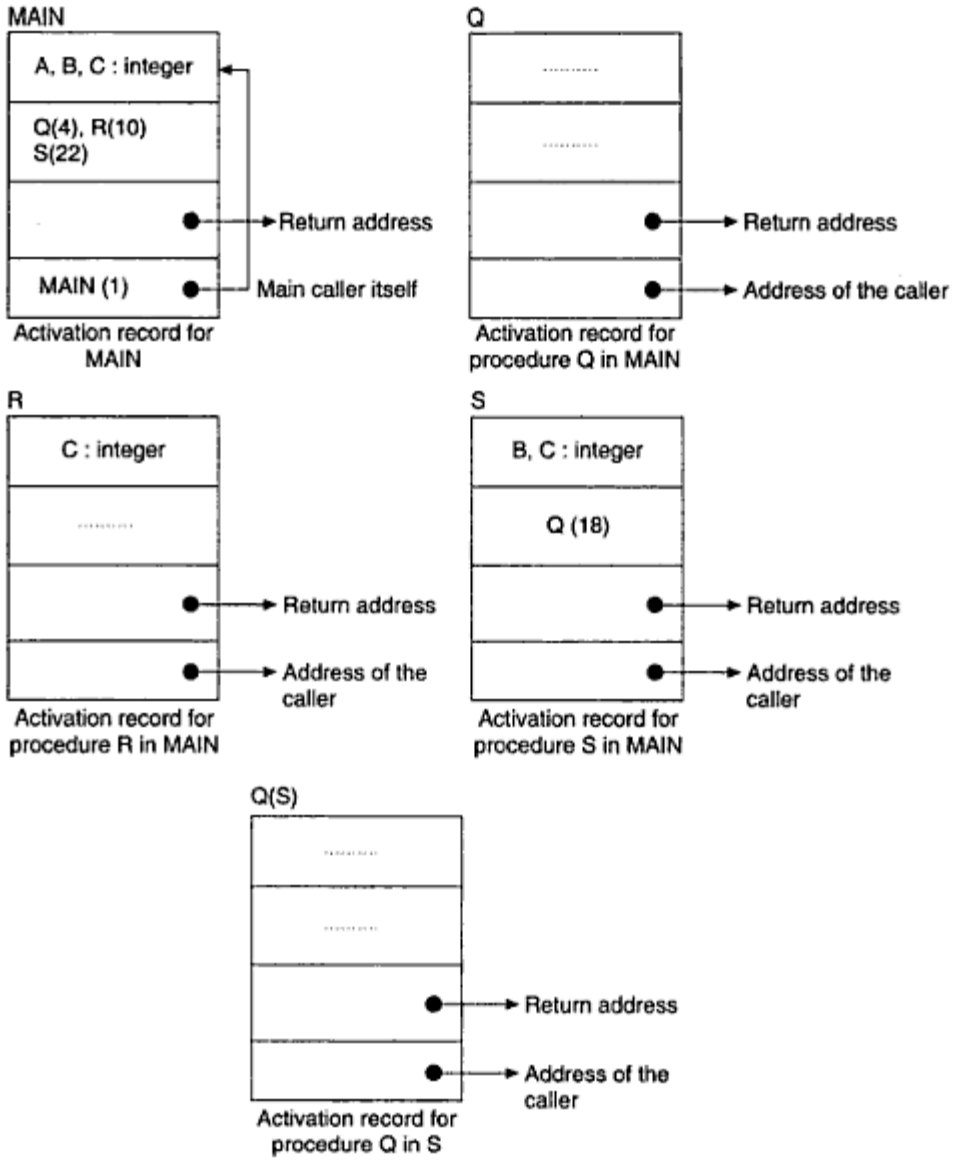


Figure 4.15 Activation records of various procedures.

However, the field for the pointer to the locator depends on the scope rule. For the dynamic scope rule, there is no need to maintain such a field.

Now, during the execution, the activation records of various procedures are either to be pushed or to be popped to and from the stack as per the requirement of the program. We will consider the stack represented with a single linked list, whose node structure is as shown in Figure 4.16.

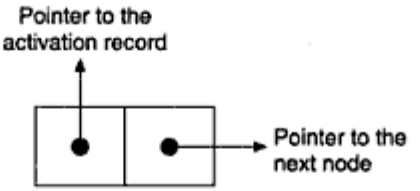
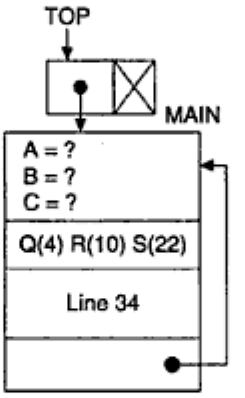


Figure 4.16 Node structure for linked list representation of a stack.

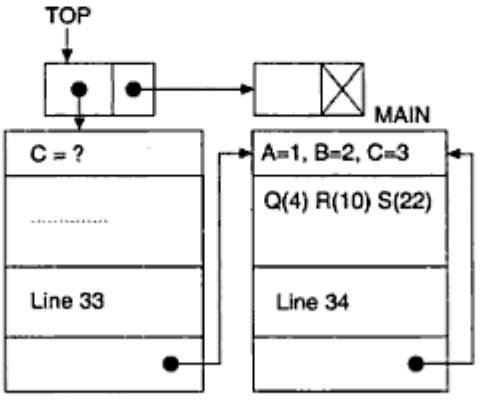
Implementation of static scope rule

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 (Figure 4.14) is illustrated in Figure 4.17.

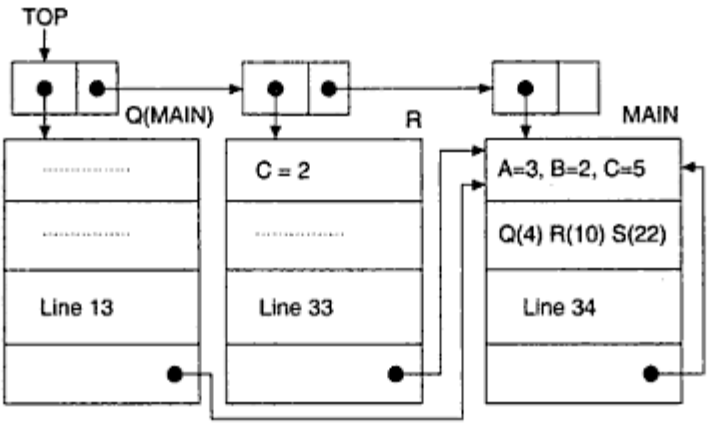


(a) MAIN begins its execution at line 28. Its activation record is PUSHed into the stack

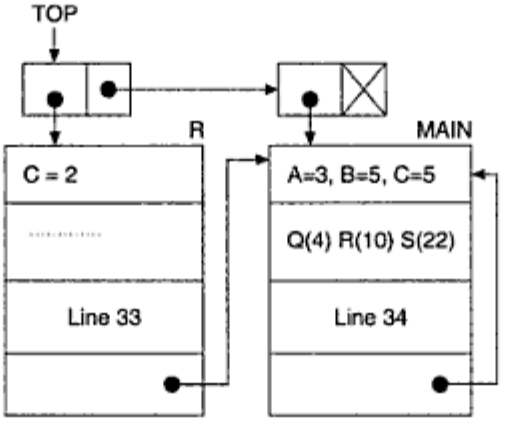


(b) Program control reaches the line 32, R is invoked and its activation record is PUSHed into the stack

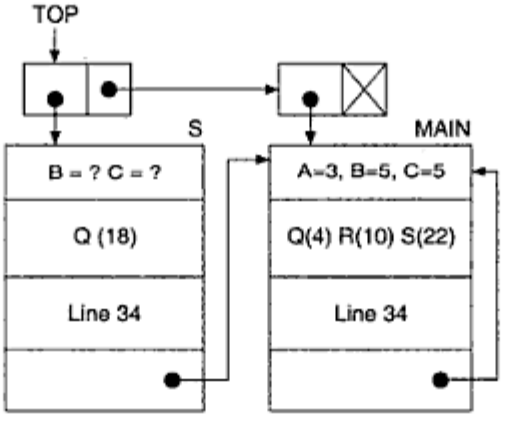
Figure 4.17 Continued.



(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.

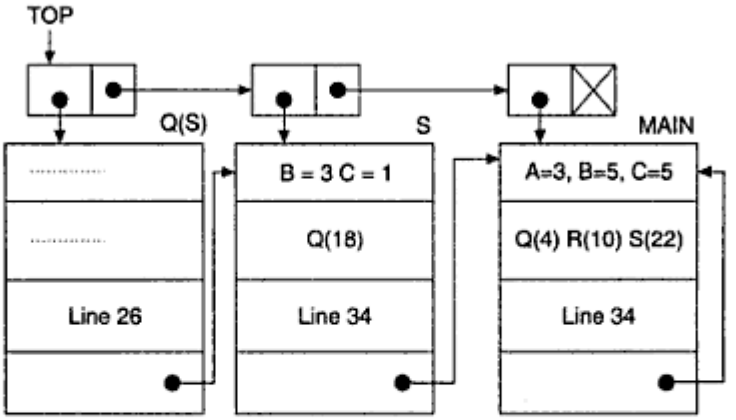


(d) When Q finishes its execution, control gets the return address from its activation record which is line 13. Activation record of Q is removed. Now, references to B and A at line 13 are obtained from their declaration in MAIN.

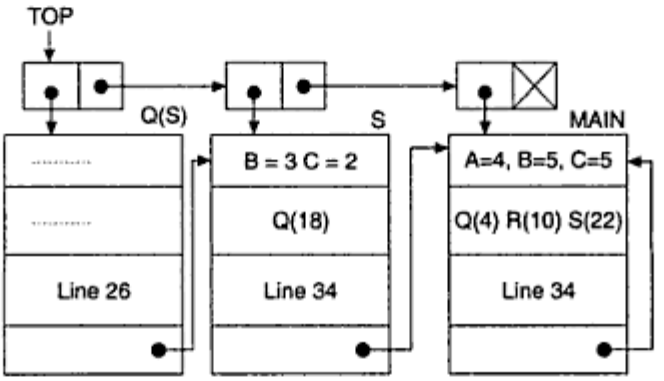


(e) When R completes its execution, control returns to line 33; the procedure S is invoked whose reference is resolved by the current activation record namely, MAIN and record of S is PUSHed into the stack

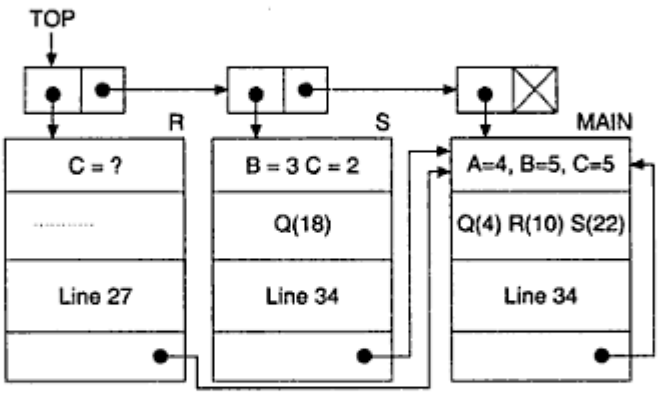
Figure 4.17 Continued.



(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.

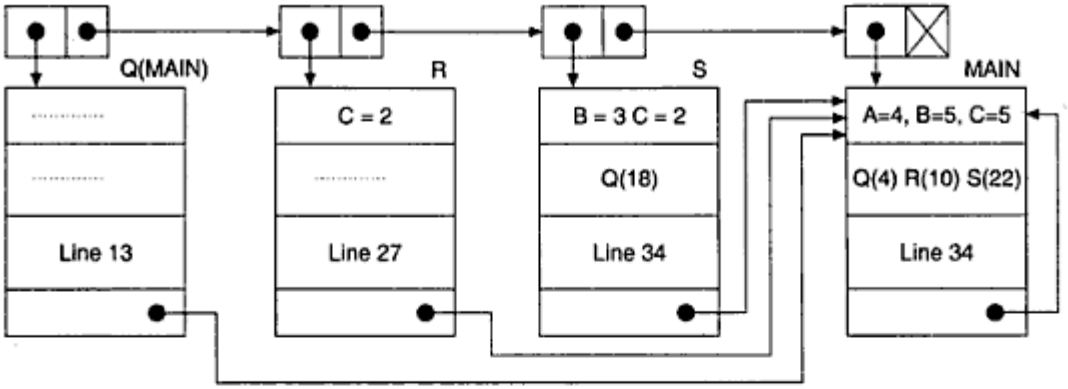


(g) Execution of Q is started and references of A and C at lines 19 and 20 are resolved from S and MAIN, respectively.

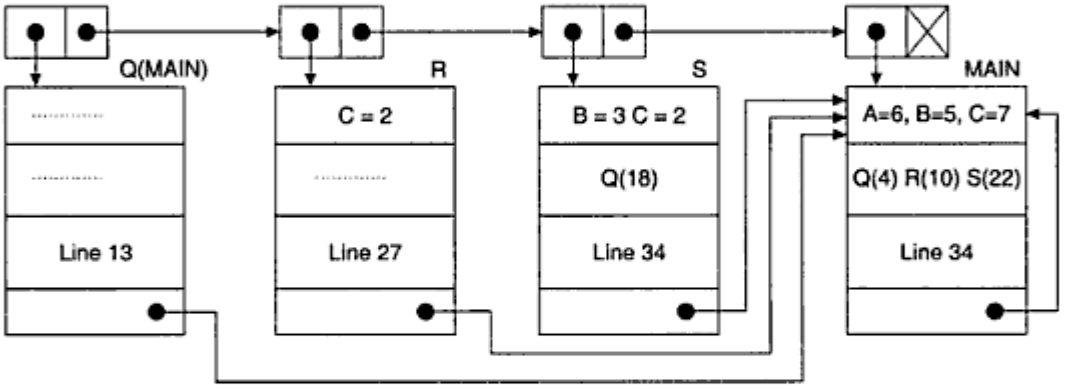


(h) When Q finishes its execution, control returns to line 26, its activation is then POPed and procedure R is invoked. This R is resolved from the activation record of MAIN.

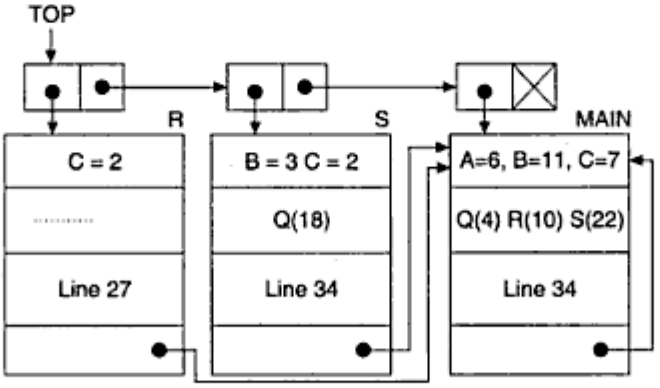
Figure 4.17 Continued.



(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.

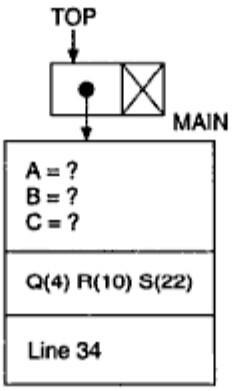
Figure 4.17(a)–(k) Execution of MAIN using the static scope rule.

Implementation of dynamic scope rule

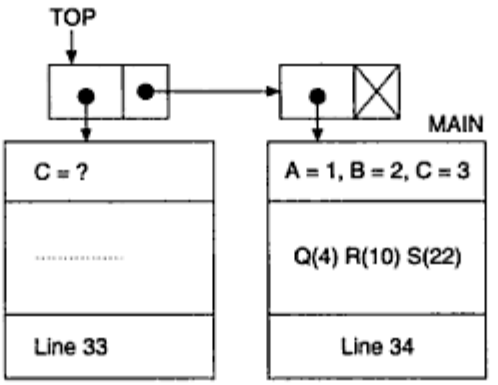
Implementation of the dynamic scope rule is much easier than the implementation of the static scope rule. For the dynamic scope rule, the structure of an activation record is the same as for the static scope rule except that here it is not required to maintain a pointer field to store the address of 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 structure as mentioned in Figure 4.14, its execution using the dynamic scope rule is illustrated in Figure 4.18.

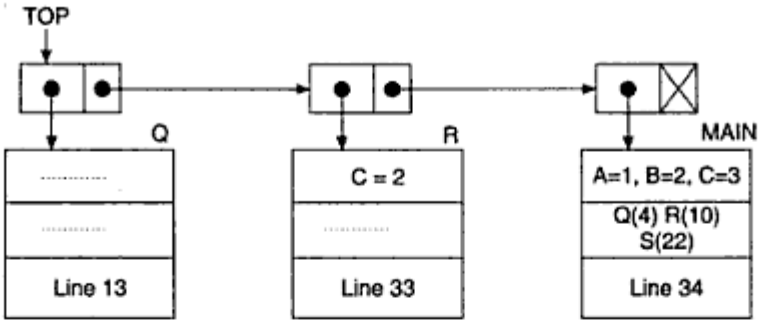


(a) Main begins its execution at line 28. Its activation record is PUSHed into the stack.

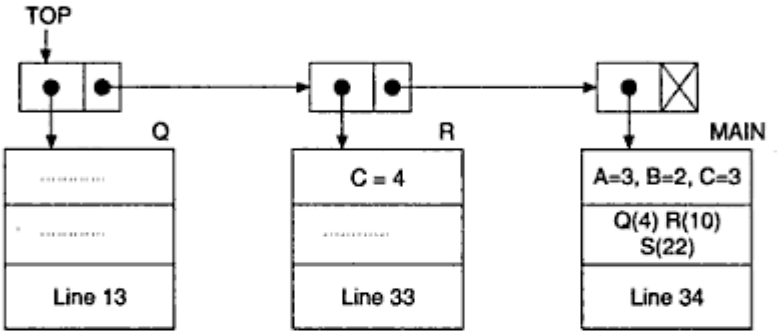


(b) During the execution of MAIN when control reaches line 32 and call of R occurs, the activation record of R is PUSHed into the stack.

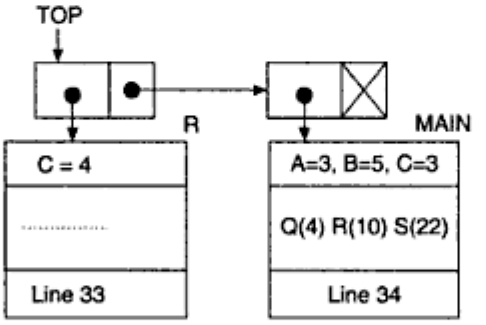
Figure 4.18 Continued.



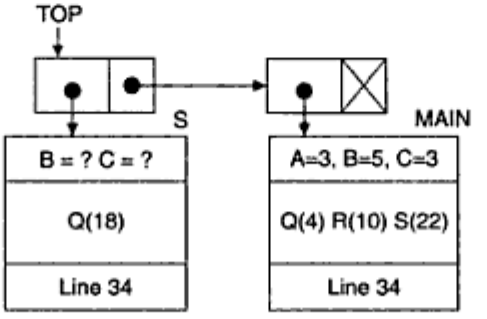
(c) Control reaches line 12, the execution of Q is initiated. This Q will be referred from the first activation record present in the stack, that is, from MAIN.



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

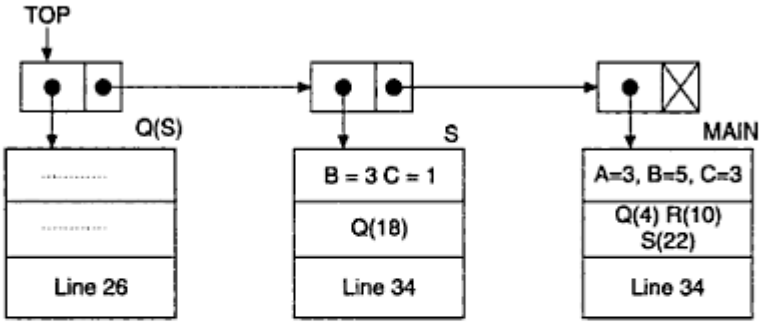


(e) Q completes its execution, control returns to line 13; A and B are referred from the activation record of MAIN. B is updated.

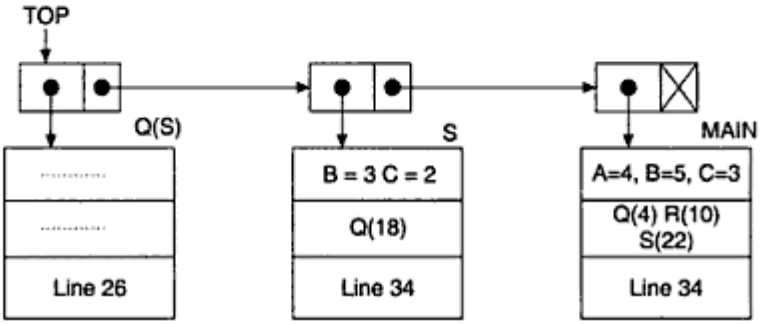


(f) When R finishes its execution, control returns to line 33; call of S occurs and the activation record of S is PUSHed into the stack.

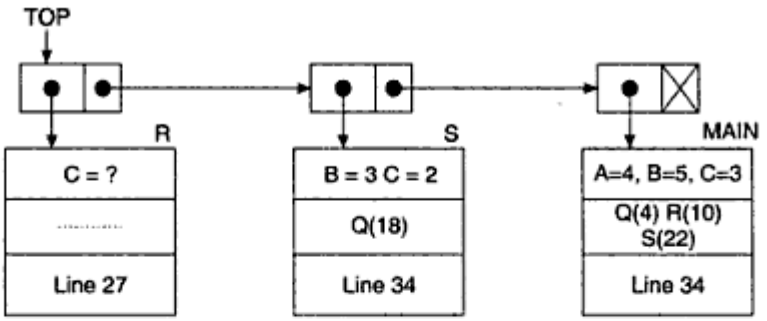
Figure 4.18 Continued.



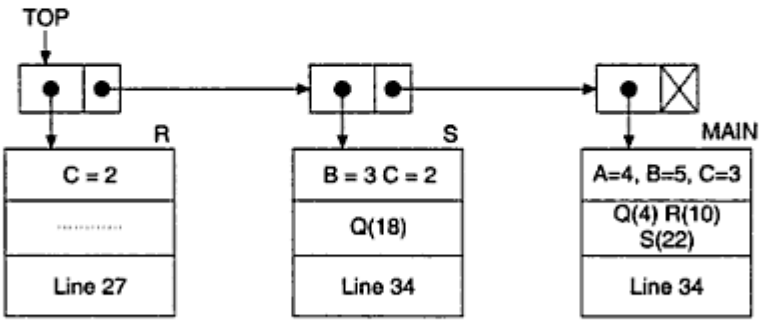
(g) Execution of S begins at line 22 (referred from MAIN) and references of B and C are from the activation record of S. When control reaches line 25, invocation of Q occurs. Reference of Q is resolved from the first occurrence, that is, from the activation of S in stack.



(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.

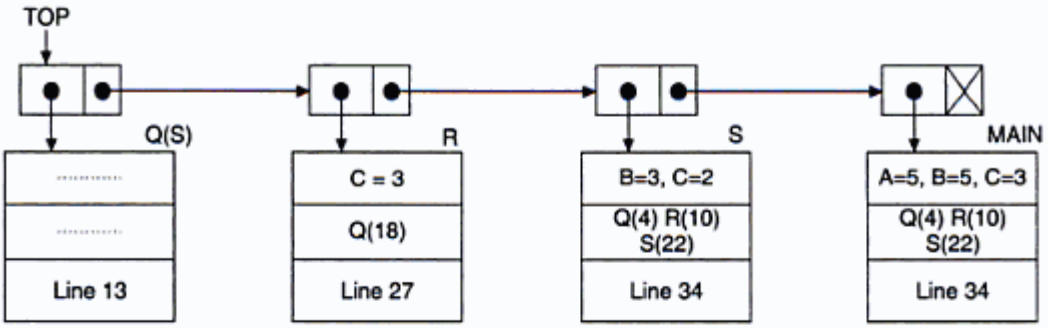


(i) Q finishes its execution and control reaches line 26, R is initiated. Its activation record is PUSHed into the stack, control jumps to line 10.

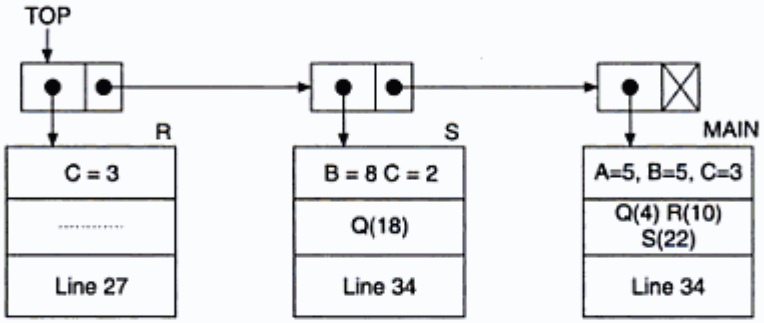


(j) During the execution of R, C is resolved from the activation record of R. Next, when Q is invoked, reference of Q is resolved from the first occurrence, that is, from S. The Q is at line 18.

Figure 4.18 Continued.



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



(i) 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.

Assignment 4.8

Consider the following recursive function:

```
Function Fibonacci (n: integer)
Begin
    If (n = 0) or (n = 1) then
        fibo = 1
    else
        fibo = Fibonacci (n – 1) + Fibonacci (n – 2)
End
```

Suppose the size of an activation record required for function Fibonacci (n) is as follows:

For local variables and passed parameter = 4 bytes
For return address = 2 bytes
For the node structure (on stack) it requires 4 bytes.

If the above function is run on a computer with a stack of 640 bytes, estimate the maximum value of n for which the memory will not underflow. Give reasons for your answer.

Assignment 4.9

Consider the following program:

```
Program TEST
    Procedure A
        Procedure B
            Procedure C
                Begin A
                .....
                .....
                End A
            Begin B
            .....
            Y : call A
            .....
            End B
        Begin A
        .....
        call B
        .....
        End A
    Begin TEST
    .....
    X : call A
    .....
    End TEST
```

Assuming suitable activation records for various procedures, draw the display structure just after the procedures:

(i) marked as *X*, and
(ii) marked as *Y*

are called.

Make sure to indicate which of the two procedures named *A* you are referring to.

4.6 PROBLEMS TO PONDER

4.1 Suppose we need to maintain two stacks of the same type of items in a program. If the two stacks are stored in separate arrays, then one stack might overflow while there are considerable unused spaces in the other. To avoid this situation it is better to maintain the two stacks in the same array as shown in Figure 4.19.

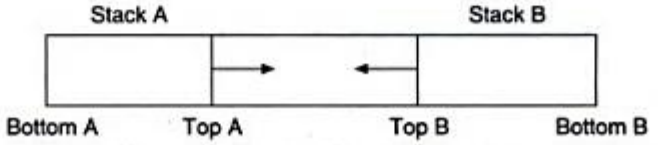


Figure 4.19 Double stack structure.

In this structure, one stack, say A, grows from one end of the array and the other stack, say B, starting from the other end, grows in the opposite direction, that is, towards the direction of A.

Write algorithms as well as the C++ program of the above-mentioned double stack structure for the following operations:

PUSHA, PUSHB, POPA and POPB.

4.2 Repeat the same concept as in Problem 4.1 but for three stacks in an array.

4.3 Draw a schematic diagram showing how to plan six stacks in a single array.

4.4 The efficient method used in evaluating a polynomial of the form

$$P_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_{n-1}x + a_n$$

is by nesting using Horner's rule, as shown below:

$$P_n(x) = (\cdots (((a_0x + a_1)x + a_2)x + \cdots + a_{n-1}) \cdots)x + a_n$$

Show how this can be carried out using a stack.

4.5 Consider the following arithmetic expression in postfix notation:

7 5 2 + * 4 1 5 – / –

(a) Find the value of the expression.
(b) Find the equivalent prefix form of the above expression.
(c) Find the value of the expression from its prefix notation.

4.6 Devise a method to convert an infix expression into its prefix form that includes (,), +, –, /, *.

4.7 Write a method to convert a postfix expression to infix for the same set of symbols as in Problem 4.6.

4.8 An arithmetic expression is given in postfix form; it is required to convert it into its equivalent prefix form. Write a method for this.

(*Hint:* You can use the method as obtained in Problems 4.6 and 4.7 and combine them suitably.)

4.9 Add the exponentiation operator to your repertoire for Problems 4.6, 4.7 and 4.8.

4.10 In our discussion, we have only been concerned with binary operators. In an arithmetic expression, there are also some operators like plus (+) and minus (–) which can be used both for binary as well as unary. Repeat Problems 4.6, 4.7 and 4.8 assuming unary plus (+) and unary minus (–) operators in the expression.

(*Hint:* It is an easy matter to distinguish different occurrences of unaries, an operator denotes a binary operator if it does not occur either at the beginning of an expression or immediately after a left parenthesis.)

4.11 Consider the following arithmetic expressions:

(a) A^ B^ C/D
(b) A* B/C

(c) $-A+B-C/A$ for $A=2, B=3, C=-4$.
(d) (A + B) * C + D/(B + A * C) + D
(e) A/B ^ C + D * E - A * C

Convert the above expression into

(i) Postfix notation
(ii) Prefix notation

4.12 List all the prime factors of the given integers in descending order. (*Hint:* Use stack.)

4.13 Devise a method that will produce all permutations of the first N integers using stack.

4.14 There is a variation in the original Euclid's algorithm for computing the greatest common divisor of two integers M and N. According to the modified Euclid's algorithm,

$$\text{GCD}(M,N)=\begin{cases}\text{GCD}(M-N,N) & \text{if } M \geq N\\ M & \text{if } N=0\\ \text{GCD}(M,N-M) & \text{if } N>M\end{cases}$$

Using only a stack, write a procedure to compute the GCD as per the modified Euclid's method.

REFERENCES

Bruno, J. L. and T. Lassagne, The generation of optimal code for a stack machine, *Journal of the ACM*, July 1975.

Donald E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, Reading, Massachusetts, 1984.

Forsythe, A.I., T.A. Keenan, et al., *Computer Science: A First Course*, John Wiley & Sons, New York, 1986.

Harrison, M.C., *Data Structures and Programming*, Glenview, Berkeley, California, 1985.

Sethi, Rajeev and J. Ullman, The generation of optimal code for arithmetic expressions, *Journal of the ACM*, Vol. 10, October 1970.

5

Queues

5.1 INTRODUCTION

A *queue* is a simple but very powerful data structure to solve numerous computer applications. Like stacks, queues are also useful to solve various system programs. Let us discuss some simple applications of queues in our everyday life as well as in computer science before undertaking the study this data structure.

Queuing in front of a counter

Suppose there are a number of customers in front of a counter to get service (say, to collect tickets or to withdraw/deposit money in a teller of a bank), Figure 5.1(a). The customers are forming a queue and they will be served in the order they arrived, that is, a customer who comes first will be served first.

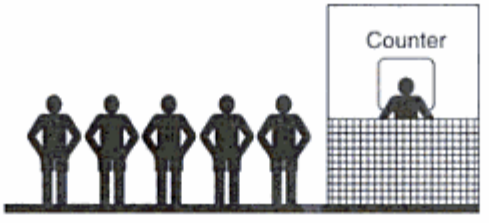


Figure 5.1(a) Queue of customers.

Traffic control at a turning point

Suppose there is a turning point in a highway where the traffic has to turn. See Figure 5.1(b). All the traffic will have wait in a line till it gets the signal for moving. On getting the 'Go' signal the vehicles will turn on a first come, first turn basis.

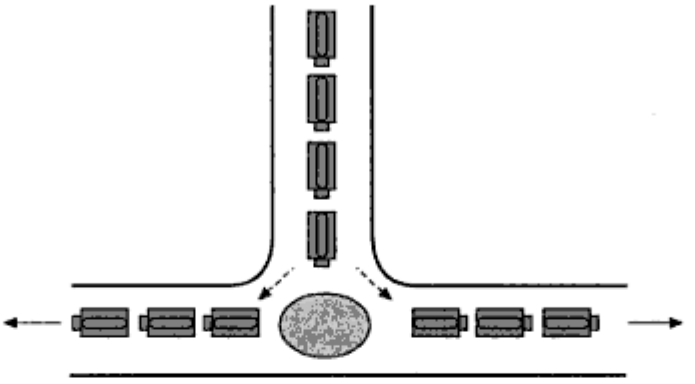


Figure 5.1(b) Traffic passing at a turning point.

Process synchronization in multi-user environment

In a multi-user environment, more than one process is handled by the monitor (operating system). See Figure 5.1(c). The three different states that a process may have are the following: READY, RUNNING, and AWAITED. A process is in the READY state when it is submitted to the system for execution. A process is in the RUNNING state if it is currently under execution. Similarly, a process will be transferred to the AWAITED state when it requires resource(s) which is/are busy. In order to synchronize the execution of processes, the monitor has to maintain two queues, namely Q1 and Q2, for READY and AWAITED states respectively where a process which entered a queue first will be exited first.

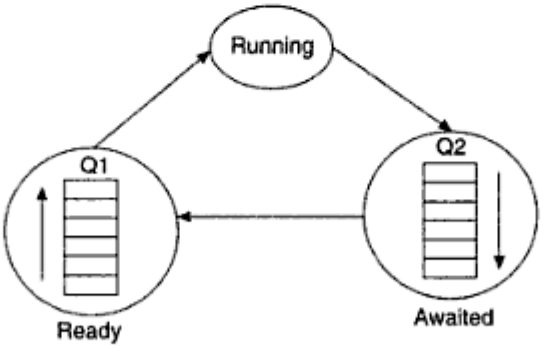


Figure 5.1(c) Queues of processes.

Resource sharing in a computer centre

In a computer centre, where resources are limited compared to the demand, users must sign a waiting register. See Figure 5.1(d). The user who has been waiting for a terminal for the longest

period of time gets hold of the resource first, then the second candidate, and so on. Here the waiting list maintains a queue and the first signed will be the first allowed.

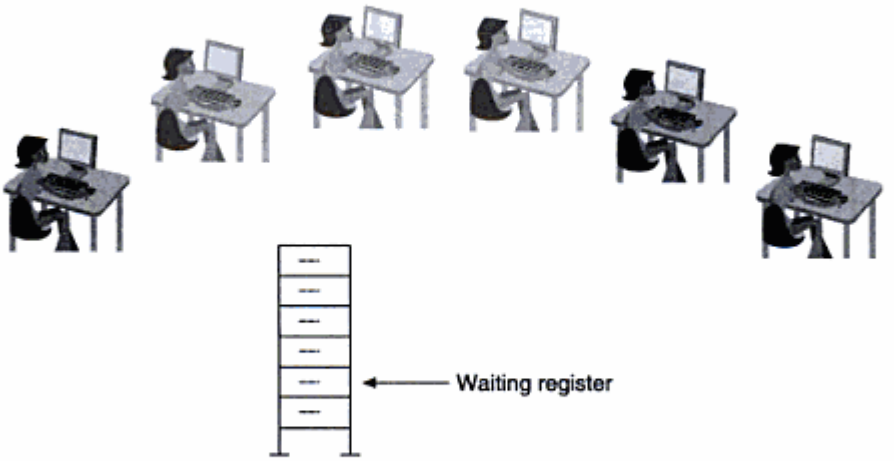


Figure 5.1(d) A waiting queue of users in a computer centre.

5.2 DEFINITION

Like a stack, a queue is an ordered collection of homogeneous data elements; in contrast with the stack, here, insertion and deletion operations take place at two extreme ends.

A queue is also a *linear* data structure like an array, a stack and a linked list where the ordering of elements is in a linear fashion. The only difference between a stack and a queue is that in the case of stack insertion and deletion (PUSH and POP) operations are at one end (TOP) only, but in a queue insertion (called ENQUEUE) and deletion (called DEQUEUE) operations take place at two ends called the REAR and FRONT of the queue, respectively. Figure 5.2 represents a model of a queue structure.

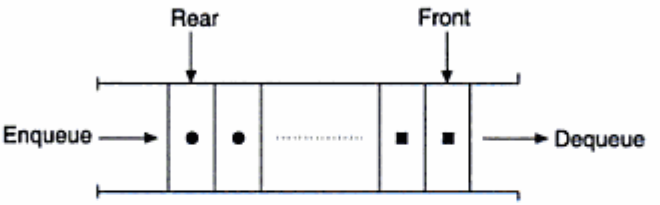


Figure 5.2 Model of a queue.

An element in a queue is termed ITEM; the number of elements that a queue can accommodate is termed LENGTH.

From the examples mentioned in Section 5.1 and the definition as stated above, it is evident that a data in a queue is processed in the same order as it had entered, that is, on a first-in, first-out basis. This is why a queue is also termed first-in first-out (FIFO).

5.3 REPRESENTATION OF QUEUES

There are two ways to represent a queue in memory:

1. Using an array
2. Using a linked list

The first kind of representation uses a one-dimensional array and it is a better choice where a queue of fixed size is required. The other representation uses a double linked list and provides a queue whose size can vary during processing.

The following two subsections describe the representation of queues in memory.

5.3.1 Representation of a Queue using an Array

A one-dimensional array, say $Q[1 \ldots N]$, can be used to represent a queue. Figure 5.3 shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

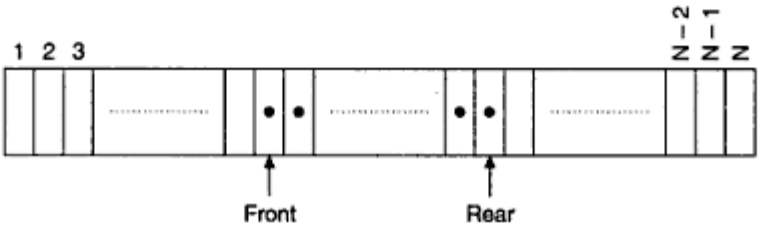


Figure 5.3 Array representation of a queue.

Three states of a queue with this representation are given below:

Queue is empty

FRONT = 0
REAR = 0 (and/or)

Queue is full

REAR = N
FRONT = 1 (when full by compact)

Queue contains elements ≥ 1

FRONT ≤ REAR
Number of elements = REAR – FRONT + 1

Now let us define the operation ENQUEUE to insert an element into a queue.

Algorithm Enqueue

Input: An element ITEM that has to be inserted.
Output: The ITEM is at the REAR of the queue.
Data structure: Q is the array representation of a queue structure; two pointers FRONT and REAR of the queue Q are known.

```
Steps:
1.  If (REAR = N) then                                  // Queue is full
2.      Print "Queue is full"
3.      Exit
4.  Else
5.      If (REAR = 0) and (FRONT = 0) then              // Queue is empty
6.          FRONT = 1
7.      EndIf
8.      REAR = REAR + 1                 // Insert the item into the queue at REAR
9.      Q[REAR] = ITEM
10. EndIf
11. Stop
```

The deletion operation DEQUEUE can be defined as given below:

Algorithm Dequeue

Input: A queue with elements. FRONT and REAR are the two pointers of the queue Q.
Output: The deleted element is stored in ITEM.
Data structures: Q is the array representation of a queue structure.

```
Steps:
1.  If (FRONT = 0) then
2.      Print "Queue is empty"
3.      Exit
4.  Else
5.      ITEM = Q[FRONT]                                 // Get the element
6.      If (FRONT = REAR)          // When the queue contains a single element
7.          REAR = 0                                    // The queue becomes empty
8.          FRONT = 0
9.      Else
10.         FRONT = FRONT + 1
11.     EndIf
12. EndIf
13. Stop
```

Let us trace the above two algorithms with a queue of size = 10. Suppose the current state of the queue is FRONT = 8, REAR = 9. Ten operations are requested as under:

1. DEQUEUE	2. ENQUEUE	3. ENQUEUE
4. DEQUEUE	5. DEQUEUE	6. DEQUEUE
7. ENQUEUE	8. ENQUEUE	9. DEQUEUE
10. DEQUEUE		

Figure 5.4 presents the status of the queue when these operations are carried out.

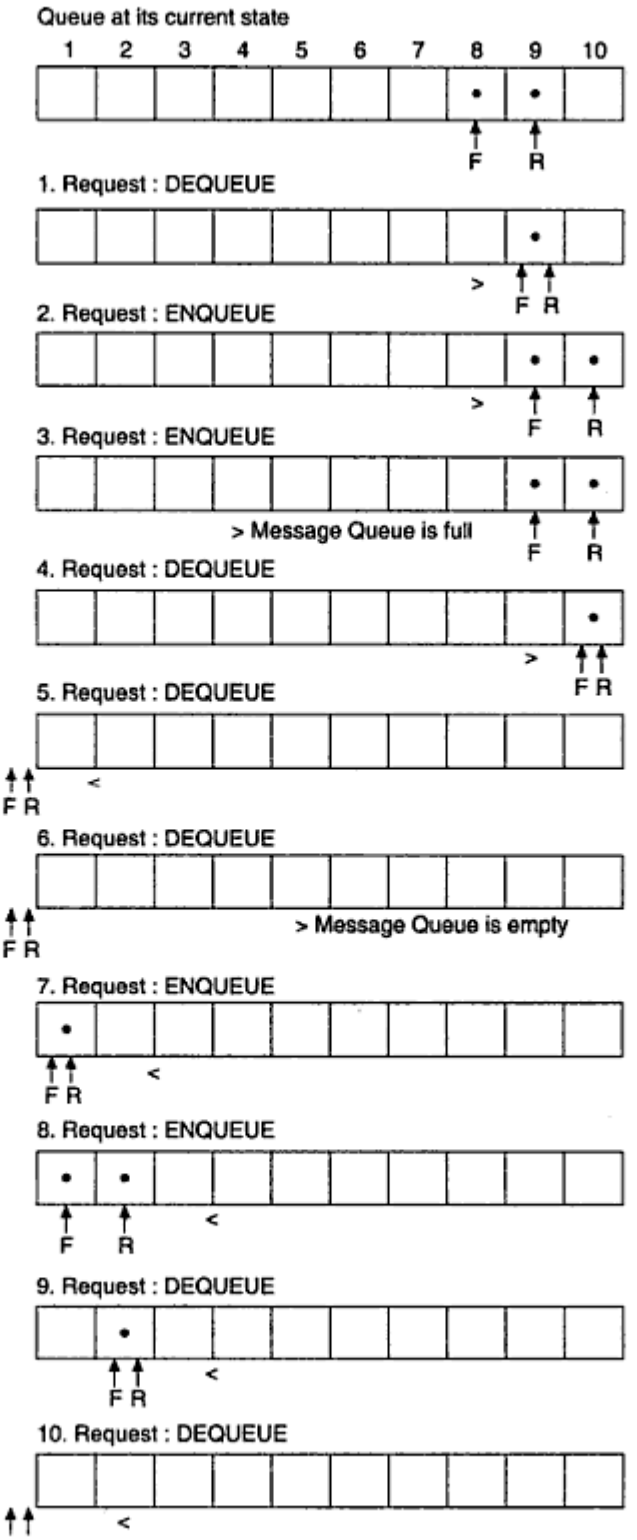


Figure 5.4 Operations on a queue.

There is one potential problem with this representation. From Figure 5.4, we can see that with this representation, a queue may not be full, still a request for insertion operation may be denied. For example, on request (3) (in Figure 5.4) 8 rooms are available but insertion is not possible as the insertion pointer reaches the end of the queue. This is simply a wastage of the storage. This type of representation can be recommended for an application where the queue is emptied at certain intervals.

Assignment 5.1

The algorithm *Enqueue* may fail even though there is memory space available. One way to avoid this problem is to rewrite the algorithm *Enqueue* and *Dequeue*. Two solutions are suggested as given below:

Suggestion 1: Rewriting the algorithm *Enqueue*.

Whenever the REAR pointer gets to the end of the queue (Figure 5.5), test whether the pointer FRONT is at location 1 or not; if not, shift all the elements so that they are wrapped from the beginning and thus make room for a new item.

Suggestion 2: Rewriting the algorithm *Dequeue*.

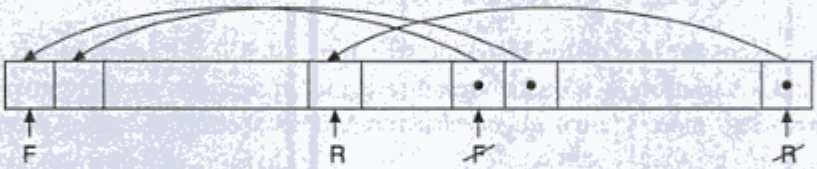


Figure 5.5 Shifting elements when R reaches to the end.

After the end of each deletion, all the elements at the trail are shifted once towards the front; here the idea is to fix the FRONT pointer always at 1. The queue which follows such operations is termed a *dynamic queue*. Rewrite operations ENQUEUE and DEQUEUE for a dynamic queue.

5.3.2 Representation of a Queue using a Linked List

One more limitation of a queue, other than the inadequate service of insertion represented with an array, is the rigidness of its length. In several applications, the length of the queue cannot be predicated before and it varies abruptly. To overcome this problem, another preferable representation of a queue is with a linked list. Here, we select a double linked list which allows us to move both ways. Figure 5.6 shows the double, linked list representation of a queue. The pointers FRONT and REAR point the first node and the last node in the list.

Figure 5.6 A double linked list representation of a queue.

Two states of the queue, either empty or containing some elements, can be judged by the following tests:

Queue is empty

FRONT = REAR = HEADER

HEADER→RLINK = NULL

Queue contains at least one element

HEADER→RLINK ≠ NULL

The insertion and deletion operations are straightforward and the same as in the algorithm *InsertEnd_DL* (for *Enqueue*) and algorithm *DeleteFront_DL* (for *Dequeue*); these two algorithms are already defined in Section 3.4 of Chapter 3.

Assignment 5.2

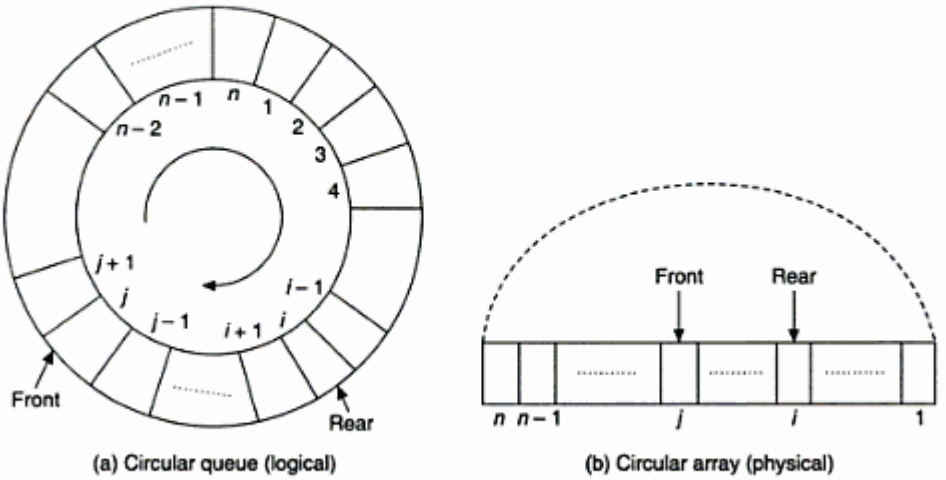
Explore the possibilities of representing a queue using an ordinary single linked list or circularly single linked list.

5.4 VARIOUS QUEUE STRUCTURES

So far we have discussed two different queue structures, that is, either using an array or using a linked list (and a variation of a queue structure using an array as an assignment). Other than these, there are some more known queue structures. This section discusses them.

5.4.1 Circular Queue

As pointed at the end of Section 5.3.1, for a queue represented using an array when the REAR pointer reaches the end, insertion will be denied even if room is available at the front. One way to avoid this is to use a circular array. Physically, a circular array is the same as an ordinary array, say $A[1 \ldots N]$, but logically it implies that $A[1]$ comes after $A[N]$ or after $A[N]$, $A[1]$ appears. Figure 5.7 shows logical and physical views of a circular array.



(a) Circular queue (logical) (b) Circular array (physical)

Figure 5.7 Logical and physical views of a circular queue.

The principle underlying the representation of a circular array is as stated below:

Both pointers will move in a clockwise direction. This is controlled by the MOD operation; for example, if the current pointer is at i then shift to the next location will be i MOD LENGTH + 1, $1 \leq i \leq$ LENGTH (where LENGTH is the queue length). Thus, if i = LENGTH (that is at the end), then the next position for the pointer is 1.

With this principle the two states of the queue regarding, i.e. empty or full, will be decided as follows:

Circular queue is empty

FRONT = 0

REAR = 0

Circular queue is full

FRONT = (REAR MOD LENGTH) + 1

The following two algorithms describe the insertion and deletion operations on a circular queue.

Algorithm Enqueue_CQ

Input: An element ITEM to be inserted into the circular queue.

Output: Circular queue with the ITEM at FRONT, if the queue is not full.

Data structures: CQ be the array to represent the circular queue. Two pointers FRONT and REAR are known.

Steps:

```
1.  If (FRONT = 0) then                          // When the queue is empty
2.      FRONT = 1
3.      REAR = 1
4.      CQ[FRONT] = ITEM
5.  Else                                         // Queue is not empty
6.      next = (REAR MOD LENGTH) + 1
7.      If (next ≠ FRONT) then                   // If the queue is not full
8.          REAR = next
9.          CQ[REAR] = ITEM
10.     Else
11.         Print "Queue is full"
12.     EndIf
13. EndIf
14. Stop
```

Algorithm Dequeue_CQ

Input: A queue CQ with elements. Two pointers FRONT and REAR are known.

Output: The deleted element is ITEM if the queue is not empty.

Data structures: CQ is the array representation of circular queue.

Steps:

```
1.  If (FRONT = 0) then
2.      Print "Queue is empty"
3.      Exit
4.  Else
5.      ITEM = CQ[FRONT]
6.      If (FRONT = REAR) then              // If the queue contains a single element
7.          FRONT = 0
8.          REAR = 0
9.      Else
10.         FRONT = (FRONT MOD LENGTH) + 1
11.     EndIf
12. EndIf
13. Stop
```

In order to trace these two algorithms, let us consider a circular queue of LENGTH = 4. The following operations are requested. Different states of the queue while processing these requests are illustrated in Figure 5.8.

1. ENCQUEUE (A)
2. ENCQUEUE (B)
3. ENCQUEUE (C)
4. ENCQUEUE (D)
5. DECQUEUE
6. ENCQUEUE (E)
7. DECQUEUE
8. ENCQUEUE (F)
9. DECQUEUE
10. DECQUEUE
11. DECQUEUE
12. DECQUEUE

Assume that initially the queue is empty, that is, FRONT = REAR = 0.

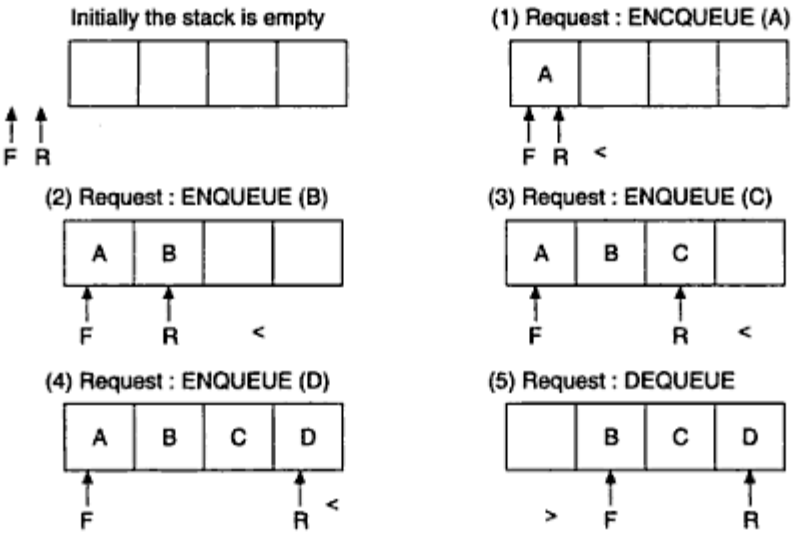


Figure 5.8 Continued.

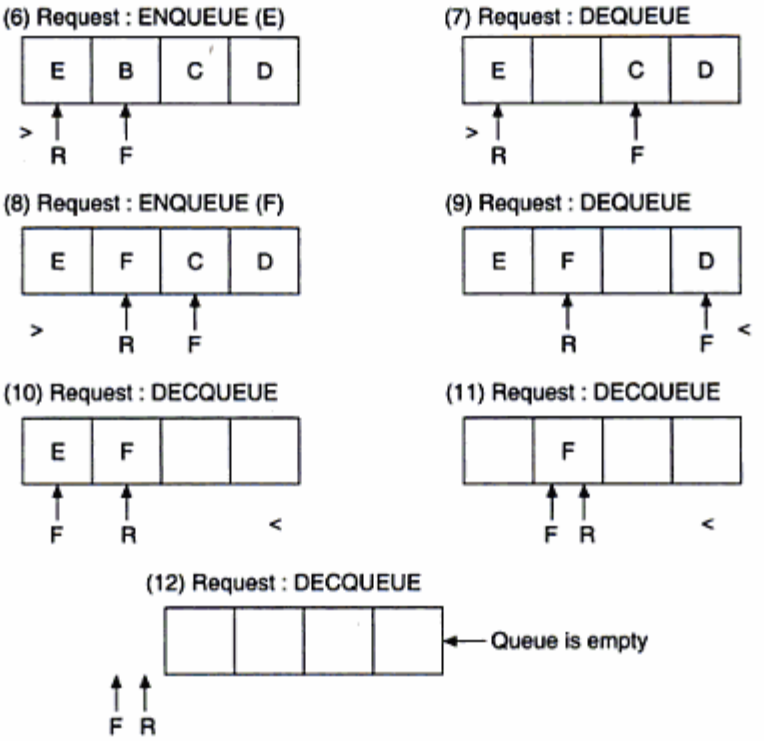


Figure 5.8 Tracing insertion and deletion operations on a circular queue.

Assignment 5.3

The following two algorithms are proposed for a circular queue.

Algorithm 1 Enqueue1_CQ

Steps:

1. **If** ((REAR + 1) MOD LENGTH = FRONT) **then**
2. **Print** "Queue is full"
3. **Exit**
4. **Else**
5. CQ[REAR] = ITEM
6. REAR = (REAR + 1) MOD LENGTH
7. **EndIf**
8. **Stop**

Algorithm 2 Dequeue2_CQ

Steps:

1. **If** (REAR = FRONT) **then**
2. **Print** "Queue is empty"
3. **Exit**
4. **Else**

(Contd.)

5. ITEM = CQ[FRONT]
6. FRONT = (FRONT + 1) MOD LENGTH
7. **EndIf**
8. **Stop**

(a) Decide the queue that fits with these algorithms.

(b) Do the two algorithms utilize properly the whole of available memory? If your answer is 'NO', then devise the necessary modification so that this deficiency is overcome.

5.4.2 Deque

Another variation of the queue is known as deque (may be pronounced 'deck'). Unlike a queue, in deque, both insertion and deletion operations can be made at either end of the structure. Actually, the term deque has originated from **d**ouble **e**nded **que**ue. Such a structure is shown in Figure 5.9.

Figure 5.9 A deque structure.

It is clear from the deque structure that it is a general representation of both stack and queue. In other words, a deque can be used as a stack as well as a queue.

There are various ways of representing a deque on the computer. One simpler way to represent it is by using a double linked list. Another popular representation is using a circular array (as used in a circular queue).

The following four operations are possible on a deque which consists of a list of items:

1. Push_DQ(ITEM): To insert ITEM at the FRONT end of a deque.
2. Pop_DQ( ): To remove the FRONT item from a deque.
3. Inject(ITEM): To insert ITEM at the REAR end of a deque.
4. Eject( ): To remove the REAR ITEM from a deque.

These operations are described for a deque based on a circular array of length LENGTH. Let the array be DQ[1 ... LENGTH].

Algorithm Push_DQ

Input: ITEM to be inserted at the FRONT.
Output: Deque with a newly inserted element ITEM if it is not full already.
Data structures: DQ being the circular array representation of a deque.

Steps:

```
1.  If (FRONT = 1) then                                   // If FRONT is at extreme left
2.      ahead = LENGTH
3.  Else                                  // If FRONT is at extreme right or the deque is empty
4.      If (FRONT = LENGTH) or (FRONT = 0) then
5.          ahead = 1
6.      Else
7.          ahead = FRONT – 1                          // FRONT is at an intermediate position
8.      EndIf
9.      If (ahead = REAR) then
10.         Print "Deque is full"
11.         Exit
12.     Else
13.         FRONT = ahead                                                  // Push the ITEM
14.         DQ[FRONT] = ITEM
15.     EndIf
16. EndIf
17. Stop
```

Algorithm Pop_DQ

/* This algorithm is the same as the algorithm *Dequeue_CQ* */

Algorithm Inject

/* This algorithm is the same as the algorithm Enqueue_CQ */

Algorithm Eject_DQ

Input: A deque with elements in it.
Output: The item is deleted from the REAR end.
Data structures: DQ being the circular array representation of deque.

Steps:

```
1.  If (FRONT = 0) then
2.      Print "Deque is empty"
3.      Exit
4.  Else
5.      If (FRONT = REAR) then                    // The deque contains single element
6.          ITEM = DQ[REAR]
7.          FRONT = REAR = 0                                  // Deque becomes empty
8.      Else
9.          If (REAR = 1) then                                // REAR is at extreme left
10.             ITEM = DQ[REAR]
11.             REAR = LENGTH
12.         Else
13.             If (REAR = LENGTH) then                       // REAR is at extreme right
14.                 ITEM = DQ[REAR]
```

```
15.                REAR = 1
16.             Else                          // REAR is at an intermediate position
17.                ITEM = DQ[REAR]
18.                REAR = REAR - 1
19.             EndIf
20.          EndIf
21.       EndIf
22.    EndIf
23.    Stop
```

There are, however, two known variations of deque:

1. Input-restricted deque
2. Output-restricted deque.

These two types of variations are actually intermediate between a queue and a deque. Specifically, an *input-restricted deque* is a deque which allows insertions at one end (say REAR end) only, but allows deletions at both ends. Similarly, an *output-restricted deque* is a deque where deletions take place at one end only (say FRONT end), but allows insertions at both ends. Figure 5.10 represents two such variations of deque.

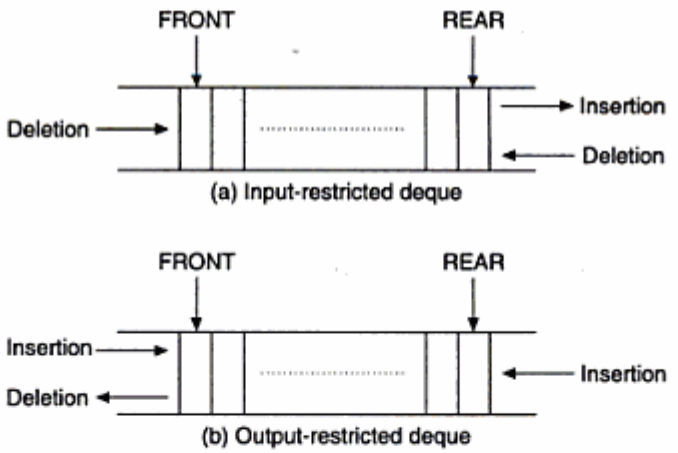


Figure 5.10 Types of deque.

Assignment 5.4

1. Using the linked list representation of a deque, obtain the four operations PUSHQ( ), POPQ( ), INJECT( ), and EJECT( ).
2. Using the circular array representation of a deque, obtain the following:
   (a) Insertion operation into an input-restricted deque.
   (b) Deletion operations from an input-restricted deque.
3. Repeat problem 2 for an output-restricted deque.

5.4.3 Priority Queue

A *priority queue* is another variation of queue structure. Here, each element has been assigned a value, called the *priority* of the element, and an element can be inserted or deleted not only at the ends but at any position on the queue. Figure 5.11 shows a priority queue.

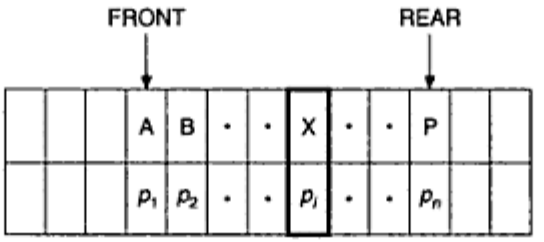


Figure 5.11 View of a priority queue.

With this structure, an element X of priority p_i may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority, that is, instead of adding it after the REAR it may be inserted at an intermediate position dictated by its priority value.

Note that the name priority queue is a misnomer in the sense that the data structure is not a queue as per the definition; a priority queue does not strictly follow the first-in first-out (FIFO) principle which is the basic principle of a queue. Nevertheless, the name is now firmly associated with this particular data type. However, there are various models of priority queue known in different applications. Let us consider a particular model of priority queue.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Here, process means two basic operations namely insertion or deletion. There are various ways of implementing the structure of a priority queue. These are:

(i) Using a simple/circular array
(ii) Multi-queue implementation
(iii) Using a double linked list
(iv) Using heap tree.

We will now see what each of these implementations is. (Heap tree implementation of priority queue will be discussed in Chapter 7 of this text.)

Priority queue using an array

With this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the REAR end as usual. The deletion operation will then be performed in either of the two following ways:

(a) Starting from the FRONT pointer, traverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front-most element, shift all its trailing elements after the deleted element one stroke each to fill up the vacant position (see Figure 5.12).

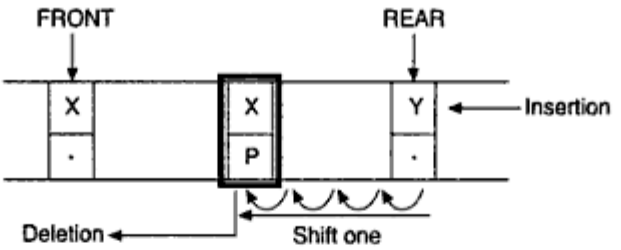


Figure 5.12 Deletion operation in an array representation of a priority queue.

This implementation, however, is very inefficient as it involves searching the queue for the highest priority element and shifting the trailing elements after the deletion. A better implementation is as follows:

(b) Add the elements at the REAR end as earlier. Using a stable sorting algorithm*, sort the elements of the queue so that the highest priority element is at the FRONT end. When a deletion is required, delete it from the FRONT end only (see Figure 5.13).

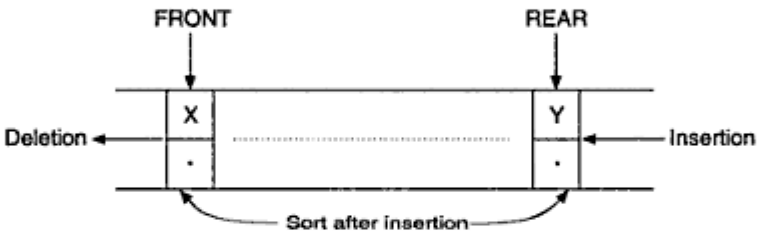


Figure 5.13 Another array implementation of a priority queue.

The second implementation is comparatively better than the first one; here the only burden is to sort the elements. The algorithms of the above two implementations are left as assignments to the reader.

Multi-queue implementation

This implementation assumes N different priority values. For each priority p_i there are two pointers F_i and R_i corresponding to the FRONT and REAR pointers respectively. The elements between F_i and R_i are all of equal priority value p_i. Figure 5.14 represents a view of such a structure.

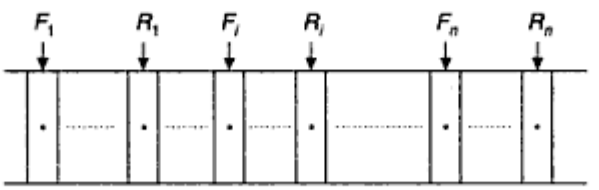


Figure 5.14 Multi-queue representation of a priority queue.

With this representation, an element with priority value p_i will consult F_i for deletion and R_i for insertion. But this implementation is associated with a number of difficulties:

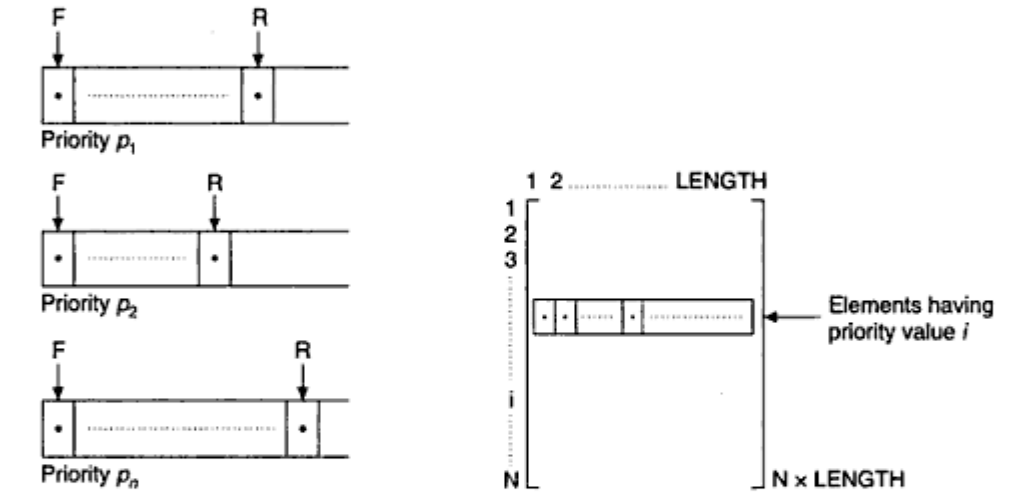
(i) It may lead to a huge shifting in order to make room for an item to be inserted.

(ii) A large number of pointers are involved when the range of priority values is large.

In addition to the above, there are two other techniques to represent a multi-queue, which are shown in Figures 5.15(a) and 5.15(b).

It is clear from Figure 5.15(a) that for each priority value a simple queue is to be maintained. An element will be added into a particular queue depending on its priority value.

The priority queue as shown in Figure 5.15(b) is in some way better than the multi-queue with multiple queues. Here one can get rid of maintaining several pointers for FRONT and REAR in several queues. A multi-queue with multiple queues has one advantage that one can have different queues of arbitrary length. In some applications, it is seen that the number of occurrences of elements with some priority value is much larger than the other value, thus demanding a queue of larger size.



(a) Multiple queue with simple queues

(b) Multiple queue with a matrix

Figure 5.15 Multi-queue implementation with multiple simple queues and matrix.

Both the above representations are not economic from the memory utilization point of view; much of the memory space remains vacant.

Algorithms for insertion and deletion operations for multi-queue implementation are left as exercises for the student.

Linked list representation of a priority queue

This representation assumes the node structure as shown in Figure 5.16. LLINK and RLINK are two usual link fields, DATA is to store the actual content and PRIORITY is to store the priority value of the item. We will consider FRONT and REAR as two pointers pointing the first and last nodes in the queue, respectively. Here all the nodes are in sorted order according to the priority values of the items in the nodes. The following is an instance of a priority queue:

*A sorting algorithm is stable if the relative positions of two identical items remain the same in the unsorted and sorted list (for details, see Chapter 10).

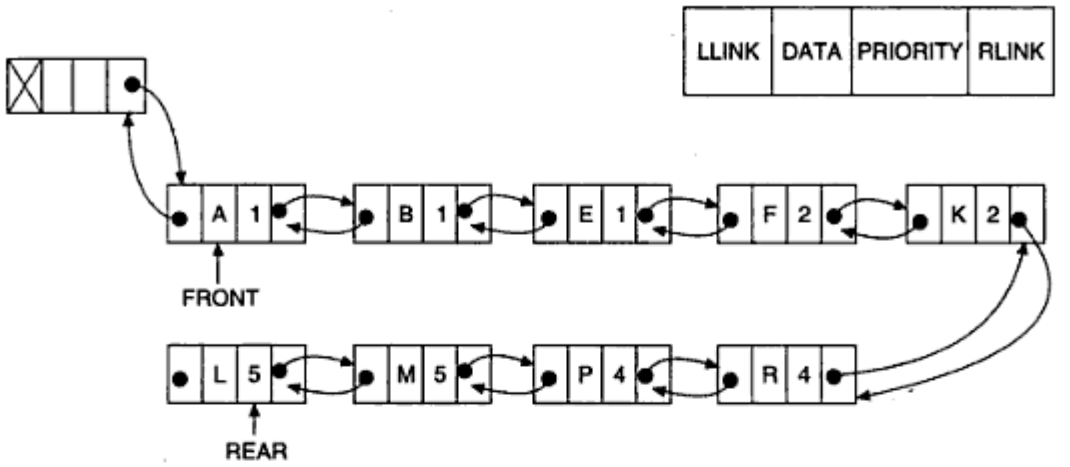


Figure 5.16 Linked list representation of a priority queue.

With this structure, to delete an item having priority p, the list will be searched starting from the node under pointer REAR and the first occurring node with PRIORITY = p will be deleted. Similarly, to insert a node containing an item with priority p, the search will begin from the node under the pointer FRONT and the node will be inserted before a node found first with priority value p, or if not found then before the node with the next priority value. The following two algorithms *Insert_PQ* and *Delete_PQ* are used to implement the insertion and deletion operations on a priority queue.

Algorithm Insert_PQ

Input: The ITEM and its priority P value of a node that is to be inserted.
Output: A new node inserted.
Data structures: Linked list structure of priority queue; HEADER as the pointer to the header.

Steps:

```
1.  ptr = HEADER                                          // Start from the first node
2.  new = GetNode(NODE)                                   // Avail a new node
3.  new→DATA = ITEM                                       // Get initialized the node with ITEM
4.  new→PRIORITY = P
5.  While (ptr→RLINK ≠ NULL) and (ptr→PRIORITY < P) do    // Search for the position
6.      ptr = ptr→RLINK
7.  EndWhile
8.  If (ptr→RLINK = NULL) then      // If the list is empty or the item is with the largest
                                                          priority value
9.      ptr→RLINK = new
10.     new→LLINK = ptr
11.     new→RLINK = NULL                                  // The node is inserted as the last node
12.     REAR = new
13. Else
```

(Contd.)

```
14.     If (ptr→PRIORITY ≥ P) then                    // First occurrence is found
15.         ptr1 = ptr→.LLINK                         // Insert the new node
16.         ptr1→RLINK = new                          // Before the node with priority > P
17.         new→RLINK = ptr
18.         ptr→LLINK = new
19.         new→LLINK = ptr1
20.     EndIf
21. EndIf
22. FRONT = HEADER→RLINK                              // Set the FRONT pointer
23. Stop
```

Similarly, the algorithm for deletion can be described as follows:

Algorithm Delete_PQ

Input: The priority P of the element which has to be deleted.
Output: The element that is being deleted.
Data structures: Linked list structure of priority queue; HEADER as the pointer to the header.

Steps:

```
 1. If (REAR = NULL) then
 2.     Print "Queue is empty"
 3.     Exit
 4. Else
 5.     ptr = REAR
 6.     While (ptr→PRIORITY > P) and (ptr ≠ HEADER) do
 7.         ptr = ptr→LLINK
 8.     EndWhile
 9.     If (ptr = HEADER) or (ptr→PRIORITY < P)
10.         Print "No item with priority", P
11.         Exit
12.     Else
13.         If (ptr→priority = P) then
14.             ptr1 = ptr→LLINK
15.             ptr2 = ptr→RLINK
16.             If (ptr = REAR)                       // If the last node to be deleted
17.                 REAR = ptr1
18.                 ptr1→RLINK = NULL
19.             Else                                  // Other than last node
20.                 ptr1→RLINK = ptr2                 // Deleted
21.                 ptr2→LLINK = ptr1
22.             EndIf
23.         EndIf
```

(Contd.)

```
24.      EndIf
25.      item = ptr→DATA
26.      ReturnNode(item)
27.   EndIf
28.   Stop
```

Assignment 5.5

(a) Describe the insertion and deletion algorithms for the following priority queue structures:

(i) Queue represented with a single large array.

(ii) Multi-queue representation with a single large array.

(iii) Multi-queue representation with multiple simple queue.

(iv) Multi-queue with matrix representation.

(b) The algorithms *Insert_PQ* and *Delete_PQ* assume that the highest priority element will be deleted first.

Modify the two algorithms so that the lowest priority element will be deleted first.

5.5 APPLICATIONS OF QUEUES

Numerous applications of queue structures are known in computer science. One major application of queues is in simulation. Another important application of queues is observed in the implementation of various aspects of an operating system. A multiprogramming environment uses several queues to control various programs. And, of course, queues are very much useful to implement various algorithms. For example, various scheduling algorithms are known to use varieties of queue structures.

This section highlights a few applications and then illustrates how powerful queues are to solve different problems.

5.5.1 Simulation

Simulation is modelling of a real-life problem, or in other words, it is the model of a real-life situation in the form of a computer program. The main objective of the simulation program is to study the real-life situation under the control of various parameters which affect the real problem, and is a research interest of system analysts or operation research scientists. Based on the results of simulation, the actual problem can be solved in an optimized way.

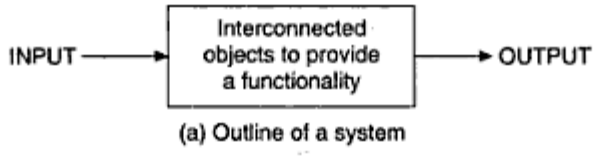
Another advantage of simulation is to experiment the danger area. For example, areas such as military operations are safer to simulate than to field test, simulation being free from any risk as well as inexpensive.

Simulation is a classical area where queues can be applied. Before discussing simulated modelling, let us study a few terms related to it.

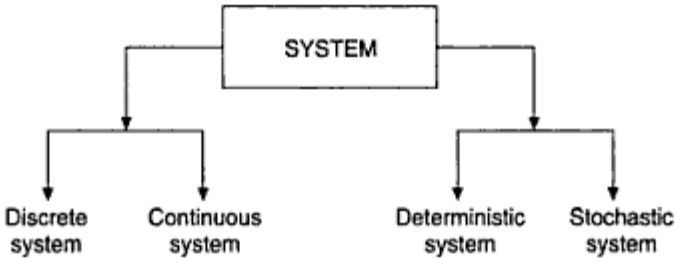
Any process or situation that is to be simulated is called a *system.* A system is a collection of interconnected objects which accepts zero or more inputs and produces at least one output,

(see Figure 5.17(a)). For example, a computer program is a system where instructions are the interconnected objects and inputs or initialization values are the inputs and the results obtained during the execution constitute the output. Similarly, a ticket reservation counter is also a system (you can easily guess about its input, output and functionality). Note that a system can be composed of one or more smaller system(s).

A system can be divided into different types as shown in Figure 5.17(b). A system is discrete if the input/output parameters are of discrete values. For example, a customer arriving at a ticket reservation counter is a discrete parameter, whereas water flowing through a pipe to a reservoir is an example of a continuous system since the parameter is of the continuous type.



(a) Outline of a system



(b) Classification of systems

Figure 5.17 System and simulation.

A system is deterministic if from a given set of inputs and initial conditions of the system, the final outcome can be predicted. For example, a program to calculate the factorial of an integer is a deterministic system. On the other hand, a stochastic system is based on randomness: its behaviour cannot be predicted before hand. As another example, the number of customers waiting in front of a ticket reservation counter at any instant cannot be forecasted. There may be some systems which are intermixes of both deterministic and stochastic.

After getting an idea about the various types of systems, let us define various kinds of simulation models. There are two kinds of simulation models: *event-driven simulation* and *time-driven simulation*; these are decided according to how the state of a system changes. In the case of time-driven simulation, the systems changes in its states with the change time, and in event-driven simulation, the system changes its state whenever a new event reaches the system or exits from the system.

Now let us consider a system, its model for simulation study and then the application of queues in it. Consider a system as a ticket selling centre. There are two kinds of tickets available, namely T1 and T2, which customers are to purchase. Two counters C1 and C2 are available (Figure 5.18). Also assume that the time required for issuing a ticket of T1 and T2 are t_1 and t_2, respectively. Two queues Q1 and Q2 are possible for the counters C1 and C2, respectively. With this description of the system, two models are proposed:

Model 1

- Any counter can issue both types of tickets.
- A customer on arrival joins the queue which has a lesser number of customers; if both queues are equally crowded, then to Q1, the queue of counter C1.

Model 2

- Two counters are earmarked, say C1 for selling T1 only and C2 for selling T2 only.
- A customer on arrival goes to either queue Q1 or queue Q2, depending on whether the ticket T1 or T2 is to be purchased.

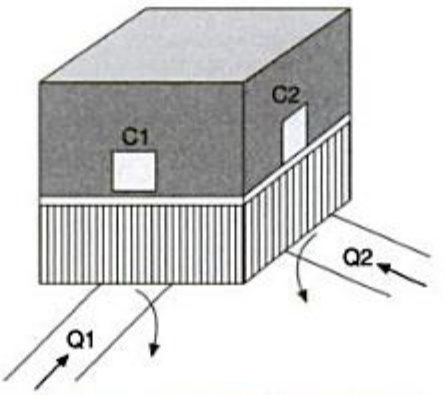


Figure 5.18 A ticket selling counter.

To simplify the simulation model, the underlying assumptions are made:

1. Queue lengths are infinite.
2. One customer in a queue is allowed to purchase one ticket only.
3. Let λ_1 and λ_2 be the mean arrival rates of customers for tickets T1 and T2, respectively. The values for λ_1 and λ_2 will be provided by the system analyst.
4. Let us consider the *discrete probability distribution* (also called *Poisson distribution*) for the arrival of customers to the centre. Poisson distribution gives a probability function

$$P(t) = 1 - e^{-\lambda t}$$

where $P(t)$ = the probability that the next customer arrives at time t, and λ = the mean arrival rate. Thus, if we assume N to be the total population of customers in a day, then

$$N_1 = N_1 P(t) = N_1(1 - e^{-\lambda_1 t}) \tag{5.1}$$

is the number of customers who arrived at the centre for ticket T1 at time t, and $N_2 = N_2 P(t) = N_2(1 - e^{-\lambda_2 t})$ is the number of customers who arrived the centre for ticket T2 at time t.

5. A clock is maintained with an initial value (to dictate the opening and closing of counters) when the counter is made available to the customer, etc.

With these basic assumptions and definitions, the proposed simulation model may be termed the *discrete deterministic time-driven* simulation model. Before describing the algorithms, let us assume the following keywords that will be assumed in our algorithm:

1. An abstract data type CUSTOMER to simulate a customer with the following components:
   CUSTOMER
   - `TicketReq`—the customer requesting for the type of ticket
   - t_1—time of arrival of the customer
   - t_2—time of departure of the customer
   - t—amount of time served when a customer is currently under service
2. SETCUSTOMER1( ) and SETCUSTOMER2( ) are the two functions which set the data for `TicketReq` as T1 and T2, t as τ_1 and τ_2 for the object of CUSTOMER. τ_1 in each case will be set by the current value of the CLOCK.
3. LENGTH(Q) is a function to return the length of a queue Q.
4. ENQUEUE(CUSTOMER) and DEQUEUE( ) are the two functions to add a customer into the queue and to remove a customer from the queue, respectively.
5. SEECUSTOMER(Q) function is to attend a customer which is at front of a queue Q by the counterman. This function is actually to read the `TicketReq` and t for a customer C.
6. EMPTY(Q) returns 'TRUE' if the queue Q is empty else 'FALSE'.

Now, we are in a position to write the algorithms for Model1 and Model2, which are stated as follows:

Algorithm SimulationModel_1

Inputs:
λ_1 = Mean arrival rate of customers for ticket T_1
λ_2 = Mean arrival rate of customers for ticket T_2
N = Total population assumed
OPEN_HOURS = Total time duration for which the centre remains open in a day
t_1 = Time required to process for a ticket T_1
t_2 = Time required to process for a ticket T_2

Outputs:
λ_{Q1} = Average queue length of the queue Q_1
λ_{Q2} = Average queue length of the queue Q_2
Γ_{T_1} = Average waiting time of the customers for ticket T_1
Γ_{T_2} = Average waiting time of the customers for ticket T_2
Γ_{C_1} = Total service time that the counter C_1 served
Γ_{C_2} = Total service time that the counter C_2 served

Data structures: Two queues Q1 and Q2 represented with double linked lists. An abstract data type CUSTOMER.

Steps:

```
/* INITIALIZATION */
1.  Q1 and Q2 are initialized as empty
2.  CLOCK to maintain the current time is set at zero
3.  L1 = L2 = 0                          // These are required for average lengths of Q1 and Q2
4.  WT1 = WT2 = 0                        // For storing the waiting times of customers for T1 and T2
5.  NT1 = NT2 = 0                        // Number of customers served with tickets T1 and T2
6.  NC1T1 = NC1T2 = 0                    // Number of customers served in counter C1
                                            for ticket T1, T2
7.  NC2T2 = NC2T1 = 0                    // Number of customers served in counter C2
                                            for ticket T1, T2
    /*START SIMULATION FOR A DAY*/
9.  While (CLOCK < OPEN_HOURS) do
        /*Generate population of customers*/
10.     N1 = N * (1 – e^λ1 * CLOCK)
11.     N2 = N * (1 – e^λ2 * CLOCK)
    /*Add customers into the queues*/
12.     l1 = Length(Q1)                  // Current lengths of the queue Q1
13.     l2 = Length (Q2)                 // Current lengths of the queue Q2
14.     If (N1 > N2) then
15.         While (N2 > 0) do
16.             c = SetCustomer1()       // Customer for T1
17.             If (l1 ≤ l2) then
18.                 Enqueue1(c)          // Add to the smaller queue
19.                 l1 = l1 + 1
20.             Else
21.                 Enqueue2 (C)
22.                 l2 = l2 + 1
23.             EndIf
24.             N1 = N1 – 1
25.             c = SetCustomer2 ()      // Customer for T2
26.             If (l1 ≤ l2) then        // Add to the smaller queue
27.                 Enqueue1 (c)
28.                 l1 = l1 + 1
29.             Else
30.                 Enqueue1 (c)
31.                 l2 = l2 + 1
32.             EndIf
33.             N2 = N2 – 1
34.         EndWhile
```

(Contd.)

```
35.        While (N1 > 0) do      // Add the remaining customer for ticket of type T2
36.            c = SetCustomer1 ()
37.            If (l1 ≤ l2) then                          // Add to the smaller queue
38.                Enqueue1 (c)
39.                l1 = l1 + 1
40.            Else
41.                Enqueue2 (c)
42.                l2 = l2 + 1
43.            EndIf
44.            N1 = N1 – 1
45.        EndWhile
46.    Else            // Customers for T2 are more than those for T1, that is, N2 > N1
47.        While (N1 > 0) do    // One customer from T1 type and other from T2 type
48.            c = SetCustomer1 ()
49.            If (l1 < l2) then
50.                Enqueue1 (c)
51.                l1 = l1 + 1
52.            Else
53.                Enqueue2 (c)
54.                l2 = l2 + 1
55.            EndIf
56.            N1 = N1 – 1
57.            c = SetCustomer2 ()
58.            If (l1 < l2) then
59.                Enqueue1 (c)
60.                l1 = l1 + 1
61.            Else
62.                Enqueue2 (c)
63.                l2 = l2 + 1
64.            EndIf
65.            N2 = N2 – 1
67.        EndWhile
68.        While (N2 > 0) do
69.            c = SetCustomer2 ()
70.            If (l1 ≤ l2) then
71.                Enqueue1 (c)
72.                 l1 = l1 + 1
73.            Else
74.                Enqueue2 (c)
75.                l2 = l2 + 1
```

(Contd.)

```
76.             EndIf
77.             N2 = N2 − 1
78.         EndWhile
79.         L1 = L1 + l1, L2 = L2 + l2        // Sum up the lengths of queues for average
                                                 calculation
    /*Servicing the customers in queues*/
    /*AT COUNTER C1*/
80.         If (EMPTY(Q1) = TRUE) then
81.             Go to Step 105
82.         Else
83.             c = SeeCustomer(Q1)           // Look for the customer in front of Q1
84.             If (c − TicketReq = T1) then  // The customer for ticket type T1
85.                 If (c − t > 0) then       // If not finished
86.                     c − t = c − t − 1     // Service for one unit
87.                 Else                      // Service finished
88.                     c − 2 = CLOCK         // Time of completion
89.                     WT1 = WT1 + (c − 2 − c − 1)   //Waiting time for customer
                                                         of T1
90.                     NC1T1 = NC1T1 + 1     // C1 processed one customer of T1
91.                     NT1 = NT1 + 1         // One customer of T1 is served
92.                     Dequeue1()            // Leave the customer for Q1
93.                 EndIf
94.             Else                          // The customer for ticket type T2
95.                 If (c − t > 0) then
96.                     c − t = c − t − 1
97.                 Else
98.                     c − τ2 = CLOCK
99.                     WT2 = WT2 + (c − τ2 − c − 1)
100.                    NT2 = NT2 + 1
101.                    NC1T2 = NC1T2 + 1
102.                    Dequeue1()
103.                EndIf
104.            EndIf
    /*AT COUNTER C2*/
105.            If Empty(Q2) then
106.                Go to Step 135
107.            Else
108.                c = SeeCustomer (Q2)
109.                If (c − TicketReq = T1)
110.                    If (c − t > 0) then
```

(Contd.)

```
                        c - t = c - t - 1
                    Else
                        c - τ2 = CLOCK
                        WT1 = WT1 + (c - τ2 - c - τ1)
                        NC2T1 = NC2T1 + 1
                        NT1 = NT1 + 1
                        Dequeue2()
                    EndIf
                Else
                    If (c - t > 0) then
                        c - t = c - t - 1
                    Else
                        c - τ2 = CLOCK
                        WT2 = WT2 + (c - τ2 - c - τ1)
                        NT2 = NT2 + 1
                        NC2T2 = NC2T2 + 1
                        Dequeue2 ()
                    EndIf
                EndIf
            EndIf
        EndIf
    EndIf
    CLOCK = CLOCK + 1
EndWhile                                        // Elapse one time unit
/*COMPUTE RESULTS*/
lQ1 = L1/OPEN_HOURS                  // Average queue length of the queue Q1
lQ2 = L2/OPEN_HOURS                  // Average queue length of the queue Q2
ΓT1 = WT1/OPEN_HOURS  // Average waiting time of the customers for ticket T1
ΓT2 = WT2/OPEN_HOURS  // Average waiting time of the customers for ticket T2
ΓC1 = NC1T1 - t1 + NC1T2 - t2        // Total service time that the counter C1 served
ΓC2 = NC2T1 - t1 + NC2T2 - t2        // Total service time that the counter C2 served
Stop
```

As stated earlier, the only difference between Model 1 and Model 2 is that in the case of Model 1, a customer either for ticket T_1 or T_2 is allowed in any one of the queues, whereas, in the case of Model 2, queues are earmarked, that is, queue Q_1 is only for the customers of ticket T_1 and queue Q_2 is only for the customers of ticket T_2. From the implementation point of view, Model 2 is simpler than Model 1. The algorithm for Model 2 is as follows:

Algorithm SimulationModel_2

Input: Inputs are same as in the algorithm *SimulationModel_1*
Output: Outputs are same as in the algorithm *SimulationModel_1*
Data structures: Same as in the algorithm *SimulationModel_1*

Steps:

/*INITIALIZATION*/

1. Initialization steps 1.a...1.g remain same as in the algorithm *SimulationModel_1*

/*START SIMULATION FOR A DAY*/

2. **While** (CLOCK < OPEN_HOURS) **do**

/*Generate population of customer at the instant CLOCK*/

3. $N_1 = N(1 - e^{-\lambda_1 * \text{CLOCK}})$

4. $N_2 = N(1 - e^{-\lambda_2 * \text{CLOCK}})$

/*Add customers into the queues*/

5. l_1 = **Length**(Q1)
6. l_2 = **Length** (Q2)
7. **While** ($N_1 > 0$) **do** // Add all customers for ticket T_1 to queue Q_1
8. c = **SetCustomer1**( )
9. **Enqueue1**(c)
10. $l_1 = l_1 + 1$
11. $N_1 = N_1 - 1$
12. **EndWhile**
13. **While** ($N_2 > 0$) **do** // Add all customers for ticket T_2 to queue Q_2
14. c = **SetCustomer2**( )
16. **Enqueue2** (c)
17. $l_2 = l_2 + 1$
18. $N_2 = N_2 - 1$
19. **EndWhile**
20. $L_1 = L_1 + l_1$ // Length of the queues expands after enqueues
21. $L_2 = L_2 + l_2$

/*SERVICING THE CUSTOMERS*/

/*Service at counter C_1*/

22. **If** (**Empty**(Q_1) = TRUE) **then**
23. **Go to** Step 34
24. **Else**
25. c = **SeeCustomer**(Q_1)
26. **If** ($c - t > 0$) **then**
27. $c - t = c - t - 1$
28. **Else**
29. $c - \tau_2$ = CLOCK
30. $WT_1 = WT_1 + (c - \tau_2 - c - \tau_1)$
31. $NT_1 = NT_1 + 1$
32. **EndIf**
33. **EndIf**

/*Service at counter C_2*/

(*Contd.*)

```
34.     If (Empty(Q2) = TRUE) then
35.         Go to Step 46
36.     Else
37.         c = SeeCustomer (Q2)
38.         If (c – t > 0) then
39.             c – t = c – t – 1
40.         Else
41.                 c – τ2 = CLOCK
42.             WT2 = WT2 + (c – τ2 – c – t1)
43.             NT2 = NT2 + 1
44.         EndIf
45.     EndIf
46.     CLOCK = CLOCK + 1
47. EndWhile
    /* COMPUTE RESULTS */
48. lQ1 = L1/OPEN_HOURS                  // Average queue length of the queue Q1
49. lQ2 = L2/OPEN_HOURS                  // Average queue length of the queue Q2
50. GT1 = WT1/OPEN_HOURS          // Average waiting time of the customers for ticket T1
51. ΓT2 = WT2 / OPEN_HOURS        // Average waiting time of the customers for ticket T2
52. ΓC1 = NC1T1 – t1 + NC1T2 – t2        // Total service time that the counter C1 served
53. ΓC2 = NC2T1 – t1 + NC2T2 – t2        // Total service time that the counter C2 served
54. Stop
```

Assignment 5.6 (Simulation of a traffic control system)

Let us consider the case of an automation of a traffic control system. Suppose at a junction point three roads meet (Figure 5.19). For the traffic on the three roads, three signals X, Y and Z are available. Only one signal can be turned on at a time to allow the traffic to pass; for example, if the signal X is on, the traffic from the queue XQ will be allowed to pass either towards Y or towards Z road whereas the traffic in queues YQ and ZQ will wait for their signals. Three models are suggested in order to study the performance of the traffic control:

Model 1: Each signal when turned on allows to pass equal number of vehicles, say, N in succession.

Model 2: Different signals allow different numbers of vehicles during their turn, say, signals X, Y and Z allow N_1, N_2 and N_3 number of vehicles, respectively.

Model 3: Only that signal will be turned on where the maximum number of vehicles are waiting till the number of vehicles waiting are at par with the other remaining queues. If all the queues contain the same number of vehicles, signal X will be turned on.

(*Contd.*)

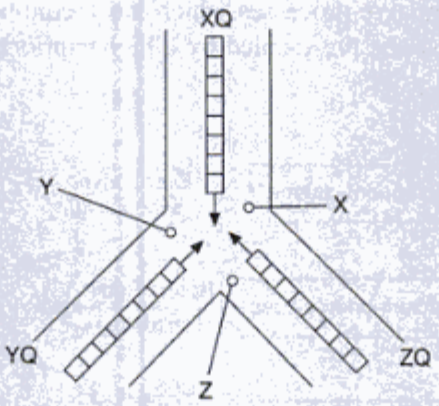


Figure 5.19 A junction of three roads.

Assume that if a queue is empty the corresponding signal is not necessary to be turned on. Write simulation programs to obtain a comparative study using the following arrival rate of vehicles.

1. According to Poisson's distribution
2. According to continuous distribution.

5.5.2 CPU Scheduling in a Multiprogramming Environment

In a multiprogramming environment, a single CPU has to serve more than one program simultaneously. This section gives a brief idea about how queues are important to manage various programs in such an environment.

Let us consider a multiprogramming environment where the possible jobs for the CPU are categorized into three groups:

1. Interrupts to be serviced. A variety of devices and terminals are connected to the CPU and they may interrupt the CPU at any moment to get a particular service from it.
2. Interactive users to be serviced. These are mainly user's programs under execution at various terminals.
3. Batch jobs to be serviced.

These are long-term jobs mainly from non-interactive users, where all the inputs are fed when jobs are submitted; simulation programs, and jobs to print documents are of this kind.

Here the problem is to schedule all sorts of jobs so that the required level of performance of the environment will be attained. One way to implement complex scheduling is to classify the workload according to its characteristics and to maintain separate process queues. So far as the environment is concerned, we can maintain three queues, as depicted in Figure 5.20. This approach is often called *multi-level queues scheduling*. Processes will be assigned to their respective queues. The CPU will then service the processes as per the priority of the queues. In the case of a simple strategy, absolute priority, the process from the highest priority queue (for example, system processes) are serviced until the queue becomes empty. Then the CPU

switches to the queue of interactive processes which has medium priority, and so on. A lower-priority process may, of course, be pre-empted by a higher-priority arrival in one of the upper-level queues.

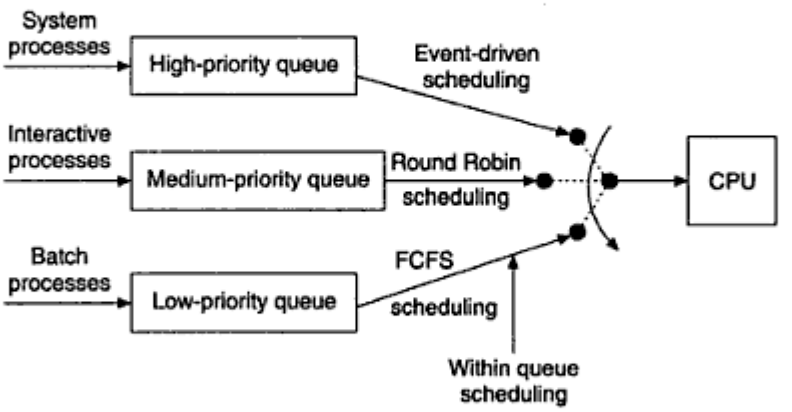


Figure 5.20 Process scheduling with multi-level queues.

Multi-level queues strategy is a general discipline but has some drawbacks. The main drawback is that when processes arriving in higher-priority queues are very high, the processes in a lower-priority queue may starve for a long time. One way out to solve this problem is to time slice between the queues. Each queue gets a certain portion of the CPU time. Another possibility is known as *multi-level feedback queue strategy*. Normally in multi-level queue strategy, as we have seen, processes are permanently assigned to a queue upon entry to the system and processes do not move between queues. The multi-level feedback queue strategy, on the contrary, allows a process to move between queues. The idea is to separate out the processes with different CPU burst characteristics. If a process uses too much of CPU time (that is, long run process), it will be moved to a lower-priority queue. Similarly, a process which is waiting for too long a time in a lower-priority queue, may be moved to a higher-priority queue. For example, consider a multi-level feedback queue strategy with three queues Q_1, Q_2 and Q_3 (Figure 5.21).

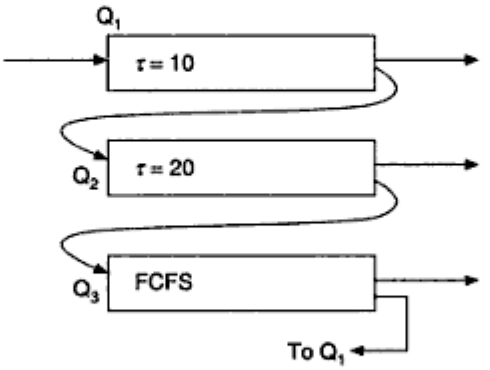


Figure 5.21 A multi-level feedback queue.

A process entering the system is put in queue Q_1. A process in Q_1 is given a time quantum τ of 10 ms, say. If it does not finish within this time, it is moved to the tail of queue Q_2. If Q_1 is empty, the process at the front of queue Q_2 is given a time quantum τ of 20 ms, say. If it does not complete within this time quantum, it is pre-empted and put into queue Q_3. Processes in queue Q_3 are serviced only when queues Q_1 and Q_2 are empty.

Thus, with this strategy, the CPU first executes all processes in queue Q_1. Only when Q_1 is empty it will execute all processes in queue Q_2. Similarly, processes in queue Q_3 will only be executed if only queues Q_1 and Q_2 are empty. A process which arrives in queue Q_1 will pre-empt a process in queue Q_2 or Q_3.

It can be observed that this strategy gives the highest priority to any process with a CPU burst of 10 ms or less. Processes which need more than 10 ms, but less than or equal to 20 ms are also served quickly, that is, they get the next highest priority over the shorter processes. Longer processes automatically sink to queue Q_3; from Q_3, processes will be served on a first-come first-serve (FCFS) basis and in the case of a process waiting for too long a time (as decided by the scheduler) it may be put into the tail of queue Q_1.

5.5.3 Round Robin Algorithm

The *round robin* (RR) algorithm is a well-known scheduling algorithm and is designed especially for time sharing systems. Here, we will see how a circular queue can be used to implement such an algorithm. Before going to implement the RR algorithm, we should first describe the algorithm with illustration. Suppose, there are n processes P_1, P_2, ..., P_n required to be served by the CPU. Different processes require different execution times. Suppose, the sequence of processes' arrivals according to their subscripts, that is, P_1 comes before P_2 and, in general, P_i comes after P_{i-1} for $1 < i \leq n$.

The RR algorithm first decides a small unit of time, called a *time quantum* or *time slice*, τ. A time quantum is generally from 10 to 100 milliseconds. The CPU starts service with P_1. P_1 gets the CPU for time τ, afterwards the CPU switches to P_2, and so on. When the CPU reaches the end of time quantum of P_n it returns to P_1 and the same process will be repeated. Now, during time sharing, if a process finishes its execution before the finishing of its time quantum, the process then simply releases the CPU and the next process in waiting will get the CPU immediately.

As an illustration, consider Table 5.2 for the set of processes:

Table 5.2 Table for process and burst time

Process	*Burst time*
P_1	7
P_2	18
P_3	5

The total CPU time required is 30 unit. Let us assume a time quantum of 4 unit. The RR scheduling for this will be as shown in Figure 5.22.

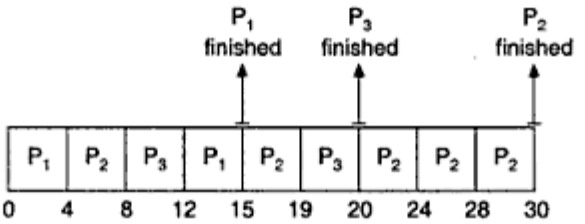


Figure 5.22 RR scheduling.

The advantage of this kind of scheduling is reduction in the average turn around time (not necessarily always true). The turn around time of a process is the time of its completion minus the time of its arrival. Thus, using the FCFS strategy,

$$\text{Average turn around time} = \frac{7 + (7+18) + (7+18+5)}{3} = \frac{62}{3} = 20.66 \text{ unit}$$

Whereas, using the RR algorithm,

$$\text{Average turn around time} = \frac{15 + 30 + 20}{3} = \frac{65}{3} = 21.66 \text{ unit}$$

See the result by repeating the calculations but using the sequence of processes as P_2, P_1 and P_3.

In time sharing systems any process may arrive at any instant of time. Generally, all the processes currently under execution are maintained in a queue. When a process finishes its execution it is deleted from the queue and whenever a new process arrives it is inserted at the tail of the queue and waits for its turn. To illustrate this, let us consider Table 5.3.

Table 5.3 Table for process events

Process	*Arrival time*	*Burst time*
P_1	0	9
P_2	1	3
P_3	9	5
P_4	14	8

The total CPU time required is 25 units. Let the time quantum be $\tau = 5$ unit. Figure 5.23 illustrates the snapshot at various instants with RR scheduling.

Now let us discuss the implementation of the RR scheduling algorithm. A circular queue is the best choice for it. It may be noted that it is not strictly a circular queue, because here a process upon completion is deleted from the queue and it is not necessarily from the front of the queue rather it can be from any position of the queue. Except this, RR scheduling follows all the properties of a queue, that is, the process which comes first gets its turn first.

The implementation of the RR algorithm using a circular queue is straightforward. Here, we use a *variable sized circular queue*; the size of the queue at any instant is decided by the number of processes in execution at that instant. Another mechanism is necessary; whenever a process is deleted, to fill the space of the deleted process, it is required to squeeze all the processes preceding to it, starting from the front pointer (Figure 5.24). (A detailed procedure for implementation is left as an exercise to the reader.)

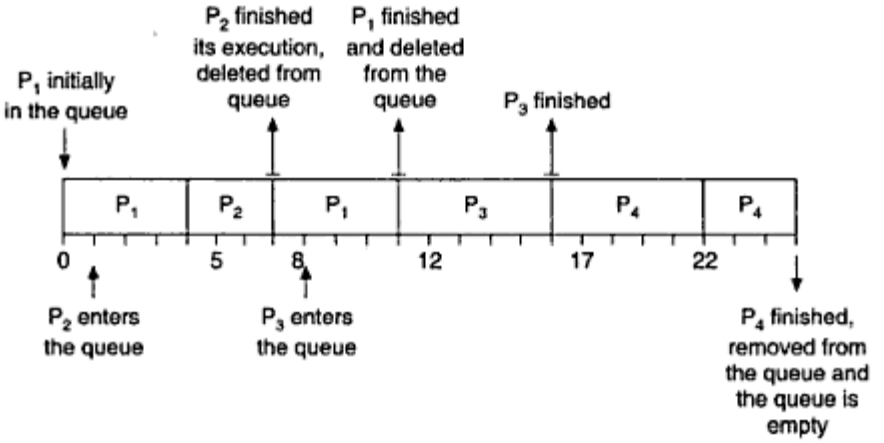


Figure 5.23 In and out in a queue during RR scheduling.

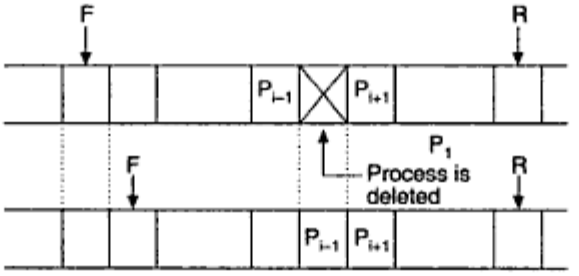


Figure 5.24 Deletion of a process from a circular queue.

5.6 PROBLEMS TO PONDER

5.1 A queue is maintained in an array, and F and R are the front location and rear location of the queue, respectively.

(a) Obtain a formula for N, the number of elements in the queue in terms of F and R.

(b) Write an algorithm to delete the ith element in the queue.

(c) Write an algorithm to insert an item X just after the ith element.

5.2 Repeat Problem 5.1 for a circular queue implemented in an array.

5.3 Write an algorithm REVERSEQ that will reverse all the elements in a queue. [*Hint:* Use a stack.]

5.4 Write an algorithm REVERSECQ that will reverse all the elements in a circular queue which is maintained (i) in an array, (ii) in a singly linked list.

5.5 It is required to split a queue into two queues so that all the elements in odd positions are in one queue and those in even positions are in another queue. Write an algorithm SPLITQ( ) to accomplish this. Assume that the queue is maintained in an array.

5.6 Repeat Problem 5.5 to obtain the algorithm SPLITCQ( ) for a circular queue.

5.7 A deque is a generalization of both stack and queue. Show how a deque can be implemented using two stacks. (That means two stacks are with you, using stack operations only you have to implement the operations of deque.)

5.8 A generalization of both queue and deque is *deck* which is stated as "Addition of elements can be made at the both ends but the deletion can be made either at the end or at the beginning."

Write the operation for deck using the operations of queue and deque only.

5.9 A priority queue can be implemented using a matrix PQ and two arrays FRONT and REAR pointing the front and rear of the queue (Figure 5.25).

Obtain the various operations of a priority queue with such an implementation.

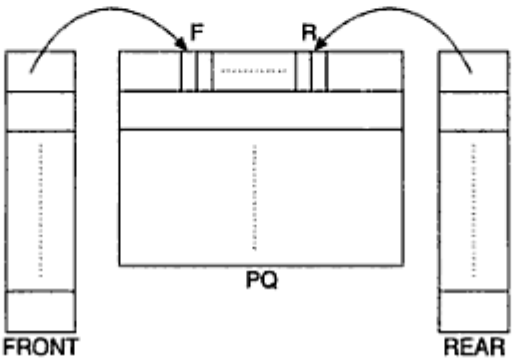


Figure 5.25 A priority queue implementation using a matrix and two arrays.

5.10 Repeat Problem 5.9 with N number of double linked lists instead of matrix PQ and two arrays of pointers FRONT and REAR pointing the front and rear of the lists.

REFERENCES

Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland, New York, 1985.

Jean Paul Tremblay and Paul G. Sorenson, *Introduction to Data Structures with Applications*, McGraw-Hill, New York, 1987.

Robert L. Kruse, Bruce P. Leung and L. Clovis Tondo, *Data Structures and Program Design in C*, Prentice-Hall of India, New Delhi.

Thomas L. Naps, *Introduction to Data Structures with C*, West Publishing Company, West Virginia, 1986.

6

Tables

In this chapter, we will discuss another important data structure called *table*. This kind of data structure plays a significant role in information retrieval. As an example, suppose that a set of n distinct records with keys K_1, K_2, ..., K_n are stored in a file. We want to find a record with a given key value, K. One simplest way is to perform a sequential search, that is, start from the location of the first record, compare the key K with the key of this record, if found then stop, else proceed to the next record and continue the same procedure. The searching time required is directly proportional to the number of records in the file. If the number of records increases our searching time also increases. However, this searching time can significantly be reduced, even can be made independent of the number of records, if we use a table called *access table*. This table may store the location of all records in the file. Figure 6.1 shows the use of an access table to retrieve any record in the file storage. Here, we assume a function f, and if this function is applied on K it returns i, an index, so that $i = f(K)$. Then the ith entry in the access table gives us the location of the record with key value K. This method of accessing any record is called *table lookup* which is, in fact, independent of the number of the records in the file.

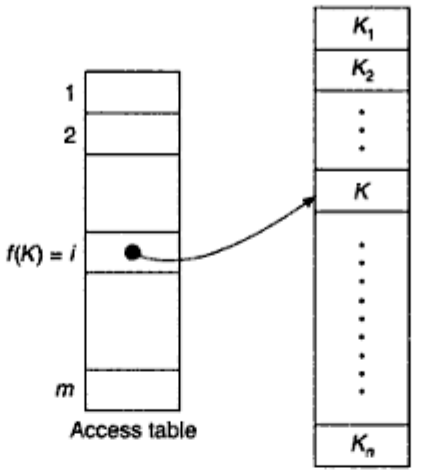


Figure 6.1 Information retrieval through table lookup.

In this chapter, we will explore the various possible kinds of tables and their usefulness. The different tables of interest are listed below:

1. Rectangular tables
2. Jagged tables
3. Inverted tables
4. Hash tables.

The following sections discuss all these tables.

6.1 RECTANGULAR TABLES

Rectangular tables are also known as *matrices*. Matrices have already been discussed in Chapter 2, Section 2.4. Since these tables are needed in various applications, almost all programming languages provide convenient and efficient means to store and access them, so that the programmer does not have to worry about the implementation details. Rectangular tables, therefore, do not require any further discussion.

6.2 JAGGED TABLES

Jagged tables are nothing but a special kind of sparse matrices such as triangular matrices, band matrices, etc., which have already been discussed in Section 2.4.2, Chapter 2. In the jagged table, we put a restriction that if elements are present in a row (or in a column) then they are contiguous. Thus in Figures 6.2(a)–(e), all are jagged tables except the table in Figure 6.2(f).

We have seen (in Section 2.4.2) how sparse matrices can be stored in a one-dimensional array and if the sparse matrices are symmetric in form then how their indexing formula can be decided so that any element in the matrices can be accessed. Recall the indexing formula for any element a_{ij} in a matrix is

$$\text{Address}(a_{ij}) = \frac{i \times (i-1)}{2} + j \tag{6.1}$$

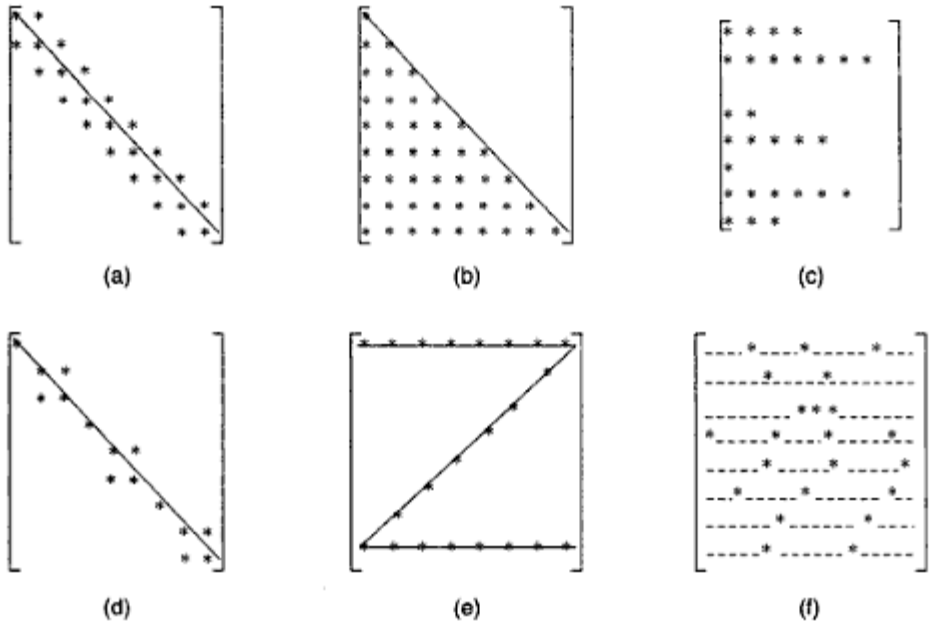


Figure 6.2 Jagged tables and sparse matrices.

This formula involves multiplication and division which are in fact inefficient from the computational point of view. Here, we will discuss another alternative but improved technique, where we can avoid multiplication and division by setting up an access table whose entries correspond to the row indices of the jagged table, such that the ith entry in the access table is

$$\frac{i \times (i-1)}{2}$$

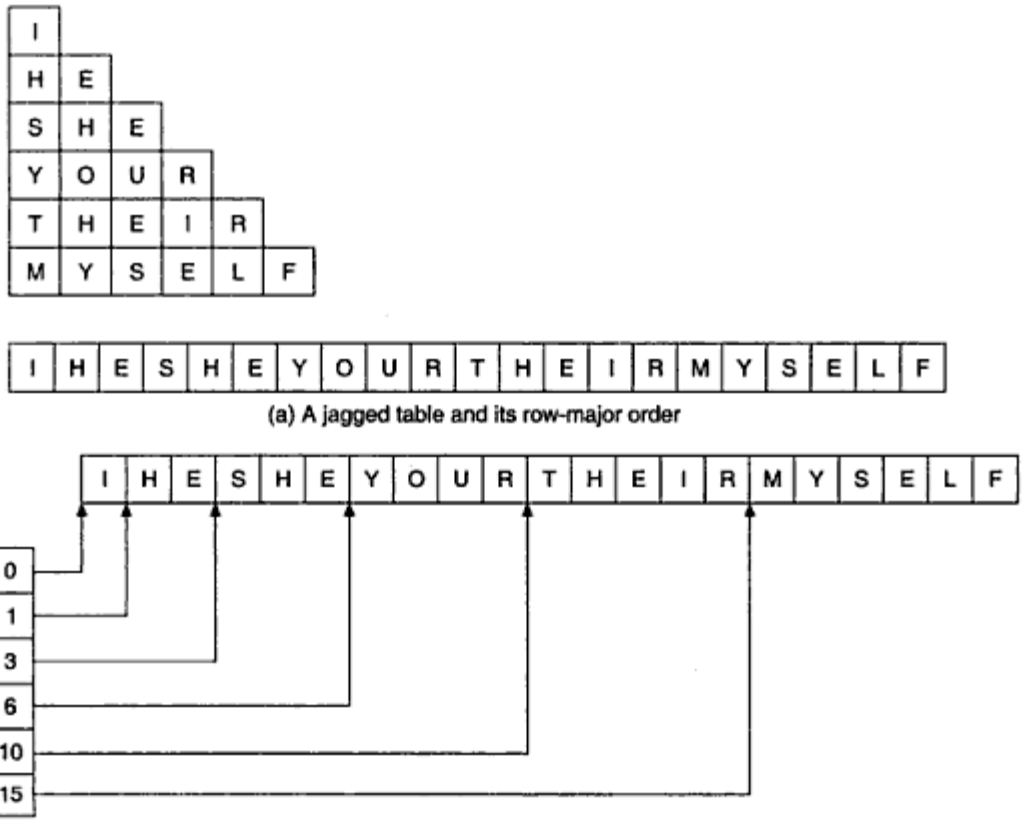
The access table is calculated only once at the time of initiation and can be stored in memory, it then can be referred each time the access of an element in the jagged table occurs. It may be noted that even during the initial calculation, for the entries in the access table, it does not require any multiplication or division but only addition such as

$$0, 1, (1 + 2), (1 + 2) + 3, \ldots$$

In Figure 6.3, the representation of a jagged table is illustrated. For example, if we want to access a_{54} (the element in the 5th row and the 4th column) then at the 5th location of the access table, we see that the entry is 10; hence the desired element is at 14 (= 10 + 4) location of the array which physically contains the element. It is assumed that the first element of the table is located at location 1 of the array.

The above-mentioned accessing technique has another advantage over the indexing formula. We can find an indexing formula even if a jagged table is asymmetric with respect to the arrangement of elements in it.

For example, in the jagged table shown in Figure 6.2(d), it is difficult to find an index formula. In this case, however, using an access table, we can easily maintain its storage in an array and can obtain faster access of elements from it.



(a) A jagged table and its row-major order

(b) Accessing through access table

Figure 6.3 Access technique of a jagged table.

Figure 6.4 illustrates another example of such accessing. Here, the setting up of the access table and its use is the same as in the case of a jagged table corresponding to symmetric sparse matrixes. An entry in the ith location of the access table can be obtained by adding the number of elements in the $(i-1)$th row of the jagged table and the $(i-1)$th entry of the access table, assuming that entry 0 is the first entry in the access table, and as before, the starting location of the array storing the elements is 1.

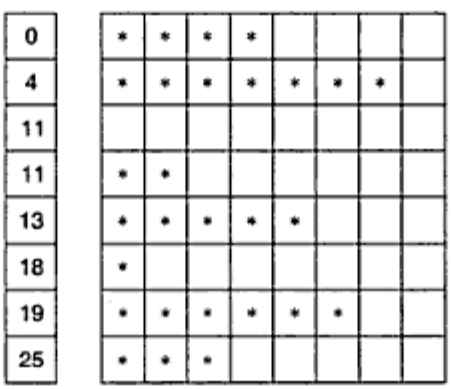


Figure 6.4 Jagged tables.

Assignment 6.1

Consider two jagged tables as shown in Figure 6.5. In one table, the row and column indices vary from 1 to n, and in the other table, the row index varies from $-n$ to n.

(a) Devise an index formula for the above two sparse matrices to store in arrays.

(b) Obtain the access tables and set their entries.

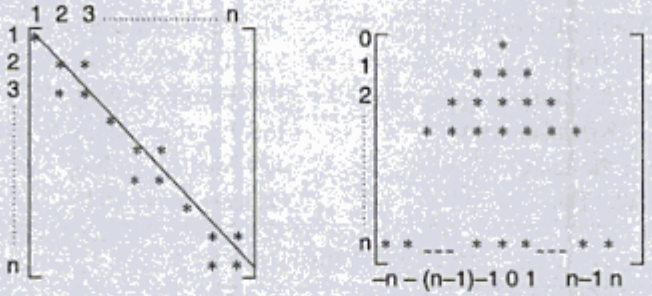


Figure 6.5 Two jagged tables for Assignment 6.1.

6.3 INVERTED TABLES

It will be judicious if we discuss the concept of inverted tables with the help of the following example.

Suppose a telephone company maintains records of all the subscribers of a telephone exchange as shown in Table 6.1. These records can be used to serve several purposes. One of them is the alphabetical ordering of the names of subscribers (say, in order to put the name of the subscriber into a telephone directory). Second, it may be the lexicographical ordering of the addresses of subscribers (say, for routine maintenance). Third, it could be the ascending order of the telephone numbers (say, in order to estimate the cabling charge from the telephone exchange to the location of telephone connection), etc. To serve all these purposes, the telephone company should maintain three sets of records: one in alphabetical order of the NAME, second, the lexicographical ordering of the ADDRESS and third, the ascending order of the phone numbers. But this way of maintaining records leads to the following serious drawbacks:

1. Requirement of extra storage: three times the actual memory.
2. Difficulty in modification of records: if a subscriber changes his address, then we have to modify this in three storages, otherwise consistency in information will be lost.

However, using the concept of inverted tables, we can avoid multiple sets of records, and we can still retrieve the records by any of the three keys almost as quickly as if the records were fully sorted by that key. Therefore, we should maintain an inverted table. In this case, this table consists of three columns: NAME, ADDRESS, and PHONE as shown in Table 6.1(b). Each column contains the index numbers of records in the order based on the sorting of the corresponding key. This inverted table, therefore, can be consulted to retrieve information.

Table 6.1 Multi-key access and its inverted table

(a) Records of a Telephone Exchange

Index	*Name*	*Address*	*Phone*
1	K.R. Narayana	Maker Towers #6	257696
2	A.B. Vajpayee	9 Vivekananda Road	257459
3	L.K. Advani	11 Von Kasturba Marg	257583
4	Mamta Banerjee	342 Patel Avenue	257423
5	Y. Sinha	5 SBI Road	257504
6	D. Kulkarni	369 Faculty Colony	257564
7	T. Krishnamurthy	185 Faculty Colony	257579
8	N. Puranjay	409 Medical Colony	257409
9	Tadi Tabi	Officers Mess #52	257871

(b) Inverted Table

Name	*Address*	*Phone*
2	7	8
6	6	4
1	1	2
3	8	5
4	9	6
8	4	7
7	5	3
9	2	1
5	3	9

6.4 HASH TABLES

There are other types of tables which help us to retrieve information very efficiently. The ideal *hash table* is merely an array of some constant size; the size depends on the application where it will be used. The hash table contains key values with pointers to the corresponding records. The basic idea of a hash table is that we have to place a key value into a location in the hash table; the location will be calculated from the key value itself. This one-to-one correspondence between a key value and an index in the hash table is known as *address calculation indexing* or more commonly *hashing*. In the present section, we will discuss hashing techniques and their related issues.

6.4.1 Hashing Techniques

The main idea behind any hashing technique is to find a one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in Figure 6.6, where K denotes a set of key values, I denotes a range of indices and H denotes the mapping function from K to I.

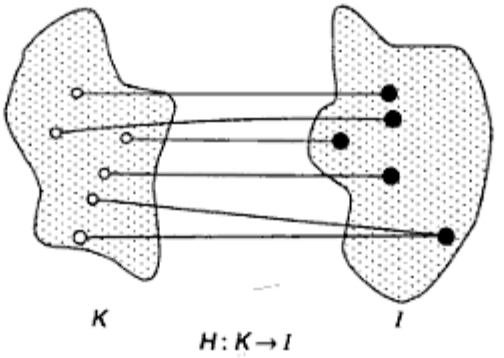


Figure 6.6 Concept of hashing.

It may be noted that the mapping is subjective, that is all key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the *hash function*. A particular hashing technique uses a particular hash function. The hash function plays a dominant role in hashing techniques. There are two principal criteria in deciding a hash function $H:K \rightarrow I$ as follows:

1. The function H should be very easy and quick to compute.
2. The function H should as far as possible give two different indices for two different key values.

As an example, let us consider a hash table of size 10 whose indices are 0, 1, 2, ..., 8, 9. Suppose a set of key values are: 10, 19, 35, 43, 62, 59, 31, 49, 77, 33. Let us assume the hash function H is as stated below:

- Add the two digits in the key.
- Take the digit at the unit place of the result as the index; ignore the digit at the tenth place, if any.

Using this hash function, the mappings from key values to indices and to hash table are shown in Figure 6.7. In this example, for the given set of key values, the hash function does

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

$H: K \rightarrow I$

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

Figure 6.7 Example of hashing.

not distribute them uniformly over the hash table; some entries are there which are empty, and in some entries more than one key value needs to be stored. Allotment of more than one key value in one location in the hash table is called *collision*. We have found three collisions for 62, 31 and 77 in the above-mentioned example.

It can be noted that $| K | = | I |$, that is, the number of key values is the same as the size of the hash table, but this is not the case always. In general, $| K | > | I |$.

The following are some hash functions which are very common and popularly applied in various applications.

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number N of keys in K. The hash function H is then defined by

$$H(k) = k(\text{MOD } h) \qquad \text{if indices start from 0}$$

or

$$H(k) = k(\text{MOD } h) + 1 \qquad \text{if indices start from 1}$$

where $k \in K$, a key value. The operator MOD defines the modulo arithmetic operation, which is equal to the remainder of dividing k by h. For example, if $k = 31$ and $h = 13$ then

$$H(31) = 31(\text{MOD } 13) = 5$$

or

$$H(31) = 31(\text{MOD } 13) + 1 = 6$$

The number h is usually chosen to be a prime number or a number without small divisors, since this usually minimizes the number of collisions. Generally, h is a prime number and equal to the size of the hash table.

Midsquare method

Another hash function which has been widely used in many applications is the midsquare method. The method is defined as follows:

The hash function H is defined by $H(k) = x$, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. This selection usually depends on the size of the hash table. It needs to be emphasized that the same criteria should be used for selecting the bits or digits for all of the keys.

As an example, suppose the key values are of the integer type, and we require 3-digit addresses. Our selection criteria are to select 3 digits at even positions starting from the right-most digit in the square. Let us see the address calculations, for 3 distinct keys and with the hash function, as defined above:

k	:	1234	2345	3456
k^2	:	1522756	5499025	11943936
$H(k)$	:	525	492	933

Here, we observe that the second, the fourth, and the sixth digits, counting from the right, are chosen for the hash addresses.

The midsquare method has been criticized because of time-consuming computation (multiplication operation), but it usually gives good results so far as the uniform distribution of the keys over the hash table is concerned.

Folding method

Another fair method for a hash function is the folding method. The method can be defined as follows:

Partition the key k into a number of parts k_1, k_2, ..., k_n, where each part, except possibly the last, has the same number of bits or digits as the required address width. Then the parts are added together, ignoring the last carry, if any. Alternatively,

$$H(k) = k_1 + k_2 + \cdots + k_n$$

where the last carry, if any, is ignored. If the keys are in binary form, the exclusive-OR operation may be substituted for addition. There are many variations known in this method. One is called the *fold shifting method,* where the even number parts, $k2$, $k4$, ... are each reversed before the addition. Another variation is called the *fold boundary method.* Here, two boundary parts, namely, k_1 and k_n, each are reversed and then added to all other parts. As an example, let us take the size of each part to be 2; the following calculations are performed on the given key values (integers) as shown below:

k:	1522756	5499025	11943936
Chopping:	01 52 27 56	05 49 90 25	11 94 39 36
Pure folding:	01 + 52 + 27 + 56 = 136	05 + 49 + 90 + 25 = 169	11 + 94 + 39 + 36 = 180
Fold shifting:	10 + 52 + 72 + 56 = 190	50 + 49 + 09 + 25 = 133	11 + 94 + 93 + 36 = 234
Fold boundary:	10 + 52 + 27 + 65 = 154	50 + 49 + 90 + 52 = 241	11 + 94 + 39 + 63 = 207

Folding is a hashing function which is also useful in converting multi-word keys into a single word so that another hashing function can be used on that. In fact, the term 'hashing' comes from this technique of 'chopping' a key into pieces.

Digit analysis method

The basic idea of this hashing function is to form hash addresses by extracting and/or shifting the extracted digits or bits of the original key. As an example, given a key value, say 6732541, it can be transformed to the hash address 427 by extracting the digits in even positions and then reversing this combination. For a given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and then rearrangement is based on some analysis. To do this, an analysis is performed to determine which key positions should be used in forming hash addresses. For each criterion, hash addresses are calculated and then a graph is plotted, then that criterion is selected which produces the most uniform distribution, that is with the smallest peaks and valleys.

This method is particularly useful in the case of static files where the key values of all the records are known in advance.

We have assumed the key values as integers in our previous discussions, but it need not be so always. In fact, any key value can be represented by a string of characters and then ASCII values of its constituent characters can be taken to convert it into a numeric value. Thus, assuming that a key value $k = k_1k_2k_3 \ldots k_n$, where each k_i is the constituent character in k. The hash function using the division method is stated as below in algorithm *HashDivision*.

Algorithm HashDivision

Input: K, the key value in the form of a string of characters whose hash address is to be calculated.
Output: *INDEX*, a positive integer as the hash address.
Data structure: Hash table in the form of an array. H is the size of the hash table which is used for modulo arithmetic operation.

Steps:

```
1. i = 1                                   // i is the pointer to the string K
2. keyVal = 0                              // To store the keyvalue of K
3. While (K[i] ≠ NULL) do
4.     keyVal = keyVal + K[i]              // Add the ASCII value of K
5.     i = i + 1                           // Move to the next character
6. EndWhile
7. INDEX = keyVal MOD H + 1                // Find the remainder modulo
8. Return (INDEX)
9. Stop
```

The algorithms for other hash functions can be designed likewise; these have been left as exercises for the student.

Assignment 6.2

1. Write the algorithms *HashMidSquare*, *HashFoldingPure*, *HashFoldingShift*, *HashFoldingBoundary*, for the *midSquare* method and different variations of the folding method, respectively, for a given key value K in the form of a string of characters.
2. Suppose a file contains 100 records. What should be the size of the hash table and hence h? Create an array of key values of the records as hash table. Generate 100 random numbers and assume them as key values. Apply different hash functions to calculate hash addresses and load the key values into the hash table.
3. Generate 100 random numbers each of 6 digits, say. Assuming these as static key values, obtain a digital analysis on it based on three different criteria. Plot a graph for the hash values obtained with these criteria and then select the best criteria.

4. A hash function will be termed good if it satisfies both the criteria, namely, quick to compute and uniformity of distribution. The first criteria can be controlled by using a suitable technique of computation. For example, suppose, for a key $k = k_1k_2k_3$, the hash function is $H(k) = k_1 + k_2 * 27 + k_3 * 27^2$. Using Horner's rule for evaluating a polynomial, this can be calculated as:

$$H(k) = (k_3 * 27 + k_2) * 27 + k_1$$

Extend the Horner's rule to compute the hash address for the following:

(a) $H(k_1k_2k_3 \cdots k_r) = \sum_{i=1}^{r} k_i 27^{i-1}$

Hint: Additions can be replaced by the bit-wise exclusive-OR operation for increased speed.

(b) $H(k_1k_2k_3 \cdots k_r) = \sum_{i=1}^{r} k_i 32^{i-1}$

Hint: Additions can be replaced by the bit-wise exclusive-OR for increased speed and multiplication by 32 is not really a multiplication, it can be done by shifting the bit by 5.

(c) A hybrid hash function $H^*(k)$ of $H(k)$ can be obtained by applying the division method over $H(k)$ as stated below:

$$H^*(k) = H(k) \text{ MOD } h$$

h being the size of the hash table.

Obtain the algorithm for such a hybrid hash function.

6.4.2 Collision Resolution Techniques

Whatever the hash function used in hashing, the complete removal of collisions is almost impossible. This can be emphasized with an example called *birth day surprise*. Suppose there is a class of 24 students and they are having the same year of birth. We want to know the probability that two students have the same date of birth. The probability can be calculated as follows:

Open the calendar of the year of their birth. Assume that there are 365 days. Start with any student, and put a tick on his birthday date on the calendar. Now, the probability that the second student has a different birthday is 364/365. Tick this date off. The probability that a third student has a different birthday is now 363/365. Continuing this way, we see that if the first $(n - 1)$ students have different birthdays, then the probability that the nth student has a different birthday is

$$\frac{365 - (n-1)}{365} \quad \text{or} \quad \frac{365 - n + 1}{365}$$

Since the birthdays of different people are independent, we obtain the probability that n students all have a different birthday is

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365 - n \times 1}{365}$$

This probability can be calculated as less than 0.5 whenever $n \geq 24$.

In other words, suppose there is a hash table of size 365 and we want to store the records of all the 24 students based on birthdays as their key values. It is therefore a fifty-fifty chance that two of the students have the same birthday and hence a collision.

So, collision in hashing cannot be ignored, whatever be the size of the hash table. The next question arises therefore is what to do if their is a collision? There are several techniques to resolve the collisions. Two important methods are listed below:

(a) Closed hashing (also called *linear probing*)
(b) Open hashing (also called *chaining*).

6.4.3 Closed Hashing

The simplest method to resolve a collision is *closed hashing*. Suppose there is a hash table of size h and the key value of interest is mapped to an address location i, with a hash function. The closed hashing then can be stated as follows:

Start with the hash address where the collision has occurred, let it be i. Then follow the following sequence of locations in the hash table and do the sequential search.

$$i, i + 1, i + 2, ..., h, 1, 2, ..., i - 1$$

The search will continue until any one of the following cases occurs:

- The key value is found.
- An unoccupied (or empty) location is encountered.
- The searches reaches the location where the search had started.

The first case corresponds to the successful search and the last two cases correspond to unsuccessful search. Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is also alternatively termed *linear probing*; probe means key comparison.

Let us illustrate the method with an example. Assume that there is a hash table of size 10 and the hash function uses the division method with remainder modulo 7, namely, $H(k) = k$ MOD $(7 + 1)$. Let us consider the build up of the hash table (initially, the table is empty) with the following set of key values:

15 11 25 16 9 8 12 8

The loading of the hash table will take place successively by performing a search for a key and inserting it into the table in an empty room if the key is not in the table and leaving if it is overflow, that is, no free room to accommodate any further key value. This is illustrated in Figure 6.8.

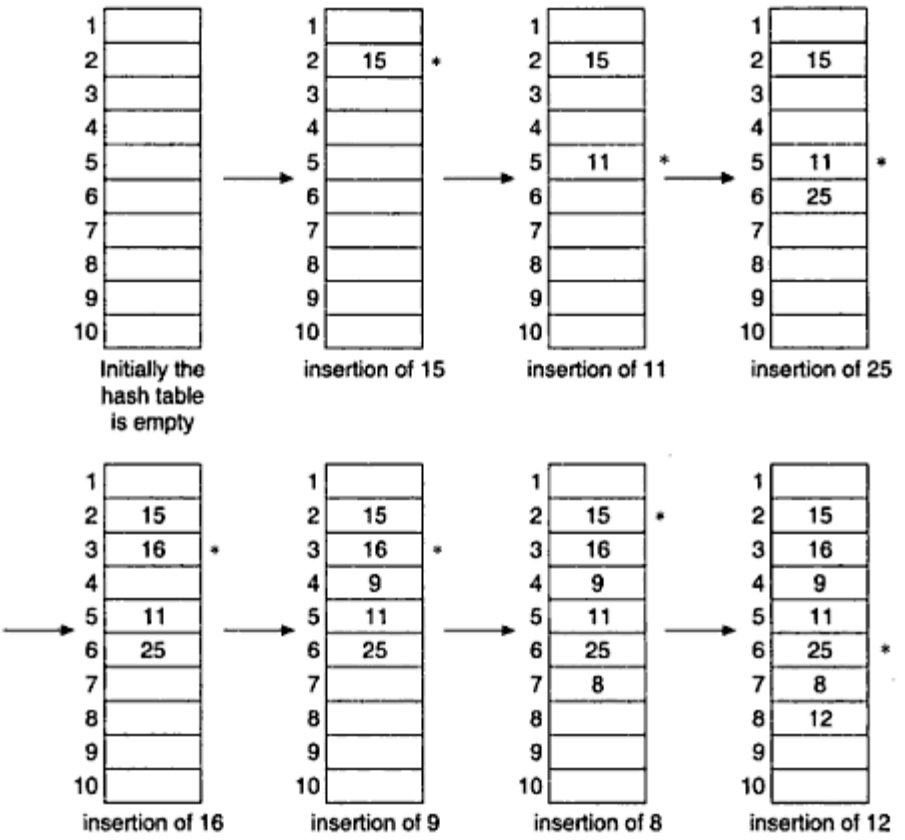


Figure 6.8 Building up a hash table.

Next, let us define the operation for searching a key-value and inserting a key-value. The algorithm *HashLinearProbe* for searching a key value *K* in a hash table of size *HSIZE* is given below:

Algorithm HashLinearProbe

Input: *K* is the key value of search. *INSERT* is a flag for the insertion operation.

Output: Return the location if it is found in the hash table else if *INSERT* is TRUE put *K* in the table if table has not overflown otherwise return NULL.

Data structures: A hash table *H* of size HSIZE in the form of an array.

Steps:

```
1.  flag = FALSE                                  // Flag for continuation of looping
2.  index = HashFunction(K)                       // Calculate the hash address using a hash function
3.  If (K = H[index]) then                        // If there is a hit
4.      Return(index)
5.      Exit                                      // End of the execution
6.  Else
7.      i = index + 1                             // Set to the next location
```

(*Contd.*)

```
 8.    While (i ≠ index) and (not flag) do
 9.       If ((H[i] = NULL) or (H[i] < 0)) then          // If the cell is free
10.          If (INSERT) then                            // True option for insertion
11.             H[i] = K                                 // Put the key value into the hash table
12.             flag = TRUE
13.          EndIf
14.       Else                                           // Cell is occupied
15.          If (H[i] = K) then                          // Match
16.             flag = TRUE
17.             Return(i)
18.             Exit                                    // End of the execution
19.          Else                                       // No match
20.             i = i MOD h+1                           // Closed looping
21.          EndIf
22.       EndIf
23.    EndWhile
24.    If (flag = FALSE) and (i = index) then    // No match and reach to the starting point
25.       Print "The table is overflow"
26.    EndIf
27. EndIf
28. Stop
```

Note Step 9 in the above algorithm. Here, we assume that whenever a key value is deleted from the hash table its corresponding entries are made negative instead of NULL. Writing an algorithm for deleting a key value is straightforward and is left as an exercise.

Drawback of closed hashing and its remedies

The major drawback of closed hashing is that, as half of the hash table is filled, there is a tendency towards *clustering*; that is key values are clustered in large groups and as a result a sequential search becomes slower and slower. This kind of clustering is typically known as *primary clustering*.

The following are some solutions known to avoid this situation:

(a) Random probing
(b) Double hashing or rehashing
(c) Quadratic probing.

Let us briefly discuss each of the above solutions.

Random probing. This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in the linear probing method. The random sequence generated by the pseudo random number generator contains all the positions between 1 and h, the highest location of the hash table. An example of a pseudo random number generator that produces such a random sequence of locations is given below:

$$i = (i + m) \text{ MOD } h + 1$$

where i is a number in the sequence, and m and h are integers that are relatively prime to each other (that is, their greatest common divisor is 1). For example, suppose $m = 5$ and $h = 11$ and initially $i = 2$, then the above-mentioned pseudo random number generator generates the sequence as:

$$8, 3, 9, 4, 10, 5, 11, 6, 1, 7, 2$$

We stop producing the numbers when the first location is duplicated. Observe that here all the numbers between 1 and 11 are generated but randomly. We can avoid primary clustering if the probe follows the said random sequence.

Double hashing. Random hashing however is not free form clustering. Another type of clustering, called *secondary clustering*, is involved here. In particular, clustering occurs when two keys are hashed into the same location. In such an instance, if the same sequence of locations is generated for two different keys by the random probing method then clustering takes place. An alternative approach to avoid the secondary clustering problem is to use a second hash function in addition to the first one. This second hash function results in the value of m for the pseudo random number generator as employed in the random probing method. This second function should be selected in such a way that the hash addresses generated by the two hash functions are distinct and the second function generates a value m for the key k so that m and h are relatively prime. Let us consider the following example.

Suppose $H_1(k)$ is the initially used hash function and $H_2(k)$ is the second one. These two functions are defined as

$$H_1(k) = (k \text{ MOD } h) + 1$$

$$H_2(k) = (k \text{ MOD } (h - 4)) + 1$$

Let $h = 11$ and $k = 50$ for an instance. Then, $H_1(50) = 7$ and $H_2(50) = 2$. Therefore, $H_1(50) \neq H_2(50)$, that is, H_1 and H_2 are independent and $m = 2$, $h = 11$ are relatively prime. Hence, using $i = [(i + 2) \text{ MOD } 11] + 1$, and initially $i = 7$, we have the random sequence as

$$10, 2, 5, 8, 11, 3, 6, 9, 1, 4, 7$$

Now, let us choose another key value which has the same hash address as that of 50 (that is, 7) with the first hash function H_1. Let it be 28 (since $H_1(28) = 28 \text{ MOD } 11 + 1 = 7$). Then

$$H_2(28) = 28 \text{ MOD } 7 + 1 = 5$$

So using $i = [(i + m) \text{ MOD } 11]$ with $i = 7$ and $m = 5$, we get the sequence:

$$2, 8, 3, 9, 4, 10, 5, 11, 6, 1, 7$$

Thus, for the two key values where the hash address is the same and using rehashing, two different random sequences are generated, thereby alleviating the secondary clustering.

Quadratic probing. Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations $i + 1$, $i + 2$, $i + 3$, etc. are probed; but in quadratic probing, the next locations to be probed are $i + 1^2$, $i + 2^2$, $i + 3^2$, etc. Mathematically, if h is the size of the hash table and $H(k)$ is the hash function then the quadratic probing searches the locations:

$$H(k) + i^2 \text{ MOD } h \quad \text{for } i = 1, 2, 3, \ldots$$

Note that in quadratic probing the increment function is i^2. It also assumes the hash table as close (or circular) as in linear probing.

This method, no doubt, substantially reduces primary clustering, but it does not probe all the locations in the table. Lemma 6.1 gives the information regarding the number of location that it can probe at most.

Lemma 6.1

If h denotes the size of the hash table then the number of distinct positions that will be probed is $(h + 1)/2$.

Proof: Suppose that the hash address for a given key k is x. Then the ith probe will look like

$$x + i^2 \bmod h = x + [1 + 3 + 5 + \cdots + (2i - 1)] \bmod h$$

or,

$$(2i - 1) \leq h$$

i.e.

$$i = \frac{h+1}{2} \tag{6.2}$$

Hence proved.

Example: Suppose h =11 and the hash address of the key is x. Then the different locations with a quadratic probe are x, $x + 1$, $x + 4$, $x + 9$, $x + 5$, $x + 3$ with $(11 + 1)/2 = 6$ probes.

Drawback of quadratic probing: For linear probing, it is not advisable to let the hash table get nearly full because in that case we may have to search the entire table and thus performance degrades. For quadratic probing, the situation is even more drastic: there is no guarantee of finding an empty cell once more than half of the table gets full or even before that if the table size is not prime. Lemma 6.2 supports the above situation.

Lemma 6.2

If quadratic probing is used and the table size is prime, then a new key value can always be inserted if the table is at least half full.

Proof (By the method of contradiction): Let the table size h be an (odd) prime number greater than 3. We show that the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Two of these locations are

$$x + i^2 \text{ MOD } h \quad \text{and} \quad x + j^2 \text{ MOD } h$$

where $0 < i, j \leq \lfloor h/2 \rfloor$, and x is the hash address of a key. Suppose by contradiction, these locations are the same, but $i = j$. Then

$$x + i^2 \text{ MOD } h = x + j^2 \text{ MOD } h$$

or

$$(i^2 - j^2) \text{ MOD } h = 0$$

or

$$(i - j) \times (i + j) \text{ MOD } h = 0$$

Since h is prime, it follows that either $i - j$ or $i + j$ is divisible by h. Again $i \neq j$, $i, j \leq \lfloor h/2 \rfloor$, so $(i - j)$ MOD $h \neq 0$. The second option is also not possible as $i, j < \lfloor h/2 \rfloor$, their sum can never be $m \times h$, for $m = 1, 2, 3, \ldots$.

Thus, the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Since the element to be inserted can also be placed in the location to which it hashes, if there are no collisions, any element has $\lfloor h/2 \rfloor$ locations into which it can be placed. Hence, proved.

In quadratic probing, it is also very crucial that the table size should be a prime. If the table size is not prime, the number of alternate locations can be severely reduced. As an example, if the table size is 16, (or a power of 2), then the only alternate locations would be at distances 1, 4, 9, etc.

6.4.4 Open Hashing

So far we have discussed the closed hashing methods of collision resolution. The closed hashing method deals with arrays as hash tables and thus we are able to refer quickly to random positions in the tables. But there are two main difficulties with this technique: First, it is very difficult to handle the situation of table overflow in a satisfactory manner. Second, the key values are haphazardly intermixed and, on the average, the majority of the keys are far from their hash locations, thus increasing the number of probes which degrades the overall performance.

To resolve these problems another hashing method called *open hashing* (also called *separate chaining*, or simply *chaining*) is known. The chaining method is discussed in the following paragraphs.

The chaining method uses a hash table as an array of pointers; each pointer points a linked list. That is, here the hash table is an array of list headers. In Figure 6.9, a hash table of size 10 is considered. The index of the hash table varies from 0 to 9 and key values are taken as integers. The hash address for a key is decided by its last digit (means the right most digit).

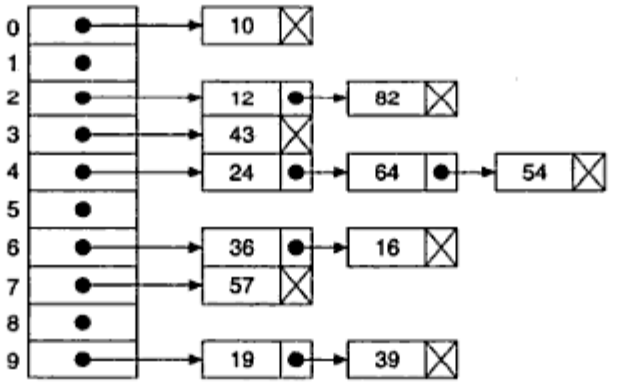


Figure 6.9 An open hashing.

For a given key value, the hash address is calculated. It then searches the linked list pointed by the pointers at that location. If the element is found it returns the pointer to the node containing that key value else inserts the element at the end of that list. The implementation of open hashing is stated in the algorithm *HashChaining* as follows:

Algorithm HashChaining

Input: K is the item of interest. *INSERT* is a flag for the option of insertion.
Output: If K is found in the hash table then return the pointer of the node which contains the key value K else insert K into the linked list when the *INSERT* flag is TRUE.
Data structure: Hash table H having size *HSIZE* storing pointer to the single linked list structure.

Steps:

```
1.  index = HashFunction(K)                    // Calculate the hash address of K
2.  ptr = H[index]                             // ptr is a pointer to any node in the list
3.   flag = FALSE                              // flag for controlling the search
4.  While (ptr ≠ NULL) and (flag = FALSE) do
5.      If (ptr→DATA = K) then                 // End of search
6.          flag = TRUE
7.          Return(ptr)
8.          Exit                               // End of execution
9.      Else
10.         ptr = ptr.LINK                     // Move to the next node
11.     EndIf
12. EndWhile
13. If (flag = FALSE) then
14.     Print "Key value does not exist"
15.     If (INSERT) then
16.         InsertEnd_SL(H[index])             // Insert it into the table
17.     EndIf
18. EndIf
19. Stop
```

A key value if it exist can be deleted from a hash table for which a procedure *HashKeyDelete*(...) can be written. This is left as an exercise for the reader.

Advantages and disadvantages of chaining

There are several advantages of the chaining method. The most important advantages are stated below:

1. An overflow situation never arises. The hash table maintains lists which can contain any number of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.

3. Insertion and deletion become a quick and an easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in a single linked list.
4. Finally, open hashing is best suitable in applications where the number of key values varies drastically as open hashing uses dynamic storage management policy.

The only disadvantage of the chaining method is that of maintaining linked lists and extra storage space for link fields.

Assignment 6.3

As an alternative to the collision resolution technique, *bucket hashing* can be used. In this method, the hash table is a collection of buckets and each bucket contains a few number of key values decided by the bucket size. Let us assume that all buckets are of the same size. Here, the hash function calculates an address of a bucket and then finally the key value is searched in that bucket. Figure 6.10 illustrates this concept for a hash table with buckets whose size is 3.

(a) Compare bucket hashing with open hashing and closed hashing.

(b) Write algorithms to search a key value, insert a key value and delete a key value in bucket hashing.

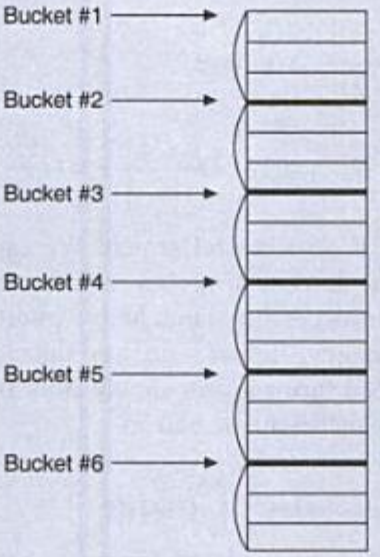


Figure 6.10 Bucket hashing.

6.4.5 Comparison of Collision Resolution Techniques

We will conclude the discussion of hash tables by giving an analytical comparison of various collision resolution techniques discussed. Let us define the *load factor*, λ, of a hash table as

$$\lambda = \frac{\text{Total number of key values}}{\text{Size of the hash table}} \tag{6.3}$$

So $\lambda = 1.0$ means that the number of key values is the same as the total capacity of the hash table. We also define $S(\lambda)$ and $U(\lambda)$ as

$S(\lambda)$ = average number of probes for a successful search.

$U(\lambda)$ = average number of probes for an unsuccessful search.

These two quantities will measure the performance of collision resolution methods.

Analysis of closed hashing

To analyze the performance of closed hashing, let us assume the case of random probing and ignore the problem of clustering for the sake of simplicity.

Let us first consider the case of unsuccessful search. It is evident that the probability that the first probe hits an occupied cell is λ, the load factor. The probability that a probe hit an empty cell is $1 - \lambda$. The probability that the unsuccessful search terminates in exactly two probes is therefore $\lambda(1 - \lambda)$. Arguing similarly this way, the probability that exactly k probes are made in an unsuccessful search is $\lambda^{k-1}(1 - \lambda)$. The average number of probes for an unsuccessful search is therefore

$$U(\lambda) = \sum_{k=1}^{\infty} k\lambda^k(1-\lambda) = (1-\lambda)\sum_{k=1}^{\infty} k\lambda^k \tag{6.4a}$$

Since $\lambda \leq 1$ and $\sum_{k=1}^{\infty} k\lambda^k = \frac{1}{(1-\lambda)^2}$, we have

$$U(\lambda) = (1-\lambda)\frac{1}{(1-\lambda)^2} = \frac{1}{1-\lambda} \tag{6.4b}$$

Next, let us consider the case of a successful search. We can think of this problem through insertion of key values. Then the number of probes required will be exactly one more than the number of probes made in the unsuccessful search before inserting the item. Let us consider the case when the table is initially empty. In that state, key values are inserted one at a time. Now as the items are inserted, the load factor grows slowly from 0 to λ. Thus, we can express the average number of probes in a successful search as

$$\begin{aligned} S(\lambda) &= \frac{1}{\lambda}\int_0^{\lambda} U(x)\,dx \\ &= \frac{1}{\lambda}\int_0^{\lambda} \frac{1}{1-x}\,dx \\ &= \frac{1}{\lambda}\ln\frac{1}{1-\lambda} \end{aligned} \tag{6.5}$$

A similar calculation can be performed for closed hashing with linear probing. This is left as an assignment for the student.

Assignment 6.4

For closed hashing with linear probing prove that:

$$S(\lambda) = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \tag{6.6a}$$

and

$$U(\lambda) = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right) \tag{6.6b}$$

Analysis of open hashing

Let us recall the case of chaining. In chaining, we move to the linked list before doing any probes. Suppose that a list contains n key values. Assuming that the key values are equally probable in any list, the expected number of key values on any list is n/h, h being the size of the hash table. This is nothing but l, the load factor. Now, if the list contains n items, the number of key comparisons for an unsuccessful search is n. Thus, the average number of probes for an unsuccessful search is

$$U(\lambda) = \lambda \tag{6.7}$$

Now, suppose the search is successful. From the analysis of sequential search over a list of n items, we can write

$$\text{Number of comparisons } = \frac{1}{n}\sum_{i}^{n} i = \frac{n+1}{2} \tag{6.8}$$

Assume that an item is equally probable in any place. Since the average number of key values in any list is λ, the average number of probes in a successful search is

$$S(\lambda) = \frac{\lambda + 1}{2} \tag{6.9}$$

We can draw several conclusions from the results thus obtained. Let us draw a graph (Figure 6.11) for these results. From this graph, the following points are evident:

1. Open hashing always requires fewer probes than closed hashing.
2. Chaining is especially advantageous when the load factor is significantly low.
3. With closed hashing and successful search, linear probing is not significantly slower if λ is high. For unsuccessful searches, however, clustering will occur which quickly degenerates into a long sequential search.

We might therefore conclude that if searches are quite likely to be successful and the load factor is moderate, closed hashing is quite satisfactory, but in other circumstances open hashing is promising.

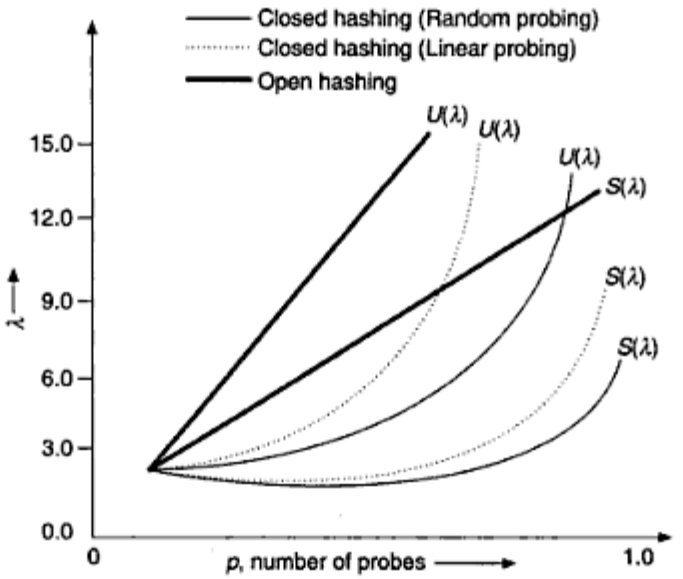


Figure 6.11 Comparison of various collision resolution techniques.

6.5 PROBLEMS TO PONDER

6.1 Suppose there is a hash table of size H. If λ be the load factor and ω be the word size for a key value, then find the total storage space required for the following cases:

(a) Open hashing (assume that one word is required for a link field)

(b) Closed hashing.

6.2 A hash function is defined as $H(k) = r_i$, where $r_1, r_2, ..., r_n$ is a sequence of random numbers between 1 and n (each integer appears exactly once).

(a) Prove that if the hash table is not full then this hashing always resolves collision.

(b) Does this technique eliminate clustering?

(c) If l be the load factor of the table, what is the expected time for a

(i) successful search?

(ii) unsuccessful search?

6.3 In quadratic hashing, the probes are carried out in the sequence

$$H(k) + q^2,\ H(k) + (q-1)^2,\ ....,\ H(k) + 1,\ H(k),\ H(k) - 1,\ ...,\ H(k) - q^2$$

where $q = (h-1)/2$, h being the size of the hash table. Prove that this method resolves collision and avoids clustering.

6.4 In open hashing, we can save time if the nodes in chain are kept in order by key value. How many probes, on an average will be done in the case of

(a) unsuccessful search?

(b) successful search?

6.5 The hash function should be such that it can be calculated very quickly.

(a) Show how if i^2 is known then $(i + 1)^2$ can be obtained from it by addition only.

(b) Show that the following expression generates random numbers between 1 and m, if m and c are prime to each other:

$$y = (y + c) \text{ MOD } m \text{ expression}$$

Assume a suitable starting value for y.

REFERENCES

Bell, J., A hash code eliminating secondary clustering, The quadratic quotient method, *Communication of the ACM*, 13, 1970.

Enbody, R.J. and H.C. Du, Dynamic hashing schemes, *Computing Surveys*, 20, 1988.

Gonnet, G.H. and R. Baeza Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Massachusetts, 1988.

Gotlieb, C.C. and L.R. Gotlieb, *Data Types and Structures*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

Guibas, L.J. and E. Szemerdi, The analysis of double hashing, *Sciences*, 16, 1978.

Knuth, D.E., Sorting and Searching, *The Art of Computer Programming*, 3, Addison-Wesley, Reading, Massachusetts, 1984.

Maurrer, W.D. and T.G. Lewis, Hash table methods, *Computing Surveys*, 7, 1995.

McKenzie, B.J., R. Harries, and T. Bell, Selecting a hashing algorithm, *Software Practice and Experience*, 20, 1990.

Morris, R., Scatter storage techniques, *Communication of the ACM*, 11, 1998.

7

Trees

So far we have learnt about arrays, stacks, queues and linked lists, which are known as linear data structures. These are termed *linear* because the elements are arranged in a linear fashion (that is, one-dimensional representation). Another very useful data structure is the *tree*, where the elements appear in a non-linear fashion, which requires a two-dimensional representation. Figure 7.1 is an example of such a representation.

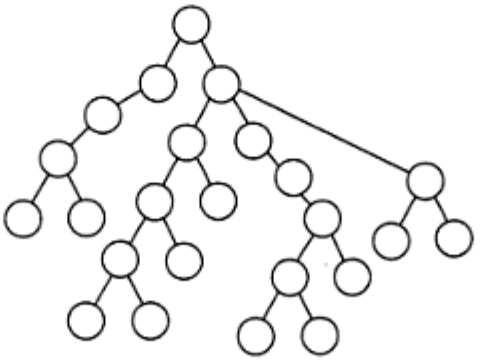


Figure 7.1 Tree—a non-linear representation of data.

There are numerous examples where a tree structure is the efficient means to maintain and manipulate data. In general, where the hierarchy relationship among data is to be preserved, a

tree is used. Figure 7.2 shows a family hierarchy of various members represented as a tree which maintains the relationship among them.

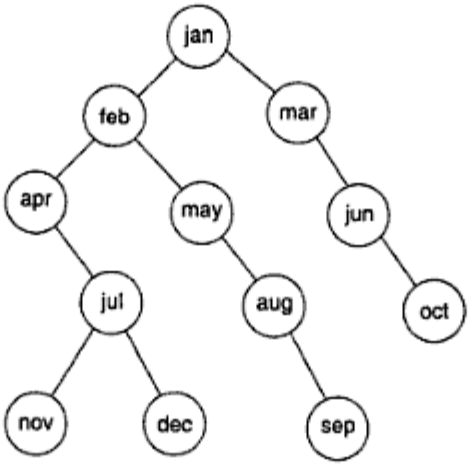


Figure 7.2 A family hierarchy in the form of a tree.

This hierarchy not only gives the ancestors and successors of a family member, but some other information too. For example, if we assume the left-side member as a female and the right-side member as a male then sister, brother, uncle, aunt, grandfather, grandmother, etc. can also be implied automatically.

As another example, let us consider an algebraic expression

$$X = (A - B)/((C * D) + E)$$

where different operators have their own precedence value. A tree structure to represent the same is shown in Figure 7.3. Here, operations having the highest precedence are at the lowest level; everything is stated explicitly as to which operands is for which operator. Moreover, associativity can easily be imposed if we place the operators on the left or right side.

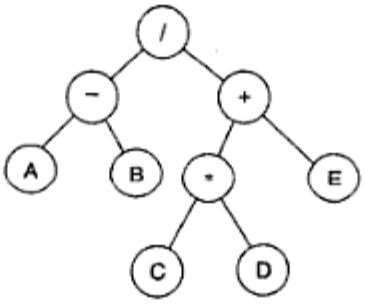


Figure 7.3 An algebraic expression in the form of a tree.

These two examples illustrate how powerful this data structure is to maintain a lot of information automatically. Besides, there are other advantages of this data structure such as insertion, deletion, searching, etc. which are more efficient in trees than in linear data structures.

Before going to study this data structure, let us introduce the basic terminologies of a tree which will be referred to in subsequent discussions.

7.1 BASIC TERMINOLOGIES

Node. This is the main component of any tree structure. The concept of the node is the same as that used in a linked list. A *node* of a tree stores the actual data and links to the other node. Figure 7.4(a) represents the structure of a node.

Parent. The *parent* of a node is the immediate predecessor of a node. Here, X is the parent of Y and Z. See Figure 7.4(b).

Child. If the immediate predecessor of a node is the parent of the node then all immediate successors of a node are known as *child*. For example, in Figure 7.4(b), Y and Z are the two child of X. The child which is on the left side is called the *left child* and that on the right side is called the *right child*.

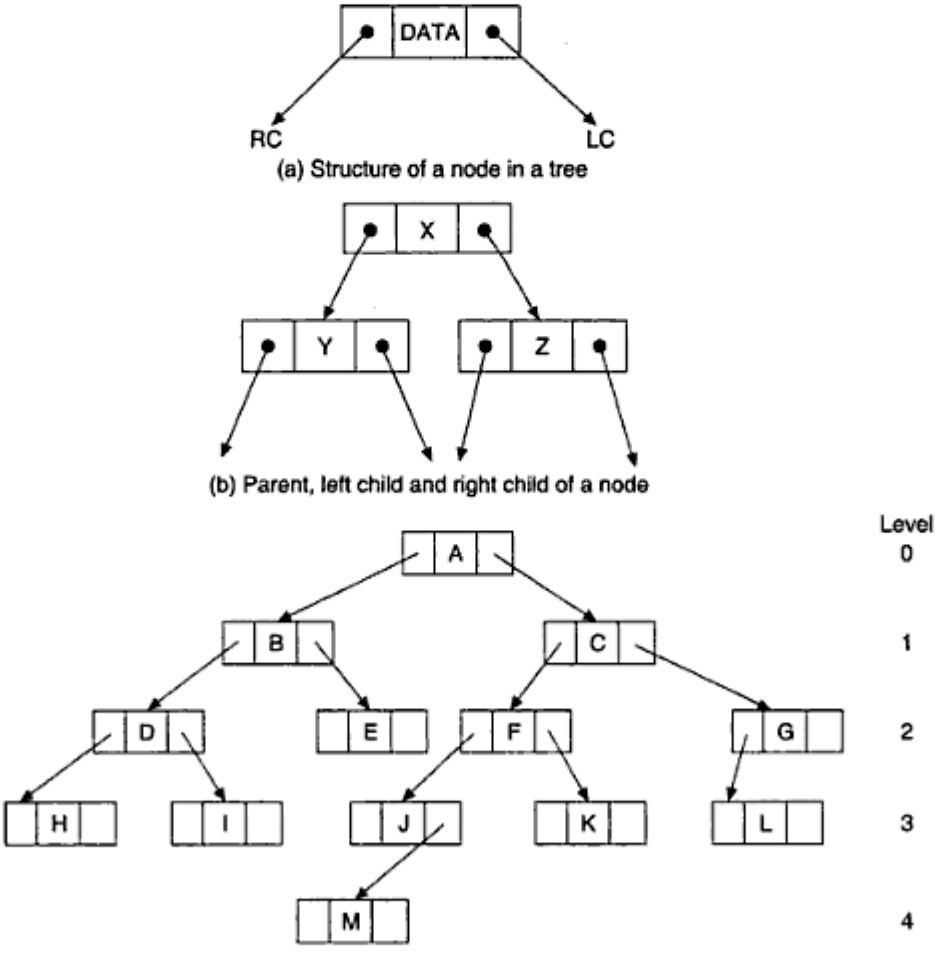


Figure 7.4 A tree and its various components.

Link. This is a pointer to a node in a tree. For example, as shown in Figure 7.4(a), LC and RC are two *links* of a node. Note that there may be more than two links of a node.

Root. This is a specially designated node which has no parent. In Figure 7.4(c), *A* is the *root* node.

Leaf. The node which is at the end and does not have any child is called *leaf* node. In Figure 7.4(c), *H*, *I*, *K*, *L*, and *M* are the leaf nodes. A leaf node is also alternatively termed terminal node.

Level. *Level* is the rank in the hierarchy. The root node has level 0. If a node is at level l, then its child is at level $l + 1$ and the parent is at level $l - 1$. This is true for all nodes except the root node, being at level zero. In Figure 7.4(c), the levels of various nodes are depicted.

Height. The maximum number of nodes that is possible in a path starting from the root node to a leaf node is called the *height* of a tree. For example, in Figure 7.4(c), the longest path is *A–C–F–J–M* and hence the height of this tree is 5. It can be easily seen that $h = l_{max} + 1$, where h is the height and l_{max} is the maximum level of the tree.

Degree. The maximum number of children that is possible for a node is known as the *degree* of a node. For example, the degree of each node of the tree as shown in Figure 7.4(c) is 2.

Sibling. The nodes which have the same parent are called *siblings*. For example, in Figure 7.4(c), *J* and *K* are siblings.

Different texts use different terms for the above defined terms, such as *depth* for height, *branch* or *edge* for link, *arity* for degree, *external* node for leaf node and *internal* node for a node other than a leaf node.

Assignment 7.1

(a) Observe the tree given in Figure 7.5. Find the following with reference to this tree:

(i) The height of the tree.
(ii) Level (H), level (C) and level (K).
(iii) Degree of the tree.
(iv) The longest path in the tree.
(v) Parent (M), sibling (I), child (B).

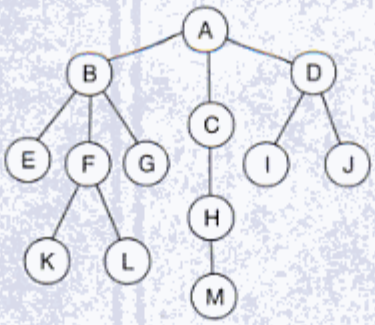


Figure 7.5

(Contd.)

(b) In your computer, there is a directory structure. Starting from the root directory to '.exe' files, draw a tree for the entire file system. Then answer the following:
  (i) What is the height of the directory system?
  (ii) What is the level of the C++ compiler?
  (iii) What is its degree?
  (iv) Which file(s) is/are on the longest path?

7.2 DEFINITION AND CONCEPTS

Let us define a tree. A *tree* is a finite set of one or more nodes such that:

(i) there is a specially designated node called the root,

(ii) the remaining nodes are partitioned into n ($n > 0$) disjoint sets T_1, T_2, ..., T_n, where each T_i ($i = 1, 2, ..., n$) is a tree; T_1, T_2, ..., T_n are called subtrees of the root.

To illustrate the above definition, let us consider the sample tree shown in Figure 7.6.

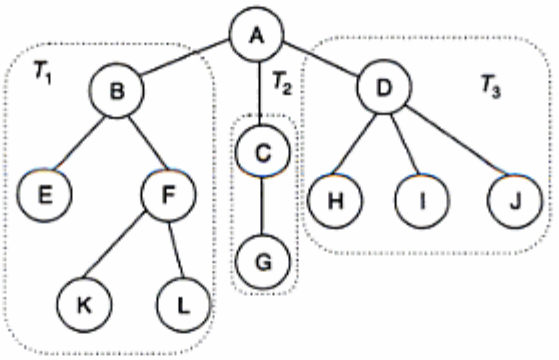
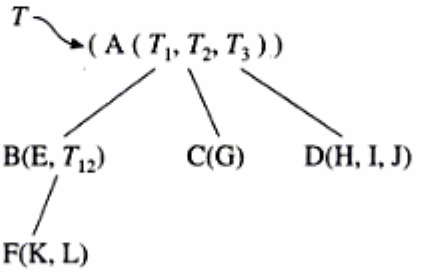


Figure 7.6 A sample tree T.

In the sample tree T, there is a set of 12 nodes. Here, A is a special node being the root of the tree. The remaining nodes are partitioned into 3 sets T_1, T_2, and T_3; they are sub-trees of the root node A. By definition, each sub-tree is also a tree. Observe that a tree is defined recursively. The same tree can be expressed in a string notation as shown below:



Or more precisely,

$$T = (A(B(E, F(K, L))), C(G), D(H, I, J))$$

Assignment 7.2
Draw a tree with the given string notation:

(A (B (C (E), F, D), G (H, (I(J)))))

7.2.1 Binary Trees

A *binary tree* is a special form of a tree. Contrary to a general tree, a binary tree is more important and frequently used in various applications of computer science. Like a general tree, a binary tree can also be defined as a finite set of nodes, such that:

(i) T is empty (called the empty binary tree), or

(ii) T contains a specially designated node called the root of T, and the remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called the left sub-tree and the right sub-tree, respectively.

Figure 7.7 depicts a sample binary tree.

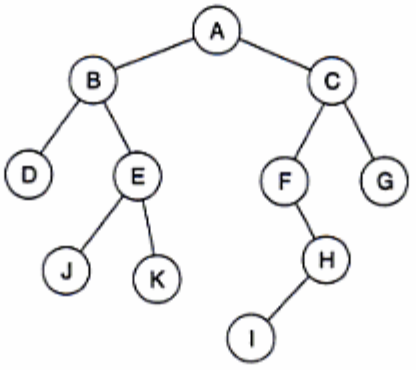


Figure 7.7 A sample binary tree with 11 nodes.

One can easily notice the main difference between the definitions of a tree and a binary tree. A tree can never be empty but a binary tree may be empty. Another difference is that in the case of a binary tree a node may have at most two children (that is, a tree having degree = 2), whereas in the case of a tree, a node may have any number of children.

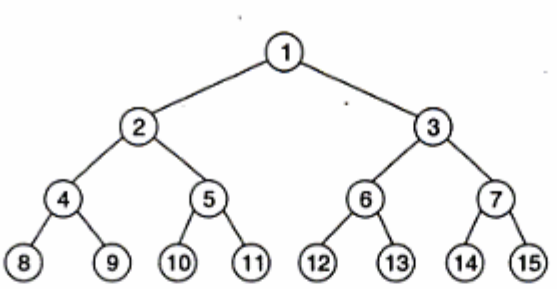
Two special situations of a binary tree are possible: full binary tree and complete binary tree.

Full binary tree

A binary tree is a *full binary* tree if it contains the maximum possible number of nodes at all levels. Figure 7.8(a) shows such a tree with height 4.

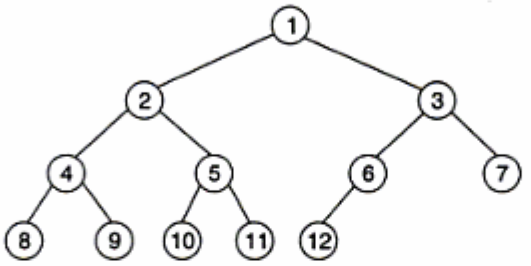
Complete binary tree

A binary tree is said to be a *complete binary* tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible. Figure 7.8(b) depicts a complete binary tree.



Level	Nodes
0	1
1	2
2	4
3	8

(a) A full binary tree of height 4



Level	Nodes
0	1
1	2
2	4
3	5

(b) A complete binary tree of height 14

Figure 7.8 Two special cases of binary trees.

Observe that the binary tree represented in Figure 7.7 is neither a full binary tree nor a complete binary tree.

7.2.2 Properties of a Binary Tree

A binary tree possesses a number of properties. These properties are very much useful and are listed below as Lemmas 7.1–7.7.

Lemma 7.1
In any binary tree, the maximum number of nodes on level l is 2^l, where $l \geq 0$.

Proof: The proof of the above lemma can be done by induction on l. The root is the only node on level $l = 0$. Hence, the maximum number of nodes on level $l = 0$ is $2^0 = 1 = 2^l$.

Suppose for all i, $0 \leq i < l$ and for some l, the above formula is true, that is, the maximum number of nodes at level i is 2^i. Since each node in a binary tree has degree 2, so from each node at level i, there may be at most 2 nodes at level $i + 1$. Thus, the maximum number of nodes at level $i + 1$ is $2 \times 2^i = 2^{i+1}$. Therefore, if the formula is true for any i, then it is also true for $i + 1$.

Hence, the proof (see Figure 7.9).

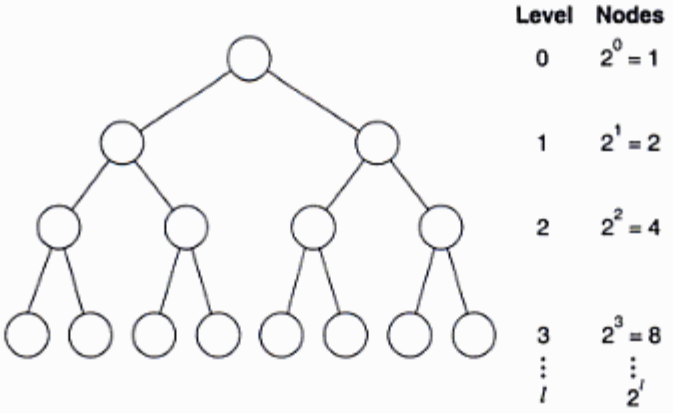


Figure 7.9 Binary tree showing Lemma 7.1.

Lemma 7.2
The maximum number of nodes possible in a binary tree of height h is $2^h - 1$.

Proof: The maximum number of nodes is possible in a binary tree if the maximum number of nodes are present at each level. Thus, the maximum number of nodes in a binary tree of height h is

$$n = \sum_{i=0}^{l_{max}} 2^l \quad \text{(where } l_{max} \text{ is the maximum level of the tree)}$$

$$= \frac{2^{l_{max}+1} - 1}{2 - 1} \quad \text{(using the formula of a geometric series)}$$

$$= 2^{l_{max}+1} - 1$$

From the definition of height, we have $h = l_{max} + 1$, hence we can write

$$n_{max} = 2^h - 1 \tag{7.1}$$

Lemma 7.3
The minimum number of nodes possible in a binary tree of height h is h.

Proof: A binary tree has the minimum number of nodes if each level has the minimum number of nodes. The minimum number of nodes possible at every level is only one when every parent has one child. Such kinds of trees called *skew binary trees* are shown in Figure 7.10.

Thus, a *skew binary* tree is the tree having only *one* path which contains h number of nodes if h is its height. Hence, $n_{min} = h$.

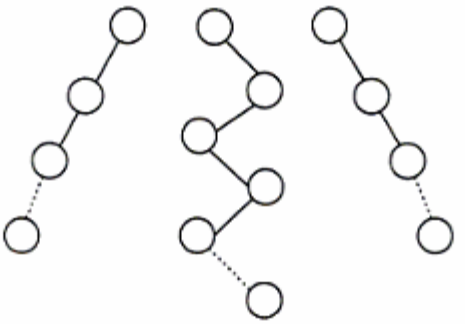


Figure 7.10 Skew binary trees containing the minimum number of nodes.

Lemma 7.4

For any non-empty binary tree, if n is the number of nodes and e is the number of edges, then $n = e + 1$.

Proof: By the induction method, we have

Number of nodes, n	*Number of edge e*	$n = e + 1$
1	0	$1 = 0 + 1$
2	1	$2 = 1 + 1$
3	2	$3 = 2 + 1$
4	3	$4 = 3 + 1$
⋮	⋮	⋮
n'	$n' - 1$	$n' = (n' - 1) + 1$

Continuing this, let the above be true for any number of nodes n'. Thus, $n' = e' + 1$.

Now, if we add one more node into the binary tree having n' nodes, then it will increase one more edge in the binary tree. Thus,

$$n' + 1 = (e' + 1) + 1$$

or

$$n' + 1 = ((n' - 1) + 1) + 1$$

where $e' = n' - 1$; therefore

$$n' + 1 = (n' + 1 - 1) + 1$$

This implies that, if the formula is true for any n, then it is also true for $n + 1$. Hence, $n = e + 1$ is true.

Lemma 7.5

For any non-empty binary tree T, if n_0 is the number of leaf nodes (degree = 0) and n_2 is the number of internal nodes (degree = 2), then $n_0 = n_2 + 1$.

Proof: Suppose n is the total number of nodes in T and n_i is the number of nodes having degree i, $0 \leq i \leq 2$. So, we have

$$n = n_0 + n_1 + n_2 \tag{7.2}$$

as T is a binary tree and no other kinds of nodes are possible.

If e be the number of edge in T, then

$$e = n_0 \times 0 + n_1 \times 1 + n_2 \times 2$$

or

$$e = n_1 + 2n_2 \tag{7.3}$$

Again, from Lemma 7.4, we have

$$n = e + 1 \tag{7.4}$$

Thus, from (7.2) and (7.3), we can write

$$n = 1 + n_1 + 2n_2 \tag{7.5}$$

And from (7.2) and (7.5), we have

$$n_0 + n_1 + n_2 = 1 + n_1 + 2n_2$$

or

$$n_0 = n_2 + 1 \tag{7.6}$$

Hence, the result.

Lemma 7.6

The height of a complete binary tree with n number of nodes is $\lceil \log_2 (n+1) \rceil$.

Proof: Let h be the height of the complete binary tree. Then we can write the following inequality:

$$n \leq 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1}$$

or

$$n \leq 2^h - 1 \tag{7.7}$$

or

$$2^h \geq n + 1$$

Taking logarithm on both sides, we get

$$h \geq \log_2(n + 1)$$

or

$$h = \lceil \log_2(n + 1) \rceil \tag{7.8}$$

Hence, the result.

Lemma 7.7

The total number of binary trees possible with n nodes is

$$\frac{1}{n+1} {}^{2n}C_n \tag{7.9}$$

Proof: The proof of this lemma is beyond the scope of this book; it can be obtained from the *The Art of Computer Programming: Fundamental Algorithms*, Vol. 1, D.E. Knuth, Addison-Wesley, Reading, Massachusetts, 1984.

Some other properties of binary tree are given in Assignment 7.3.

Assignment 7.3

Prove that

(a) In a binary tree of height h, there are at most 2^{h-1} leaf nodes.

(b) The maximum and minimum levels that are possible for a binary tree with n nodes are

$$l_{max} = n - 1 \tag{7.10a}$$

$$l_{min} = \lceil \log_2(n + 1) - 1 \rceil \tag{7.10b}$$

Hint: A binary tree has a large value for a level when it is a skew tree and the minimum value for a level when it is a complete binary tree.

(c) A binary tree (other than the skew binary tree) with n nodes must have at least one path of length $\lceil \log_2 (n+1) \rceil$.

Assignment 7.4

(a) In an m-ary tree (that is, the degree of the tree = m), if n_i is the number of nodes of degree i (i = 0, 1, ..., m), then prove that

$$n_0 = 1 + \sum_{i=2}^{m} (i-1) n_i \tag{7.11}$$

Note: This is a general formula for Lemma 7.5, when m = 2.

(b) Show the various binary trees possible with n = 3, 4 and 5 and then verify the Lemma 7.7.

(c) Construct a complete binary tree with 26 alphabets and hence verify the Lemma 7.6 related to it.

7.3 REPRESENTATIONS OF BINARY TREE

A (binary) tree must represent a hierarchical relationship between a parent node and (at most two) child nodes. There are two common methods used for representing this conceptual structure. One is implicit approach called *linear* (or *sequential*) representation, where using an array we do not require the overhead of maintaining pointers (links). The other is explicit approach known as *linked* representation that uses pointers. Whatsoever be the representation, the main objective is that one should have direct access to the root node of the tree, and for any given node, one should have direct access to the children of it.

7.3.1 Linear Representation of a Binary Tree

This type of representation is static in the sense that a block of memory for an array is allocated before storing the actual tree in it, and once the memory is allocated, the size of the tree is restricted as permitted by the memory.

In this representation, the nodes are stored level by level, starting from the zero level where only the root node is present. The root node is stored in the first memory location (as the first element in the array).

Following rules can be used to decide the location of any node of a tree in the array (assuming that they array index starts from 1):

1. The root node is at location 1.
2. For any node with index i, $1 < i \leq n$ (for some n):

   (a) $\text{PARENT}(i) = \lfloor i/2 \rfloor$ (7.12a)

   For the node when $i = 1$, there is no parent.

   (b) $\text{LCHILD}(i) = 2 * i$ (7.12b)

   If $2 * i > n$, then i has no left child.

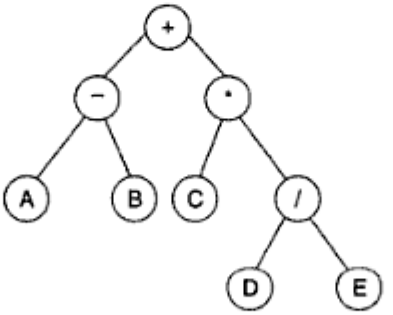
   (c) $\text{RCHILD}(i) = 2 * i + 1$ (7.12c)

   If $2 * i + 1 > n$, then i has no right child.

For example, let us consider the case of representation of the following expression in the form of a tree:

$$(A - B) + C * (D/E)$$

The binary tree will appear as in Figure 7.11(a). The representation of the same tree using an array is shown in Figure 7.11(b).



(a) A binary tree

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+	−	*	A	B	C	/	·	·	·	·	·		D	E	·

(b) Array representation of the binary tree

Figure 7.11 Sequential representation of a binary tree.

The next question that arises is how can the size of an array be estimated? This value can be obtained easily if the binary tree is a full binary tree. As we know, from Lemma 2, a binary tree of height h can have at most $2^h - 1$ nodes. So, the size of the array to fit such a binary tree is $2^h - 1$.

To understand this, a full binary tree and the index of its various nodes when stored in an array are illustrated in Figure 7.12.

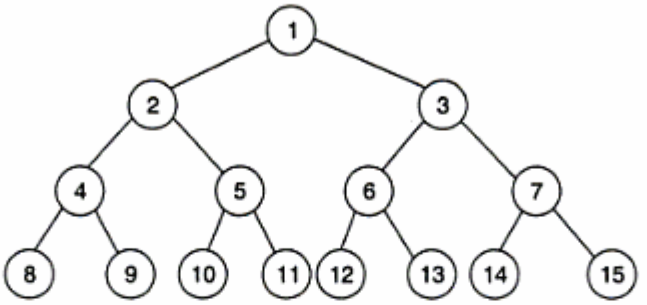


Figure 7.12 A full binary tree of height 4 and labels of the nodes representing array locations of nodes.

One can easily realize the fact that if the tree is a full binary tree then there will be an efficient use of storage in the array representation (no array location will be left empty); on the other hand, for a binary tree other than a full binary tree there is a wastage of memory.

Lemma 7.8 gives us an estimation about the maximum and minimum sizes of the array to store a binary tree with n nodes.

Lemma 7.8

The maximum and minimum sizes that an array may require to store a binary tree with n number of nodes are

$$\text{Size}_{\max} = 2^n - 1 \tag{7.13a}$$

$$\text{Size}_{\min} = 2^{\lceil \log_2 (n+1) \rceil} - 1 \tag{7.13b}$$

Proof If the height of a binary tree denotes the maximum number of nodes in the longest path of the tree, then for a tree with n nodes the maximum height possible is $h_{\max} = n$. Such a binary tree is termed a *skew binary* tree (see Figure 7.13).

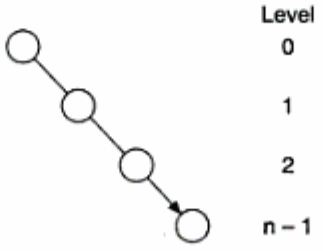


Figure 7.13 A skew binary tree with maximum height.

If we store such a binary tree, then it can be seen that the first location is for the root node, the third location $(2^2 - 1)$ for the right child of the root node (second node), the 7th $(2^3 - 1)$ location for the right child of the second node (third node), and so on. Thus, for a node at level i—it is actually the $(i + 1)$th node—its location in the array is at $2^{i+1} - 1$. So, the last node (nth node) which is at the $(n - 1)$th level will be at the location $(2^n - 1)$. This is therefore the maximum size of the array. Hence,

$$\text{Size}_{\max} = 2^n - 1 \tag{7.14a}$$

Now, when the tree is a full (or complete) binary tree, then we need a minimum sized array to accommodate the entire tree. In the case of a full or complete binary tree, with n nodes, the minimum height that is possible is $h_{\min} = \lceil \log_2(n + 1) \rceil$. In that case, the last element will be stored at $(2^{h_{\min}} - 1)$th location. Hence, the minimum size required is

$$\text{Size}_{\min} = 2^{h_{\min}} - 1 = 2^{\lceil \log_2 (n+1) \rceil} - 1 \tag{7.14b}$$

Note: The maximum and minimum sizes of an array required to store a binary tree can be expressed, in general, as

$$\text{Size} = 2^h - 1 \tag{7.15}$$

where h is the height of the binary tree. Here, if h is minimum then the minimum sized array will be computed, and if h is maximum then the maximum sized array will be computed.

Advantages of the sequential representation of a binary tree

The advantages of the sequential representation of a binary tree are:

1. Any node can be accessed from any other node by calculating the index and this is efficient from execution point of view.
2. Here, only data are stored without any pointers to their successor or ancestor which are mentioned implicitly.
3. Programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only means to store a tree.

Disadvantages of the sequential representation of a binary tree

With these potential advantages, there are disadvantages as well. These are:

1. Other than the full binary trees, the majority of the array entries may be empty.
2. It allows only static representation. It is in no way possible to enhance the tree structure if the array size is limited.
3. Inserting a new node to the tree or deleting a node from it are inefficient with this representation, because these require considerable data movement up and down the array which demand excessive amount of processing time.

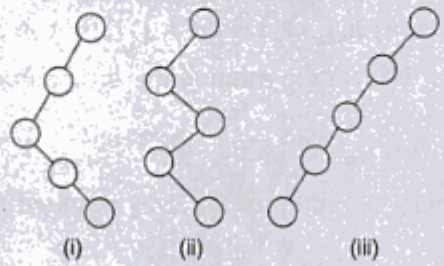
Assignment 7.5

(a) Draw the tree structure whose array representation is given in Figure 7.14(a).

(b) Verify the $Size_{max}$ as stated in Lemma 7.8 for the skew binary trees in Figure 7.14(b).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	·	C	·	·	·	D	·	·	·	·	·	·	·	E

(b) Array representation of a tree structure



(b) Skew binary tree

Figure 7.14 Array and skew sequential representation of a tree.

7.3.2 Linked Representation of a Binary Tree

In spite of simplicity and ease of implementation, the linear representation of binary trees has a number of overheads. In the linked representation of binary trees, all these overheads are taken care of. The linked representation assumes the structure of a node to be as shown in Figure 7.15.

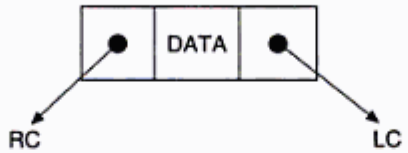
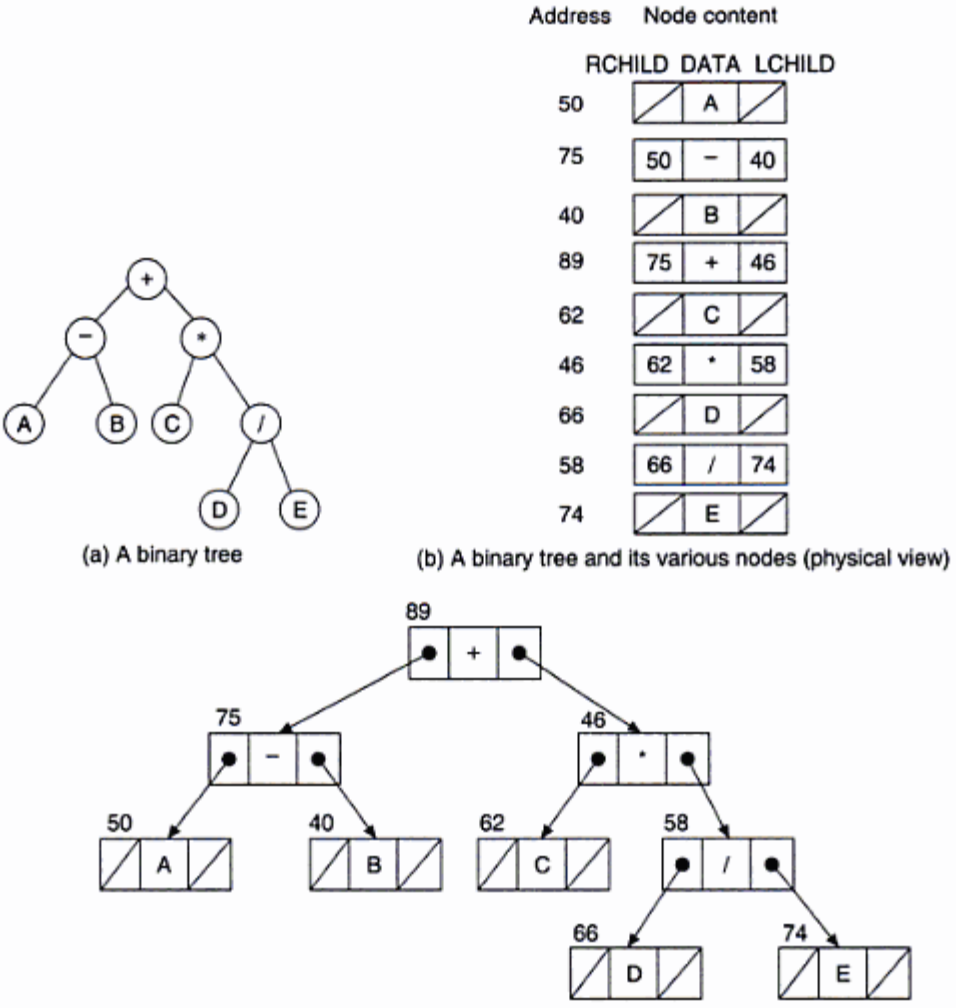


Figure 7.15 Structure of a node in linked representation.

Here, LC and RC are the two link fields used to store the addresses of left child and right child of a node; DATA is the information content of the node. With this representation, if one knows the address of the root node then from it any other node can be accessed. The two forms, that is, 'tree' structure and 'linked' structure look almost similar. This similarity implies that the linked representation of a binary tree very closely resembles the logical structure of the data involved. This is why, all the algorithms for various operations with this representation can be easily defined.

Using the linked representation, the tree with 9 nodes given in Figure 7.16(a) will appear as shown in Figure 7.16(b)–(c).



(a) A binary tree

(b) A binary tree and its various nodes (physical view)

(c) Logical view of the linked representation of a binary tree

Figure 7.16 Linked representation of a binary tree.

Another advantage of this representation is that it allows dynamic memory allocation; hence the size of the tree can be changed as and when the need arises without any limitation except when the limitation of the availability of the total memory is the problem.

However, so far as the memory requirement for a tree is concerned, the linked representation uses more memory than that required by the linear representation. Linked representation requires extra memory to maintain the pointers. Some pointers though with null values, they too need memory to store them. Lemma 7.9 estimates the number of such null links in a binary tree.

Lemma 7.9

In a linked representation of a binary tree, if there are n number of nodes then the number of null links $\lambda = n + 1$.

Proof By the method of induction. Let R be the pointer to the root node of a binary tree T. If T is empty, that is $n = 0$ then R = null, thus the number of null links, $\lambda = 0 + 1 = n + 1$. For a single node, that is $n = 1$, $\lambda = 2 = 1 + 1 = n + 1$. Similarly, for a tree T having two nodes, $\lambda = 3 = 2 + 1 = n + 1$. Here, it is true for $n = 2$. Thus, the formula is true for $n = 0$, 1 and 2 (see Figure 7.17). Let it be true for any $n = m$. Therefore, we can write $\lambda = m + 1$. Now, if we add one more node into T with m nodes, then the number of nodes will be $n' = m + 1$. The addition of the new node changes one null link into a non-null value whereas it incorporates two null links of its own; thus the net increase in null links by this new node is 1. Hence, we have $\lambda = (m + 1) + 1$ or $\lambda = n' + 1$. This shows that if the formula is true for m then it is also true for $m + 1$. As m is an arbitrary number, the above relation is proved.

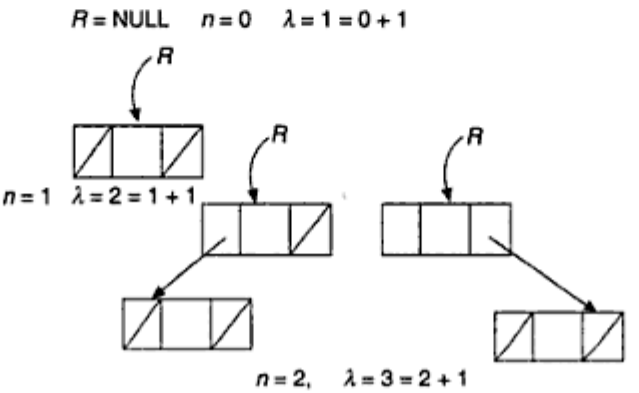


Figure 7.17 Induction for $\lambda = n + 1$ when $n = 0$, 1 and 2.

7.3.3 Physical Implementation of a Binary Tree in Memory

In this section, we will see how simple it is to implement a binary tree either using a sequential (array) or using a linked representation. In fact, the implementation is just a straightforward translation from the definition of a binary tree to algorithmic description.

The algorithm *BuildTree_1*(...) is to build a tree using an array of element type DATA and the algorithm *BuildTree_2* (...) is to build a tree using a linked structure. Both the algorithms will build a tree with user interaction.

The algorithm *BuildTree_1* (...) assumes an array A of suitable size to store all the data elements in a binary tree. The algorithm *BuildTree_2* (...) assumes a node structure which is the same as stated in Section 7.3.2. Both the algorithms follow the recursive definition.

Algorithm BuildTree_1

Input: *ITEM* is the data content of the node *I*. //A binary tree currently with node at *I*

Output: A binary tree with two sub-trees of node *I*.

Data structure: Array representation of tree.

Steps:

```
1.  If (I ≠ 0) then                                   // If the tree is not empty
2.      A[I] = ITEM                  // Store the content of the node I into the array A
3.      Node I has left sub-tree (Give option = Y/N)?
4.      If (option = Y) then                          // If node I has left sub tree
5.          BuildTree_1 (2 * I, NEWL)     // Then it is at 2*I with next item as NEWL
6.      Else
7.          BuildTree_1 (0, NULL)                     // Empty sub-tree
8.      EndIf
9.      Node I has right sub-tree (Give option = Y/N)?
10.     If (option = Y)                               // If node I has right sub-tree
11.         BuildTree_1 (2*I+1, NEWR)     //Then it is at 2*I+1 with next item as NEWR
12.     Else
13.         BuildTree_1 (0, NULL)                     // Empty sub-tree
14.     EndIf
15. EndIf
16. Stop
```

Algorithm BuildTree_2

Input: ITEM is the content of the node with pointer *PTR*.
Output: A binary tree with two sub-trees of node *PTR*.
Data structure: Linked list structure of binary tree.

Steps:

```
1.  If (PTR ≠ NULL) then                              // If the tree is not empty
2.      PTR→DATA = ITEM                  // Store the content of node at PTR
3.      Node PTR has left sub-tree (Give option = Y/N)?
4.      If (option = Y) then
5.          lcptr = GetNode(NODE)                 // Allocate memory for the left child
6.          PTR→LC = lcptr                        // Assign it to Left link
7.          BuildTree_2 (lcptr, NEWL)     // Build left sub-tree with next item as NEWL
8.      Else
9.          lcptr = NULL
10.         PTR→LC = NULL                         // Assign for an empty left sub-tree
11.         BuildTree_2(lcptr, NULL)                  // Empty sub-tree
12.     EndIf
13.     Node PTR has right sub-tree (give option = Y/N)?
14.     If (option = Y) then
15.         rcptr = GetNode (NODE)                // Allocate memory for the right child
16.         PTR→RC = rcptr                        // Assign it to right link
17.         BuildTree_ 2(rcptr, NEWR)     // Build right sub-tree with next item as NEWR
```

(*Contd.*)

```
18.     Else
19.         rcptr = NULL
20.         PTR→RC = NULL                    // Assign for an empty right sub-tree
21.         BuildTree_ 2(rcptr, NULL)
22.     EndIf
23. EndIf
24. Stop
```

7.4 OPERATIONS ON A BINARY TREE

The major operations on a binary tree can be listed as follows:

1. *Insertion.* To include a node into an existing (may be empty) binary tree.
2. *Deletion.* To delete a node from a non-empty binary tree.
3. *Traversal.* To visit all the nodes in a binary tree.
4. *Merge.* To merge two binary trees into a larger one.

Some other special operations are also known for special cases of binary trees. These will be mentioned later on. Now, let us discuss the above-mentioned operations for general binary trees.

7.4.1 Insertion

With this operation, a new node can be inserted into any position in a binary tree. Inserting a node as an internal node is generally based on some criteria which is usually in the context of a special form of a binary tree. We will discuss here a simple case of insertion of a node as an external node. Figure 7.18 shows how a node containing data content *G* can be inserted as a left child of a node having data content *E*. Two algorithms for two different storages of representations (viz. sequential storage and linked storage) are stated below.

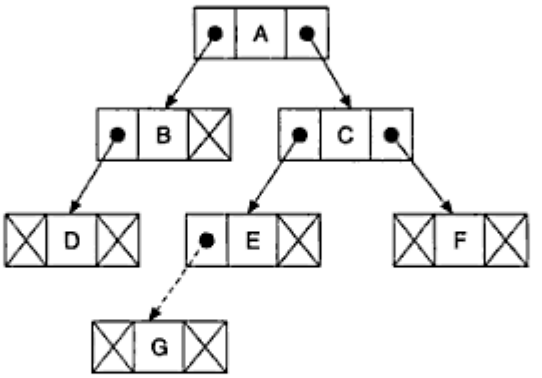


Figure 7.18 Insertion of a node as an external node into a binary tree.

The insertion procedure, in fact, is a two-step process: the first to search for the existence of a node in the given binary tree after which an insertion is made, and the second is to establish a link for the new node.

Insertion into a sequential representation of a binary tree (as a leaf node)

The following algorithm uses a recursive procedure *Search_SEQ* to search for a node containing data *KEY*.

Algorithm InsertBinaryTree_SEQ

Input: *KEY* be the data of a node after which a new node has to be inserted with data *ITEM*.
Output: Newly inserted node with data *ITEM* as a left or right child of the node with data *KEY*.
Data structure: Array *A* storing the binary tree.

Steps:

```
1.  l = Search_SEQ(1, KEY)                          // Search for the key node in the tree
2.  If (l = 0) then
3.      Print "Search is unsuccessful : No insertion"
4.      Exit
5.  EndIf                                                          // Quit the execution
6.  If (A[2 * l] = NULL) or (A[2 * l + 1] = NULL) then   // If the key node has empty link(s)
7.      Read option to read as left (L) or right (R)                // child (give option = L/R)
8.      If (option = L) then
9.          If A[2 * l] = NULL then                                    // Left link is empty
10.             A[2 * l] = ITEM                                    // Store it in the array A
11.         Else                                          // Cannot be inserted as left child
12.             Print "Desired insertion is not possible"     // as it already has a left child
13.             Exit                                  // Return to the end of the procedure
14.         EndIf
15.     EndIf
16.     If (option = R) then                                       // Move to the right side
17.         If (A[2 * l + 1] = NULL) then                          // Right link is empty
18.             A[2 * l + 1] = ITEM                                // Store it in the array A
19.         Else                                          // Cannot be inserted as right child
20.             Print 'Desired operation is not possible'      //as it already has a left child
21.             Exit                                  // Return to the end of the procedures
22.         EndIf
23.     EndIf                          // Key node has both the left child and the right child
24. Else                                          // Key node does not have any empty link
25.     Print "ITEM cannot be inserted as a leaf node"
26. EndIf
27. Stop
```

The procedure *Search_SEQ*(...) can be defined recursively as below:

Algorithm Search_SEQ

Input: *KEY* be the item of search, *INDEX* being the index of the node from where search will start.
Output: *LOCATION*, where the item *KEY* is located, if any.
Data structure: Array *A* storing the binary tree. SIZE denotes the size of *A*.

Steps:

```
1.  i = INDEX                                          // Start from the root node
2.  If (A[i] ≠ KEY) then                               // The present node is not the key node
3.      If (2 * i ≤ SIZE) then                         // If left sub-tree is not exhausted
4.          Search_SEQ(2 * i, KEY)                     // Search in the left sub-tree
5.      Else                                           // Left sub-tree is exhausted and KEY not found
6.          If (2 * i + 1 ≤ SIZE) then                 // If right sub-tree is not exhausted
7.              Search_SEQ (2 * i + 1, KEY)            // Search in the right sub-tree
8.          Else          // The KEY is found neither in left sub-tree nor in right sub-tree
9.              Return(0)                              // Return NULL address for unsuccessful search
10.         EndIf
11.     EndIf
12. Else
13.     Return(i)                                      // Return the address where KEY is found
14. EndIf
15. Stop
```

Insertion into a linked representation of a binary tree (as a leaf node)

Let us assume the node structure used in linked representation as stated in Section 7.3.2.

Algorithm InsertBinaryTree_LINK

Input: *KEY* is the data content of the key node after which a new node is to be inserted and *ITEM* is the data content of the new node that has to be inserted.

Output: A node with data component ITEM inserted as an external node after the node having data KEY if such a node exists with empty link(s), that is, either child or both children is/are absent.

Data structure: Linked structure of a binary tree. ROOT is the pointer to the root node.

Steps:

```
1.  ptr = Search_LINK(ROOT, KEY)
2.  If (ptr = NULL) then
3.      Print "Search is unsuccessful: No insertion"
4.      Exit
5.  EndIf
6.  If (ptr→LC = NULL) or (ptr→RC = NULL)      // If either or both link(s) is/are empty
7.      Read option to insert as left (L) or right (R) child (give option = L/R)
8.      If (option = L) then                                   // To insert as left child
9.          If (ptr→LC = NULL) then            // If the left link is empty then insert
10.             new = GetNode(NODE)
11.             new→DATA = ITEM
12.             new→LC = new→RC = NULL
```

(Contd.)

```
13.            ptr→LC = new
14.         Else                                  // The key node already has left child
15.            Print "Insertion is not possible as left child"
16.            Exit                                          // Quit the execution
17.         EndIf
18.      Else                                                    // If option = R)
19.         If (ptr→RC = NULL)                  // If the right link is empty then insert
20.            new = GetNode (NODE)
21.            new→DATA = ITEM
22.            new→LC = new→RC = NULL
23.            ptr→RC = new
24.         Else                                // The key node already has right child
25.            Print "Insertion is not possible as right child"
26.            Exit                                          // Quit the execution
27.         EndIf
28.      Else
29.         Print "The key node already has child"      //The key node has no empty child
30.      EndIf
31.   EndIf
32.   Stop
```

This algorithm uses the procedure **Search_LINK** to search for a node containing data *KEY*; this procedure can be defined recursively as follows:

Algorithm Search_LINK

Input: *KEY* is the item of search, PTR0 is the pointer to the linked list from where the search will start.
Output: *LOCATION*, where the item *KEY* is located, if such *ITEM* exists.
Data structure: Linked structure of binary tree having ROOT as the pointer to the root node.

Steps:

```
1.  ptr = PTR0                                          // Start from a given node
2.  If (ptr→DATA ≠ KEY)                  // If the current node is not the key node
3.     If (ptr→LC ≠ NULL)                              // If the node has left sub-tree
4.        Search_LINK(ptr→LC)                          // Search the left sub-tree
5.     Else                                  // Key is not in the left sub-tree
6.        Return(0)
7.     EndIf
8.     If (ptr→RC ≠ NULL)                             // If the node has right sub-tree
9.        Search_LINK(ptr→RC)                          // Search the right sub-tree
10.    Else                                  // Key is not in the right sub-tree
11.       Return(0)                                     // Return null pointer
```

(Contd.)

```
12.     EndIf
13.  Else
14.     Return(ptr)                    // Return the pointer to the key node
15.  EndIf
16.  Stop
```

Assignment 7.6

(a) Write procedures to form a complete binary tree by inserting nodes one after another. Assume both the representations of binary tree.

(b) A binary tree is given which is represented using a single array. How can the same tree be converted into a linked structure? Write the procedure.

(c) Repeat problem (b) when a given binary tree is represented with a linked structure. Write a procedure to build the same tree which will be stored in an array of suitable size.

7.4.2 Deletion

This operation is used to delete any node from any non-empty binary tree. Like the insertion, the deletion operation also varies from one kind of binary tree to another. Deletion operations in various cases of binary trees will be discussed in due time. Here, we will consider the case of deletion of a leaf node in any binary tree. Figure 7.19 shows how a node containing data *G* can be deleted from a binary tree.

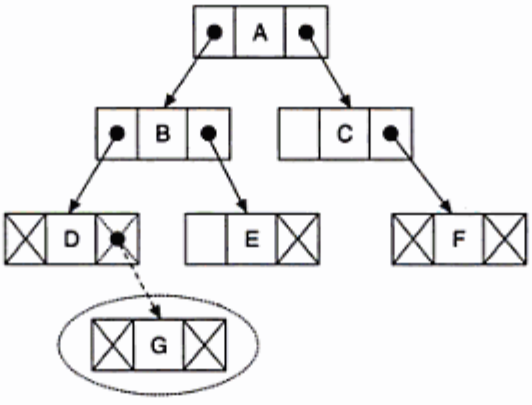


Figure 7.19 Deletion of an external node in a binary tree.

Let us see how the deletion operation can be carried out for the two kinds of storage representation of binary trees. In order to delete a node in a binary tree, it is required to reach at the parent node of the node to be deleted. The link field of the parent node which stores the location of the node to be deleted is then set by a NULL entry. Let us discuss the algorithm individually.

Deleting a leaf node from the sequential representation of binary trees

Let *SIZE* be the total length of the array *A*, where a binary tree is stored. The element that has to be deleted is *ITEM*.

Algorithm DeleteBinTree_SEQ

Input: Given *ITEM* as data of the node with which the node can be identified for deletion.
Output: A binary tree without a node having data *ITEM*.
Data structure: Array *A* storing the binary tree. *SIZE* denotes the size of *A*.

Steps:

```
1.  flag = FALSE                                          // Start from the root node
2.  l = Search_SEQ(1, KEY)                                // Start searching from starting index
3.  If l = 0 Goto Step 10
4.  If (A[2 * l] = NULL) and (A[2 * l + 1] = NULL)        // Test for the leaf node
5.      flag = TRUE                                       // If so, then delete it
6.      A[l] = NULL
7.  Else
8.      Print "The node containing ITEM is not a leaf node"
9.  EndIf
10. If (flag = FALSE)
11.     Print "Node does not exist : No deletion"
12. EndIf
13. Stop
```

Deleting a leaf node from a linked representation of a binary tree

Let *ROOT* be the pointer to the linked list storing a binary tree. The node that has to be deleted contains *ITEM* as data.

Algorithm DeleteBinTree_LINK

Input: A binary tree whose address of the root pointer is known from *ROOT* and *ITEM* is the data of the node identified for deletion.
Output: A binary tree without having a node with data *ITEM*.
Data structure: Linked structure of a binary tree having *ROOT* as the pointer to the root node.

Steps:

```
1.  ptr = ROOT                                            // Start search from the root
2.  If (ptr = NULL) then
3.      Print "Tree is empty"
4.      Exit                                              // Quit the execution
5.  EndIf
6.  parent = SearchParent(ROOT, ITEM)                     // To find the parent of the desired node
7.  If(parent ≠ NULL) then                                // If the node with ITEM exists
```

(*Contd.*)

```
 8.    ptr1 = parent→LC, ptr2 = parent→RC          // Get the sibling of the parent node
 9.    If(ptr1→DATA = ITEM) then                   // Choose the left sibling with data ITEM for
                                                                                    deletion
10.       If(ptr1→LC = NULL) and (ptr1→RC = NULL) then
11.          parent→LC = NULL                                    // Left child is deleted
12.       Else
13.          Print "Node is not a leaf node: No deletion"
14.       EndIf
15.    Else                          // Choose the right sibling with data ITEM for deletion
16.       If(ptr2→LC = NULL) and (ptr2→RC = NULL) then
17.          parent→RC = NULL                                    // Right child is deleted
18.       Else
19.          Print "Node is not a leave node: No deletion"
20.       EndIf
21.    EndIf
22. Else
23.    Print "Node with data ITEM does not exist: Deletion fails"
24. EndIf
25. Stop
```

This algorithm uses a procedure *SearchParent* to search the parent of a node containing KEY as data. This procedure can be defined recursively as below:

Algorithm SearchParent

Input: ITEM is the item of search, PTR is the pointer to the node from where the search will start.

Output: 'parent' is the address of the parent node of the node containing the data ITEM.

Data structure: Linked structure of a binary tree.

Steps:

```
 1. parent = PTR
 2. If (PTR→DATA ≠ ITEM) then
 3.    ptr1 = PTR→LC, ptr2 = PTR→RC
 4.    If (ptr1 ≠ NULL) then
 5.       SearchParent(ptr1)                          // Search in the left sub-tree
 6.    Else                                           // Key is not in the left sub-tree
 7.       parent = NULL     // ITEM is not in the left sub-tree, NULL pointer is returned
 8.    EndIf
 9.    If (ptr2 ≠ NULL) then
10.       SearchParent (ptr2)                         // Search in the right sub-tree
11.    Else                                           // Key is not in the right sub-tree
12.       parent = NULL          // ITEM is not in the right sub-tree, NULL pointer is
                                                                                    returned
```

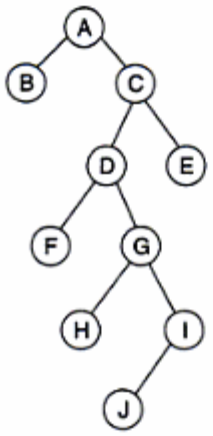
(Contd.)

```
13.     EndIf
14.  Else
15.     Return(parent)                    // Return the pointer to the parent node, if any
16.  EndIf
17.  Stop
```

Assignment 7.7

A tree can be represented using three parallel arrays. Among them, two are to store links (for left child and right child) and the third one stores the data content. Such a representation is shown in Figure 7.20.

Write a procedure to construct a tree and then perform insertion and deletion operations on it. (Assume the case of leaf node only).



	LCHILD	DATA	RCHILD
1	2	A	3
2	–	B	–
3	4	C	5
4	6	D	7
5	–	E	–
6	–	F	–
7	8	G	9
8	–	H	–
9	10	I	–
10	–	J	–

Figure 7.20 A binary tree representation with three arrays.

7.4.3 Traversals

The traversal operation is a frequently used operation on a binary tree. This operation is used to visit each node in the tree exactly once. A full traversal on a binary tree gives a linear ordering of the data in the tree. For example, if the binary tree contains an arithmetic expression then its traversal may give us the expression in infix notation, postfix notation or prefix notation.

Now a tree can be traversed in various ways. For a systematic traversal, it is better to visit each node (starting from the root) and its subtrees in the same fashion. There are six such possible ways:

1.	R	T_l	T_r	4.	T_r	T_l	R
2.	T_l	R	T_r	5.	T_r	R	T_l
3.	T_l	T_r	R	6.	R	T_r	T_l

Here, T_1 and T_r denote the left and right sub-trees of the node R, respectively. Consider a binary tree as shown in Figure 7.21.

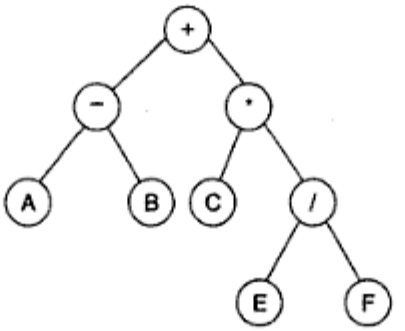


Figure 7.21 A binary tree representing an arithmetic expression.

Visit 1:

$$+\ T_{l+}T_{r+}$$

$$+\ -\ T_{l-}T_{r-}T_{r+}$$

$$+\ -\ A\ T_{lA}T_{rA}T_{r-}T_{r+}$$

$$+\ -\ A\ B\ T_{lB}T_{rB}T_{r+}$$

$$+\ -\ A\ B\ *\ T_{l*}T_{r*}$$

$$+\ -\ A\ B\ *\ C\ T_{lC}T_{rC}T_{r*}$$

$$+\ -\ A\ B\ *\ C\ /\ T_{l/}T_{r/}$$

$$+\ -\ A\ B\ *\ C\ /\ E\ T_{lE}T_{rE}T_{r/}$$

$$+\ -\ A\ B\ *\ C\ /\ E\ F\ T_{lF}T_{rF}$$

$$+\ -\ A\ B\ *\ C\ /\ E\ F$$

Likewise, one can obtain the other visits as shown below (only the result):

Visit 2:

$$A - B + C * E / F$$

Visit 3:

$$A\ B - C\ E\ F / * +$$

Visit 4:

$$F\ E / C * B\ A - +$$

Visit 5:

$$F / E * C + B - A$$

Visit 6:

$$+ * / F\ E\ C - B\ A$$

It may be noted that Visit 1 and Visit 4 are mirror symmetric; similarly, Visit 2 with Visit 5 and Visit 3 with Visit 6. So, out of six possible traversals, only three are fundamental, they are categorized as given below:

1. $R\ T_l\ T_r$ (Preorder)
2. $T_l\ R\ T_r$ (Inorder)
3. $T_l\ T_r\ R$ (Postorder)

Preorder traversal

In this traversal, the root is visited first, then the left sub-tree in preorder fashion, and then the right sub-tree in preorder fashion. Such a traversal can be defined as follows:

- Visit the root node R.
- Traverse the left sub-tree of R in preorder.
- Traverse the right sub-tree of R in preorder.

Inorder traversal

With this traversal, before visiting the root node, the left sub-tree of the root node is visited, then the root node and after the visit of the root node the right sub-tree of the root node is visited. Visiting both the sub-trees is in the same fashion as the tree itself. Such a traversal can be stated as follows:

- Traverse the left sub-tree of the root node R in inorder.
- Visit the root node R.
- Traverse the right sub-tree of the root node R in inorder.

Postorder traversal

Here, the root node is visited in the end, that is, first visit the left sub-tree, then the right sub-tree, and lastly the root. A definition for this type of tree traversal is stated below:

- Traverse the left sub-tree of the root R in postorder
- Traverse the right sub-tree of the root R in postorder
- Visit the root node R.

Observe that each algorithm contains three steps out of which two steps are defined recursively. In preorder, the root is visited first (pre), in inorder, the root is visited in between the left sub-tree and the right sub-tree (in) and in postorder, the root is visited after the left sub-tree and the right sub-tree are visited (post). Also, it may be noted that if the binary tree contains an arithmetic expression, then its inorder, preorder and postorder traversals produce infix, prefix, and postfix expressions, respectively.

Next, let us consider the implementation of the above traversals. We will define three algorithms assuming that a binary tree is represented using a linked structure. For a sequential representation of a binary tree these can be defined analogously.

Let us assume a process *Visit*(N) to imply some operation (say, display on the screen, increasing the number of node counts, etc.) while visiting the node N.

Inorder traversal of a binary tree

Recall that inorder traversal of a binary tree follows three ordered steps as follows:

- Traverse the left sub-tree of the root node R in inorder.
- Visit the root node R.
- Traverse the right sub-tree of the root node R in inorder.

Out of these steps, steps 1 and 3 are defined recursively. The following is the algorithm *Inorder* to implement the above definition.

Algorithm Inorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in inorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1. ptr = ROOT                                  // Start from ROOT
2. If (ptr ≠ NULL) then                        // If it is not an empty node
3.     Inorder(ptr→LC)                         // Traverse the left sub-tree of the node in inorder
4.     Visit(ptr)                              // Visit the node
5.     Inorder (ptr→RC)                        // Traverse the right sub-tree of the node in inorder
6. EndIf
7. Stop
```

Preorder traversal of a binary tree

The definition of preorder traversal of a binary tree, as already discussed, is presented again as follows:

- Visit the root node R.
- Traverse the left sub-tree of the root node R in preorder.
- Traverse the right sub-tree of the root node R in preorder.

The algorithm *Preorder* to implement the above definition is presented below:

Algorithm Preorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1. ptr = ROOT                                  // Start from the ROOT
2. If (ptr ≠ NULL) then                        // If it is not an empty node
3.     Visit(ptr)                              // Visit the node
4.     Preorder(ptr→LC)                        // Traverse the left sub-tree of the node in preorder
5.     Preorder(ptr→RC)                        // Traverse the right sub-tree of the node in preorder
6. EndIf
7. Stop
```

Postorder traversal of a binary tree

The definition of postorder traversal of a binary tree is repeated below:

- Traverse the left sub-tree of the root node R in postorder.
- Traverse the right sub-tree of the root node R in postorder.
- Visit the root node R.

The algorithm to implement the above is given below:

Algorithm Postorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1. ptr = ROOT                                   // Start from the root
2. If (ptr ≠ NULL) then                         // If it is not an empty node
3.     Postorder(ptr→LC)                        // Traverse the left sub-tree of the node in inorder
4.     Postorder(ptr→RC)                        // Traverse the right sub-tree of the node in inorder
5.     Visit(ptr)                               // Visit the node
6. EndIf
7. Stop
```

Assignment 7.8

Consider the following arithmetic expression:

$$(A * (B - C)) / ((D - E) * (F + G - H))$$

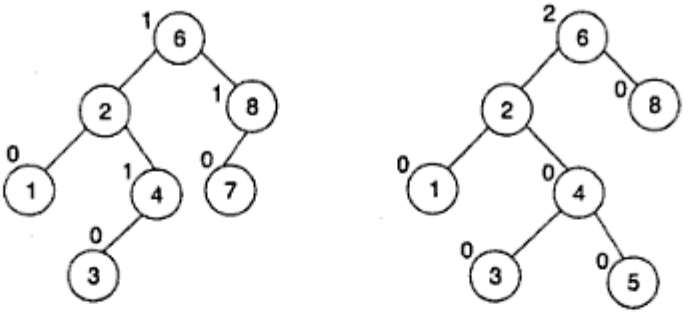
Construct the binary tree for the above expression (assume the conventional precedence of operations). Note that parentheses are explicitly mentioned and hence it is not necessary to include them in the binary tree.

For the tree so constructed, apply the three tree traversal algorithms and obtain the results of traversals.

Non-recursive implementation of traversal algorithms

The above-mentioned traversal algorithms are defined recursively and the process is straightforward in the sense that here a programmer has to convert the definitions into recursions. A language translator takes the burden to carry out the execution. Another more potentially efficient approach to these algorithms is the non-recursive implementation using a stack. This is significantly important both from the execution time point of view and the programming languages which do not have a dynamic memory management scheme.

To implement the non-recursive versions of the three traversal algorithms, we assume that a stack is maintained using an array *INFO* of suitable size and the tree is stored using three parallel arrays, as shown in Figure 7.22. Let us assume 'ptr' is a variable to locate a node in the array, that is, DATA[ptr] is the data content of a node which is stored in the array location



(a) Height balanced (b) Height unbalanced

Figure 7.53 Two binary search trees with the balance factors of each node.

It may be noted that a height balanced binary tree is always a binary search tree and a complete binary search tree is always height balanced, but the reverse may not be true.

The basic objective of the height balanced binary search tree is to perform searching, insertion and deletion operations efficiently. These operations may not be with the minimum time but the time involved is less than that of in an unbalanced binary search tree. It may be realized that the search time in the case of a complete binary search tree is minimum but insertion and deletion may not be with the minimum time always.

In the following discussion, we will see how an unbalanced binary search tree can be converted into a height balanced binary search tree. Suppose initially there is a height balanced binary tree. Whenever a new node is inserted into it (or deleted from it), it may become unbalanced. We will first see the mechanism to balance an unbalanced tree due to the insertion of a node into it.

The following steps need to be adopted.

1. *Insert node into a binary search tree:* Insert the node into its proper position following the properties of binary search tree.
2. *Compute the balance factors:* On the path starting from the root node to the node newly inserted, compute the balance factors of each node. It can be verified that a change in balance factors will occur only in this path.
3. *Decide the pivot node:* On the path as traced in Step 2, determine whether the absolute value of any node's balance factor is switched from 1 to 2. If so, the tree becomes unbalanced. The node which has its absolute value of balance factor switched from 1 to 2 marked as a special node and called the pivot node. (There may be more than one node which has its balance factor, $|bf|$ switched from 1 to 2, but the nearest node to the newly inserted node will be the pivot node.)
4. *Balance the unbalance tree:* It is necessary to manipulate pointers centred at the pivot node to bring the tree back into height balance. This pointer manipulation is well known as *AVL rotation*, the mechanism of which is illustrated next in the text.

AVL rotations

In order to balance a tree, an elegant method was devised in 1962 by two Russian mathematicians, G.M. Adelson-Velskii and E.M. Lendis, and the method is named AVL rotation in their honour.

There are four cases of rotations possible which are discussed below:

Case 1: *Unbalance occurs due to the insertion in the left sub-tree of the left child of the pivot node.*

Figure 7.54 illustrates the rotation for Case 1. In this case, the following manipulations in pointers take place:

- Right sub-tree (A_R) of left child (A) of pivot node (P) becomes the left sub-tree of P.
- P becomes the right child of A.
- Left sub-tree (A_L) of A remains the same.

This case is called LEFT-TO-LEFT insertion.

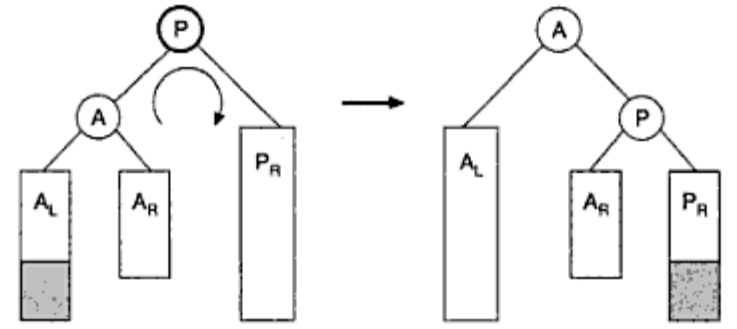


Figure 7.54 AVL rotation when unbalance occurs due to insertion in the left sub-tree of the left child of the pivot node (LEFT-TO-LEFT insertion).

Illustration. Consider the illustration shown in Figure 7.55. Node 2 is inserted into the initially height balanced tree (see Figure 7.55(a)). Here, 6 is the pivot node because this is the nearest node from the newly inserted node which has its balance factor switched from 1 to 2 (see Figure 7.55(b)). This insertion corresponds to the case of LEFT-TO-LEFT insertion. AVL rotation required is depicted in Figures 7.55(c)–(d). Final height balanced tree is shown in Figure 7.55(e) with balance factor of each node.

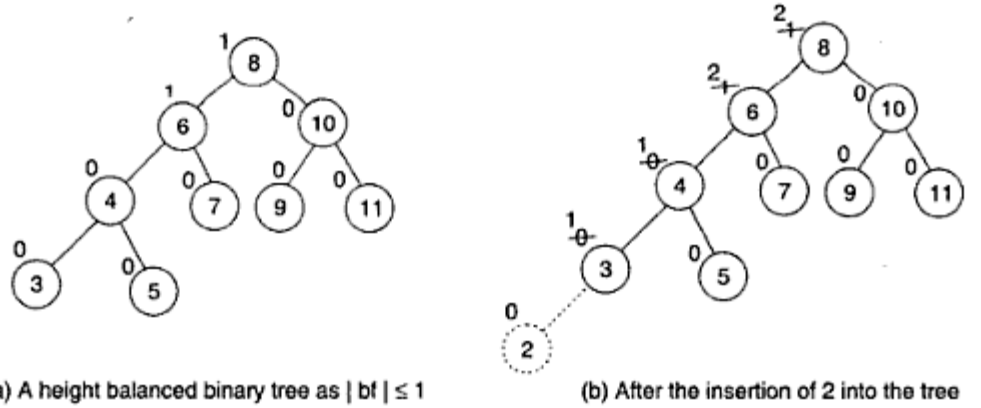


Figure 7.55 Continued.

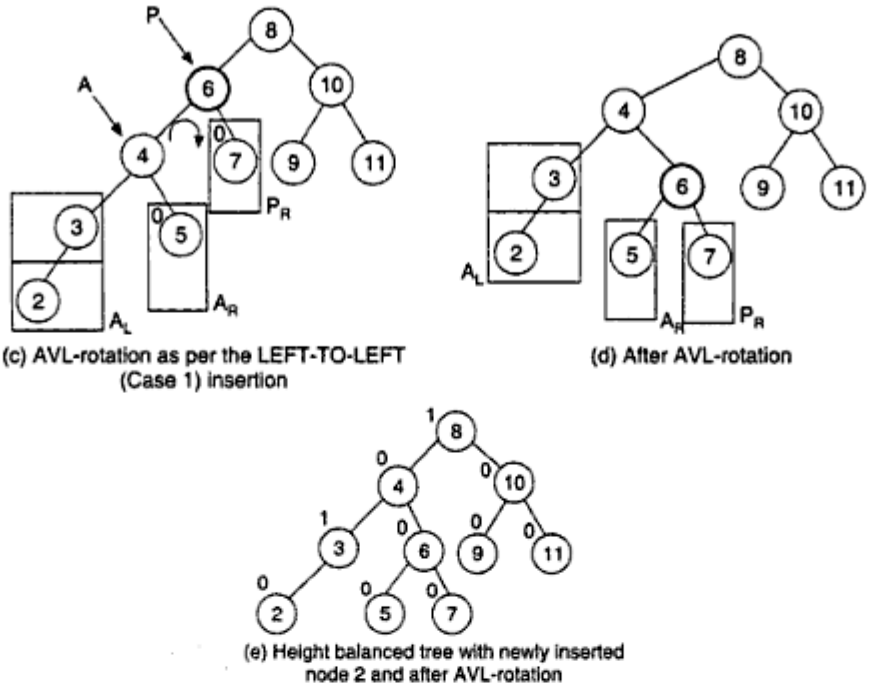


Figure 7.55(a)–(e) LEFT-TO-LEFT insertion and AVL rotation to make the tree height balanced.

Case 2: *Unbalance occurs due to insertion in the right sub-tree of the right child of the pivot node.* This case is the reverse and symmetric to Case 1. This rotation is illustrated in Figure 7.56. In this case, the following manipulations in pointers take place:

- Left sub-tree (B_L) of right child (B) of the pivot node (P) becomes the right sub-tree of P.
- P become the left child of B.
- Right sub-tree (B_R) of B remains the same.

This case is known as RIGHT-TO-RIGHT insertion.

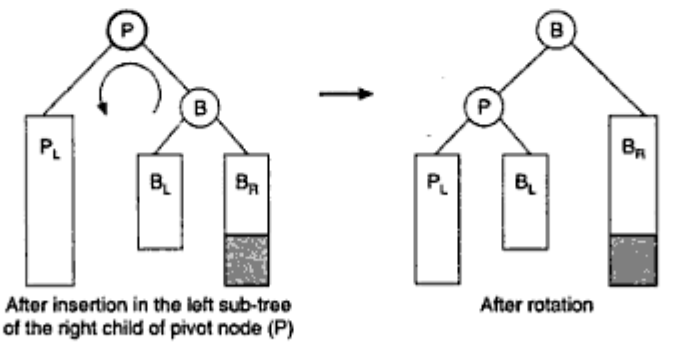


Figure 7.56 AVL rotation when unbalance occurs due to insertion in the right sub-tree of the right child of the pivot node (RIGHT-TO-RIGHT insertion).

Illustration. Case 2 of AVL rotation occurs due to RIGHT-TO-RIGHT insertion. A situation is illustrated in Figure 7.57. Due to the insertion of 12, the tree becomes unbalanced (see

SECOND EDITION

Classic Data Structures

Debasis Samanta

This book is the **second edition** of a text designed for undergraduate engineering courses in Data Structures. The treatment of the subject matter in this second edition maintains the same general philosophy as in the first edition but with significant additions. These changes are designed to improve the readability and understandability of all algorithms so that the students acquire a firm grasp of the key concepts.

The book provides a complete picture of all important data structures used in modern programming practice. It shows:

- various ways of representing a data structure
- different operations to manage a data structure
- several applications of a data structure

The algorithms are presented in English-like constructs for ease of comprehension by students, though all of them have been implemented separately in C language to test their correctness.

KEY FEATURES

- Red-black tree and spray tree are discussed in detail
- Includes a new chapter on **Sorting**
- Includes a new chapter on **Searching**
- Includes a new appendix on Analysis of Algorithms for those who may be unfamiliar with the concepts of algorithms
- Provides numerous section-wise assignments in each chapter
- Also included are exercises—Problems to Ponder—in each chapter to enhance learning

The book is suitable for students of (i) computer science, (ii) computer applications, (iii) information and communication technology (ICT), and (iv) computer science and engineering.

THE AUTHOR

DEBASIS SAMANTA, Ph.D., is Associate Professor at the School of Information Technology, Indian Institute of Technology Kharagpur. He is also the author of *Object-Oriented Programming with C++ and Java*, published by PHI Learning, New Delhi.

Dr. Samanta has contributed to numerous journal and conference publications in the areas of Information System Design, Software Testing, Human Computer Interaction, etc.

ISBN:978-81-203-3731-2

9 788120 337312

Rs. 395.00

www.phindia.com