CHAPTER 12

Abstract Data Types

(Solutions to Review Questions and Problems)

Review Questions

Q12-1. An abstract data type (ADT) is a data declaration packaged together with the operations that are meaningful for the data type. In an ADT, the operations used to access the data are known, but the implementation of the operations are hidden.

Q12-3. A queue is a linear list in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front. These restrictions ensure that the data are processed through the queue in the order in which they are received. In other words, a queue is a first in, first out (FIFO) structure. Four basic queue operations defined in this chapter are *queue*, *enqueue*, *dequeue*, and *empty*.

Q12-5. A tree consists of a finite set of elements, called nodes (or vertices), and a finite set of directed lines, called arcs, that connect pairs of the nodes. If the tree is not empty, one of the nodes, called the root, has no incoming arcs. The other nodes in a tree can be reached from the root following a unique path, which is a sequence of consecutive arcs. A binary tree is a tree in which no node can have more than two subtrees. A binary search tree (BST) is a binary tree with one extra property: the key value of each node is greater than the key values of all nodes in each left subtree and smaller than the value of all nodes in each right subtree.

Q12-7. A graph is an ADT made of a set of nodes, called vertices, and set of lines connecting the vertices, called edges or arcs. Graphs may be either directed or undirected. In a directed graph, or digraph, each edge, which connects two vertices, has a direction (arrowhead) from one vertex to the other. In an undirected graph, there is no direction.

Q12-9. General linear lists are used in situations where the elements are accessed randomly or sequentially. For example, in a college, a linear list can be used to store information about the students who are enrolled in each semester.

Problems

P12-1. Table 12.1 shows the code.

Table 12.1 *Solution to P12-1*

```
Algorithm: Emptying the contents of a stack
Purpose: It removes the items in a stack one by one
Pre: Stack S2
Post: Stack S2 is empty
Return: None
while (NOT empty (S2))
{
    pop (S2, x)          // x will be discarded
}
```

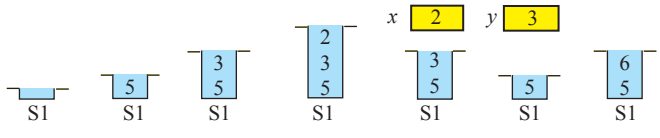
P12-3. Table 12.2 shows the code. Note that we need a temporary stack to make the order of items in S1 and S2 the same.

Table 12.2 *Solution to P12-3*

```
Algorithm: Copy the contents of a stack to another one
Purpose: It copies the elements in Stack S1 to S2
Pre: Stack S1 and S2 (empty)
Post: Both S1 and S2 holds the same items in the same order
Return: None
stack (Temp)         // Create a temporary empty stack
while (NOT empty (S1))
{
     pop (S1, x)
     push (Temp, x)
}
while (NOT empty (Temp))
{
  pop (Temp, x)
  push (S1, x)
  push (S2, x)
}
```

P12-5. Figure 12.1 shows the answer.

Figure 12.1 *Solution to P12-5*



P12-7. Table 12.3 shows the algorithm. Note that we first copy the contents of stack S1 and S2 into two temporary stack to be able to restore their contents after comparison.

Table 12.3 *Solution to P12-7*

```
Algorithm: CompareStack (S1, S2)
Purpose: Check if two stacks are the same
Pre: Given: S1 and S2
Post: S1 and S2
Return: true (S1 = S2) or false (S1 ≠ S2)
{
    flag ← true
    Stack (Temp1)
    Stack (Temp2)
    while (NOT empty (S1) and NOT empty (S2))
    {
        pop (S1, x)
        push (Temp1, x)
        pop (S2, y)
        push (Temp2, y)
        if (x ≠ y)
         flag ← false
    }
    if (NOT empty (S1) or NOT empty (S2))       flag ← false
    while (NOT empty (Temp1) and NOT empty (Temp2))
    {
        pop (Temp1, x)
        push (S1, x)
        pop (Temp2, y)
        push (S2, y)
    }
    return flag
}
```

P12-9. Table 12.4 shows the pseudocode.

Table 12.4 *Solution to P12-9*

```
while (NOT empty (Q1))
{
    dequeue (Q1, x)
    enqueue (Q2, x)
}
```

P12-11. Table 12.5 shows the pseudocode.

Table 12.5 *Solution to P12-11*

```
while (NOT empty (Q2))
{
    dequeue (Q2, x)
    enqueue (Q1, x)
}
```

P12-13.

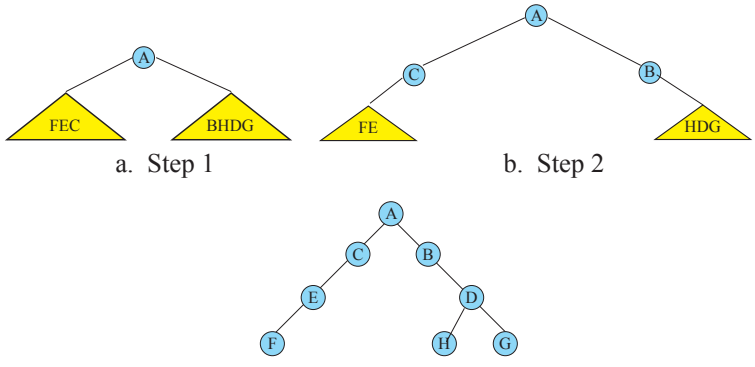
a. Since traversal is postorder, the root comes at the end: G

b. Since the traversal is preorder, the root comes at the beginning: I

c. Since traversal is postorder, the root comes at the end: E

P12-15. See Figure 12.2. The postorder traversal FECHGDBA tells us that node A is the root. The Inorder traversal FECABHDG implies that nodes FEC in the left of A are in the left subtree and nodes BHDG in the right of A are in the right subtree. Following the same logic for each subtree we build the binary tree.

Figure 12.2 *Solution to Problem 12-15*



P12-17. Table 12.6 shows the pseudocode.

Table 12.6 *Solution to P12-17*

```
Algorithm: Palindrome (String[1 ¼ n])
Purpose: It checks if a string is a palindrome
Pre: Given a string
Post:
Return: true ( if string is palindrome) or false (if not palindrome)

{
    Allocation: An array of size n is allocated
```

Table 12.6 *Solution to P12-17*

```
}

stack (Stack S)    // Stack operation (Creating an empty stack)
{
    allocate record S of two fields
    S.top ← 0
    S.count ← 0
}

push (Stack S, DataRecord x)    // Push operation
{
    S.top ← S.top +1
    S.count ← S.count + 1
    A [S.top] ← x
}

pop (Stack S, DataRecord x)    // Pop operation
{
    x ← A [S.top]
    S.top ← S.top – 1
    S.count ← S.count – 1
}

empty (Stack S)    // Empty operation (Checking if stack is empty)
{
    if (S.count = 0)    return true
    else                return false
}
```

P12-19. Table 12.7 shows the pseudocode.

Table 12.7 *Solution to P12-19*

```
Algorithm: Queue operations
Purpose: It defines four operations for a queue implemented as an array
Pre: Given an array
Post:
Return:

{
    Allocation: An array of size n is allocated
}

queue (Queue Q)    // Queue operation
{
    allocate record Q of three fields
```

Table 12.7 *Solution to P12-19*

```
    Q.count ← 0
    Q.rear ← 0
    Q.front ← 0
}

enqueue (Queue Q, DataRecord x)      // Enqueue operation
{
    if (Q.front = 0)     Q.front ← 1
    Q.count ← Q.count +1
    Q.rear ← Q.rear + 1
    A[Q.rear] ← x
}

dequeue (Queue Q, DataRecord x)    // Dequeue operation
{
    x ← A[Q.front]
    Q.front ← Q.front + 1
    Q.count ← Q.count - 1
}

empty (Queue Q)      // Empty operation
{
    if (Q.count = 0)   return true
    else               return false

}
```

P12-21. Table 12.8 shows the pseudocode.

Table 12.8 *Solution to P12-21*

```
Algorithm: General list operations
Purpose: It defines four operations for a general list using an array
Pre: Given an array
Post:
Return:

{
    Allocation: An array of size n is allocated
    Include BinarySearchArray algorithm from chapter 11
    Include ShiftDown algorithm from chapter 11
}

list (List L)      // List operation
{
    allocate record L of two fields
```

Table 12.8 *Solution to P12-21*

```
    L.count ← 0
    L.first ← 0
}

insert (List L, DataRecord x)    // Insert operation
{
    BinarySearchArray (A, n, x.key, flag, i)
    ShiftDown (A, n, i)
    A[i] ← x
    L.first ← 1
    L.count ← L.count + 1
}

delete (List L, DataRecord x)    // Delete operation
{
    BinarySearchArray (A, n, x.key, flag, i)
    x ← A[i]
    ShiftUp (A, n, i)
    L.count ← L.count - 1
    if (empty (L))
    L.first ← 0
}

retrieve (List L, DataRecord x)    // Retrieve operation
{
    BinarySearchArray (A, n, x.key, flag, i)
    x ← A[i]
}

traverse (List L, Process)    // Traverse operation
{
    walker ← 1
    while (walker £ L.count)
    {
        Process (A[walker]
        walker ← walker + 1
    }
}

empty (L)    // Empty operation
{
    if (L.count = 0)    return true
    else    return false
}
```