CHAPTER 12

Abstract Data Types

(Solutions to Odd Numbered Problems)

Review Questions

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.

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*.

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.

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.

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.

Multiple-Choice Questions

11. b 13. b 15. d 17. a 19. c 21. b
23. a 25. b

Exercises

27.

```
stack(Temp)
while (NOT empty (S1))
{
        pop (S1, x)
        push (Temp, x)                    // Temp is a temporary stack
}
while (NOT empty (Temp))
{
        pop (Temp, x)
        push (S2, x)
}
```

29.

```
stack(Temp)
while (NOT empty (S2))
{
        pop (S2, x)
        push (Temp, x)                    // Temp is a temporary stack
}
while (NOT empty (Temp))
{
        pop (Temp, x)
        push (S2, x)
}
```

31. Algorithm S12.31 shows the pseudocode.

Algorithm S12.31 *Exercise* 31

```
Algorithm: Palindrome(String[1 … n])
Purpose: It checks if a string is a palindrome
Pre: Given: a string
Post:
Return: true (the string is a palindrome) or false (the string is not a palindrome)
{
    stack (S)
    i ← 1
    while i ≤ n
    {
        C ← string[i]
        push (S, C)
        i ← i + 1
    }
    i ← 1
    while i ≤ n
    {
        pop (S, x)
        if (x ≠ string[i])        return false
    }
    return true
}
```

33.

```
while (NOT empty (Q))
{
    dequeue (Q, x)                // x will be discarded
}
```

35.

```
while (NOT empty (Q2))            // First we empty Q2.
{
    dequeue (Q2, x)
}
while (NOT empty (Q1))
{
    dequeue (Q1, x)
    enqueue (Temp, x)
}
while (NOT empty (Temp))
{
    dequeue (Temp, x)
    enqueue (Q1, x)
    enqueue (Q2, x)
}
```

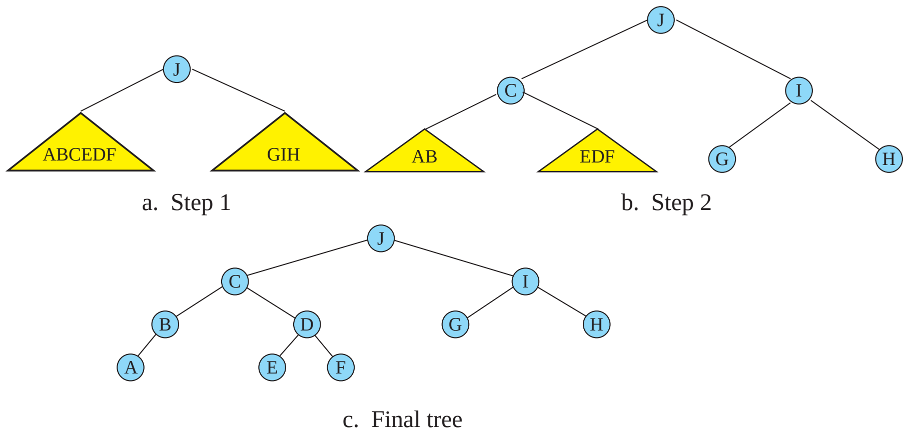
37. Algorithm S12.37 shows the pseudocode.

Algorithm S12.37 *Exercise 37*

```
Algorithm: CompareQueue(Q1, Q2)
Purpose: Check if two queues are the same
Pre: Given: Q1 and Q2
Post:
Return: true (Q1 = S2) or false (Q1 ≠ S2)
{
    flag ← true
    Queue(Temp1)
    Queue(Temp2)
    while (NOT empty (Q1) OR NOT empty (Q2))
    {
        if (NOT empty (Q1))
        {
            dequeue (Q1, x)
            enqueue (Temp1, x)
        }
        if (NOT empty (Q2))
        {
            dequeue (Q2, y)
            enqueue (Temp2, y)
        }
        if (x ≠ y)                         flag ← false
        if (NOT empty (Q1) XOR NOT empty (Q2))        flag ← false
    }
    while (NOT empty (Temp1) OR NOT empty (Temp2))
    {
        if (NOT empty (Temp1))
        {
            dequeue (Temp1, x)
            enqueue (Q1, x)
        }
        if (NOT empty (Temp2))
        {
            dequeue (Temp2, y)
            enqueue (Q2, y)
        }
    return flag
}
```

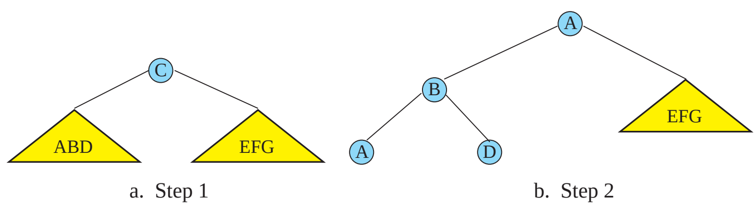
39. The preorder traversal JCBADEFIGH tells us that node J is the root. The inorder traversal ABCEDFJGIH implies that nodes ABCEDF (in the left of J) are in the left subtree and nodes GIH (in the right of J) are in the right subtree. Following the same logic for each subtree we build the binary tree as shown in Figure S11.39.

Figure S11.39 *Exercise 39*



41. The postorder traversal GFDABEC tells us that node C is the root. The inorder traversal ABDCEFG tell us that nodes ABD (in the left of C) are in the left subtree and nodes EFG (in the right of A) are in the right subtree (Figure S11.41). We can decompose the left subtree into two nodes, but the right subtree cannot be decomposed because nodes EFG are not contiguous in the postorder traversal. We cannot find the root of this subtree. There are some errors in the postorder traversal listing.

Figure S11.41



43. Algorithm S12.43 shows the pseudocode.

Algorithm S12.43 *Exercise 43*

```
Algorithm: StackADTLinkedListImplementation
Purpose: Implementing stack operations with linked list

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

push (Stack S, DataRecord x)                      // Push operation
{
        Allocate a node and a new pointer
        new ← address of the allocated node
        (*new).data ← x
        (*new).link ← null
        if (S.top = null)          S.top ← new
        else
        {
                (*new).link ← S.top
                S.top ← new
        }
        S.count ← S.count + 1
}

pop (Stack S, DataRecord x)                       // Pop operation
{
        x ← *(S.top).data
        S.top ← *(S.top).link
        S.count ← S.count –1
}

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

45. Algorithm S12.45 shows the pseudocode.

Algorithm S12.45 *Exercise 45*

```
Algorithm: QueueADTLinkedListImplementation
Purpose: Implementing queue operations with linked list
queue (Queue Q)                              // Queue operation
{
        allocate record Q of three fields
        Q.count ← 0
        Q.front ← null
        Q.rear ← null
}
enqueue (Queue Q, DataRecord x)              // enqueue operation
{
        Allocate a node and a new pointer
        new ← address of the allocated node
        (*new).data ← x
        (*new).link ← null
        if (Q.count = 0)
                Q.front ← new
                Q.rear ← new
        else
        {
                if (Q.count = 1)
                {
                        (*front).link ← new
                        rear ← (*front).link
                }
                else            (*rear).link ← new
        }
        Q.count ← Q.count + 1
}
dequeue (Queue Q, DataRecord x)              // Dequeue operation
{
        x ← A[Q.front]
        if (Q.count = 1)
                Q.front ← null
                Q.rear ← null
        else
        {
                if (Q.count = 1)
                {
                        (*front).link ← new
                        rear ← (*front).link
                }
                else            front ← (*front).link
        }
        Q.count ← Q.count – 1
empty (Queue Q)                              // Empty operation
{
        if (Q.count = 0)            return true
        else                        return false
}
```

47. Algorithm S12.47 shows the pseudocode.

Algorithm S12.47 *Exercise 47*

```
Algorithm: ListADTLinkedListImplementation
Purpose: Implementing list operations with a linked list
{
        Include SearchLinkedList algorithm from chapter 11
}

list (List L)                                   // List operation
{
        allocate record L of two fields
        L.count ← 0
        L.first ← null
}

insert (List L, DataRecord x)                   // Insert operation
{
        Allocate a node and a new pointer
        new ← address of the allocated node
        (*new).data ← x
        (*new).link ← null
        if (L.count = 0)                                        // List is empty
        {
                L.first ← new
                L.count ← L.count + 1
        else
        {
                SearchLinkedList(L, x, pre, cur, flag)
                if (flag = true)        return L                // No duplicate
                if (pre = null)                                 // Insertion at the beginning
                {
                        cur ← (*new).link
                        L.first ← new
                        L.count ← L.count + 1
                        return L
                }
                if (cur = null)                                 // Insertion at the end
                {
                        (*pre).link ← new
                        (*new).link ← null
                        L.count ← L.count + 1
                        return L
                }
                (*new).link ← cur                               // Insertion in the middle
                (*pre).link ← null
                return L
                L.count ← L.count + 1
        }
}
```

Algorithm S12.47 *Exercise 47*

```
delete (List L, DataRecord x)                       // Delete operation
{
        SearchLinkedList(L, x, pre, cur, flag)
        if (flag = false)          return L             // Target not found
        if (pre = null)                                 // Delete the first node
        {
                L.first ← (*cur).link
                L.count ← L.count − 1
                return L
        }
        (*pre).link ← (*cur).link                       // Delete other nodes
        L.count ← L.count − 1
}

retrieve (List L, DataRecord x)                     // Retrieve operation
{
        SearchLinkedList(L, x, pre, cur, flag)
        if (flag = false)          return error         // Target not found
        return (*cur).data
}

traverse (List L, Process)                          // Traverse operation
{
        walker ← 1
        while (walker ≠ null)
        {
                Process (*walker).data
                walker ← (*walker).link
        }
}

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