CHAPTER 11

Data Structures

(Solutions to Review Questions and Problems)

Review Questions

Q11-1. Arrays, records, and linked lists are three types of data structures discussed in this chapter.

Q11-3. Elements of an array are contiguous in memory and can be accessed by use of an index. Elements of a linked list are stored in nodes that may be scattered throughout memory and can only be accessed via the access functions for the list (i.e., the address of a specific node returned by a search function).

Q11-5. An array is stored contiguously in memory. Most computers use row-major storage to store a two-dimension array.

Q11-7. The fields of a node in a linked list are the data and a pointer (address of) the next node.

Q11-9. We use the head pointer to point to the first node in the linked list.

Problems

P11-1. Table 11.1 shows the algorithm in pseudocode.

Table 11.1 *Solution to P11.1*

```
Algorithm: CompareArrays (A, B)
Purpose: Compare each element in array A with the one in array B
Pre: Arrays A and B of 10 integers
Post: None
Return: true or false
{
   i ← 1
   while (i < 10)
   {
      if A[i] != B[i]          return false
      i ← i + 1
   }
   return true                 // A is equal to B
}
```

P11-3. Table 11.2 shows the routine in pseudocode that prints an array.

Table 11.2 *Solution to P11-3*

```
Algorithm: PrintArray (A, r, c)
Purpose: Print the contents of 2-D array
Pre: Given Array A, number of rows (r) and number of columns (c)
Post: Print the values of the elements of A
Return: None
{
    i ← 1
    while (i < r)
    {
        j ← 1
         while (j < c)
        {
            print A[i][j]
          j ← j + 1
         }
        i ← i + 1
    }
}
```

P11-5. Table 11.3 shows the binary search routine in pseudocode (see Chapter 8).

Table 11.3 *Solution to P11-5*

```
Algorithm: BinarySearchArray (A, n, x)
Purpose: Apply a binary search on an array A of n elements
Pre: A, n, x     // x is the target we are searching for
Post: None
Return: flag, i
{
    flag ← false
    first ← 1
    last ← n
    while (first ≤ last)
    {
        mid = (first + last) / 2
        if (x < A[mid])     last ← mid − 1
        if (x > A[mid])     first ← mid + 1
        if (x = A[mid])     first ← Last + 1
    }
    if (x > A[mid])   i ← mid + 1
    if (x < A[mid])   i ← mid
    if (x = A[mid])   flag ← true
    return (flag, i)
}
```

P11-7. Table 11.4 shows the shiftDown algorithm.

Table 11.4 *Solution to P11-7*

```
Algorithm: ShiftDown (A, n, i)
Purpose: Shift all elements starting from element with index i one place
Pre: A, n, i
Post: None
Return:
{
    j ← n
    while (j > i − 1)
    {
        A[j + 1] ← A[j]
        j ← j − 1
    }
}
```

P11-9. Table 11.5 shows the routine in pseudocode that multiplies each element of an array by a constant.

Table 11.5 *Solution to P11-9*

```
Algorithm: MultiplyConstant (A, n, C)
Purpose: Multiply all elements of an array by a constant
Pre: A, n, C      // C is the constant
Post: None
Return:
{
    i ← 1
    while (i ≤ n)
    {
        A[i] ← C × A[i]
        i ← i + 1
    }
}
```

P11-11. Table 11.6 shows the routine in pseudocode that subtract two fractions. To subtract, we change the sign of Fr1 and then add it to Fr2 by calling AddFraction algorithm.

Table 11.6 *Solution to P11-11*

```
Algorithm: SubtractFraction (Fr1, Fr2)
Purpose: Subtract two fractions (Fr2 − Fr1)
Pre: Fr1, Fr2      // Assume denominators have nonzero values
Post: None
Return: Fr3
```

Table 11.6 *Solution to P11-11*

```
{
    Fr1.num ← – Fr1.num    // Change the sign of the first fraction
    Fr3 ← AddFraction (Fr1, Fr2)    // Call AddFraction
    return (Fr3)
}
```

P11-13. Table 11.7 shows the routine in pseudocode that divide two fractions. First, we inverse the second fraction and then multiply them by calling Multiply-Fraction algorithm.

Table 11.7 *Solution to P11-13*

```
Algorithm: DivideFraction (Fr1, Fr2)
Purpose: Divide two fractions (Fr1 ÷ Fr2)
Pre: Fr1, Fr2    // Assume nonzero values
Post: None
Return: Fr3
{
    Temp ← Fr2.denom
    Fr2.denom ← Fr2.num
    Fr2.num ← Temp
    Fr3 ←  MultiplyFraction (Fr1, Fr2)    // Call MultiplyFraction
    return (Fr3)
}
```

P11-15. **SearchLinkedList** algorithm returns **pre** = *null*, **cur** pointing to the only node, and **flag** = *true*. Since **pre** = *null,* the action is *list* ← (***cur**).**link,** which is equivalent to *list* ← *null.* The result is an empty list as shown in Figure 11.1. Since *list* = *null*, the **SearchLinkedList** algorithm performs *new* ← *list.* This creates a list with a single node.

Figure 11.1 *Solution to P11-15*

P11-17. Table 11.8 shows the routine for finding the average of a linked list.

Table 11.8 *Solution to Problem 11-17*

```
Algorithm: LinkedListAverage (list)
Purpose: Evaluate average of numbers in a linked list
```

Table 11.8 *Solution to Problem 11-17*

```
Pre: list
Post: None
Return: Average value
{
    counter ← 1
    sum ← 0
    walker ← list
    while (walker ≠ null)
    {
        sum ← sum + (*walker).data
        walker ← (*walker).link
        counter ← counter + 1
    }
    average ← sum / counter
    return average
}
```

P11-19. Figure 11.2 shows that if **pre** is not null, the statements **cur ← (*cur).link** and **pre ← (*pre).link** move the two pointers together to the right. In this case the two statements are equivalent to the ones we discussed in the text. However, the statement **pre ← (*pre).link** does not work when **pre** is null because, in this case, **(*pre).link** does not exist (Figure 11.3). For this reason, we should avoid using this method.

Figure 11.2 *Part of Solution to P11-19*

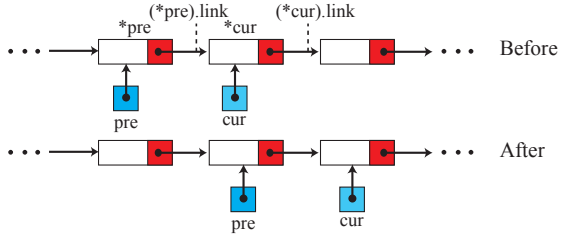

Figure 11.3 *Part of Solution to P11-19*