CHAPTER 11

Data Structures

(Solutions to Odd-Numbered Problems)

Review Questions

1. Arrays, records, and linked lists are three types of data structures discussed in this chapter.
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).
5. An array is stored contiguously in memory. Most computers use row-major storage to store a two-dimension array.
7. The fields of a node in a linked list are the data and a pointer (address of) the next node.
9. We use the head pointer to point to the first node in the linked list

Multiple-Choice Questions

11. d	13. c	15. c	17. a	19. d

Exercises

21. Algorithm S11.21 shows a routine in pseudocode that compares two arrays.

Algorithm S11.21 *Exercise 21*

Algorithm: **CompareArrays(A, B)**
Purpose: Test if every element in array **A** equals to its corresponding element in array **B**
Pre: Arrays **A** and **B** of 10 integers
Post: None
Return: *true* or *false*
{
    $i \leftarrow 1$
    while ($i \leq 10$)

Algorithm S11.21 *Exercise 21*

```
    {
        if A[i]≠B[i]   return false          // A is not equal to B
        i ← i + 1
    }
    return true                              // A is equal to B
}
```

23. Algorithm S11.23 shows a routine in pseudocode that prints an array.

Algorithm S11.23 *Exercise 23*

```
Algorithm: PrintArray (A, r, c)
Purpose: Print the contents of 2-D array
Pre: Given Array A, and values of r (number of rows) and c (number of columns)
Post: Print the values of the elements of A
Return:
{
    i ← 1
    while (i ≤ r)
    {
        j ← 1
        while (j ≤ c)
        {
            print A[i][j]
            j ← j + 1
        }
        i ← i + 1
    }
}
```

25. Algorithm S11.25 shows a binary search routine in pseudocode (see Chapter 8). Note that we use the binary search on sorted array.

Algorithm S11.25 *Exercise 25*

```
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)
```

Algorithm S11.25 *Exercise 25*

```
        {
                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    // x is found
        }
        if (x > A[mid])                  i = mid + 1
        if (x ≤ A[mid])                  i = mid
        if (x = A[mid])                  flag ← true
        return (flag, i)
}
// If flag is true, it means x is found and i is its location.
// If flag is false, it means x is not found; i is the location where the target supposed to be.
```

27. Algorithm S11.27a shows a delete routine in pseudocode. Note that this algorithm calls BinarySearch algorithm (Algorithm S11.25) and ShiftUp algorithm (Algorithm S11.27b).

Algorithm S11.27a *Exercise 27*

```
Algorithm: DeleteSortedArray(A, n, x)
Purpose: Delete an element from a sorted array
Pre: A, n, x                                         // x is the value we want to delete
Post: None
Return:
{
        {flag, i} ← BinarySearch (A, n, x)               // Call binary search algorithm
        if (flag = false)                                // x is not in A
        {
                print (x is not in the array)
                return
        }
        ShiftUp (A, n, i)                                // call shift up algorithm
        return                                           // call shift up algorithm
}
```

Algorithm S11.27b *Exercise 27*

```
Algorithm: ShiftUp (A, n, i)
Purpose: Shift up all elements one place from the last element up to element with index i.
Pre: A, n, i
Post: None
Return:
```

Algorithm S11.27b *Exercise 27*

```
{
    j ← i
    while (j ≤ n + 1)
    {
        A[j] ← A[j + 1]
        j ← j + 1
    }

}
```

29. Algorithm S11.29 shows a routine in pseudocode that adds two fractions.

Algorithm S11.29 *Exercise 29*

```
Algorithm: AddFraction(Fr1, Fr2)
Purpose: Add two fractions
Pre: Fr1, Fr2                          // Assume denominators have nonzero values
Post: None
Return: The resulting fraction (Fr3)
{
    x ← gcd (Fr1.denom, Fr2.denom)              // Call gcd (see Exercise 8.57)
    y ← (Fr1.denom × Fr2.denom) / x             // y is the least common denominator
    Fr3.num ← (y / Fr1.denom) × Fr1.num + (y / Fr2.denom) × Fr2.num
    Fr3.denom ← y
    z ← gcd (Fr3.num, Fr3.denom)                // Simplifying the fraction
    Fr3.num ← Fr3.num / z
    Fr3.denom ← Fr3.denom / z
    return (Fr3)
}
```

31. Algorithm S11.31 shows a routine in pseudocode that multiplies two fractions.

Algorithm S11.31 *Exercise 31*

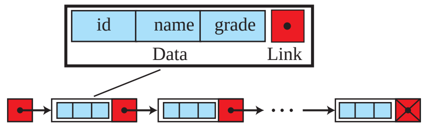
```
Algorithm: MultiplyFraction(Fr1, Fr2)
Purpose: Multiply two fractions
Pre: Fr1, Fr2                     // Assume denominators with nonzero values
Post: None
Return: Fr3
{
    Fr3.num ← Fr1.num × Fr2.num
    Fr3.denom ← Fr1.denom × Fr2.denom

    z ← gcd (Fr3.num, Fr3.denom)                // Simplifying the fraction
```

Algorithm S11.31 *Exercise 31*

```
    Fr3.num ← Fr3.num / z
    Fr3.denom ← Fr3.denom / z
    return (Fr3)
}
```

33. Figure S11.33 shows a linked list of records.

Figure S11.33 *Exercise 33*



35. Since *list* = *null*, the **SearchLinkedList** algorithm performs *new* ← *list*. This creates a list with a single node.

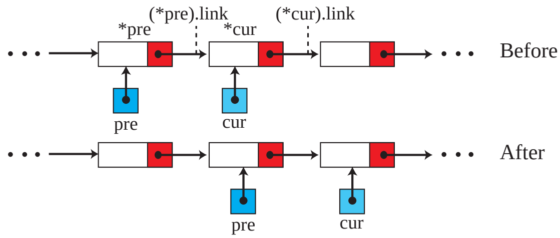
37. Algorithm S11.37 shows a routine for finding the average data in a linked list.

Algorithm S11.37 *Exercise 37*

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

39. Figure S11.39a shows that if **pre** is not null, the two 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

Figure S11.39a *Exercise 39*



because, in this case, **(*pre).link** does not exist (Figure S11.39b). For this reason, we should avoid using this method.

Figure S11.39b *Exercise 39*