CHAPTER 11

Data Structures

(Solutions to Practice Set)

Review Questions

1. Arrays, records, and linked lists are three types of data structures discussed in this chapter.
2. All elements of an array in most languages are of the same type. Elements of a record can be of the same or different types but all elements must be related to each other.
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).
4. Using indices in a pair of brackets, rather than subscripts, allows us to use variable for these indices.
5. An array is stored contiguously in memory. Most computers use row-major storage to store a two-dimension array.
6. A field is the smallest element of named data that has meaning.
7. The fields of a node in a linked list are the data and a pointer (address of) the next node.
8. The pointer identifies the next element in the linked list.
9. We use the head pointer to point to the first node in the linked list.
10. The pointer in the last node of a linked list is a *null pointer*, which means it points to nowhere.

Multiple-Choice Questions

11. d	12. b	13. c	14. b	15. c	16. d
17. a	18. a	19. d	20. b		

Exercises

21. Algorithm S11.21 shows the 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 ← 1
    while (i ≤ 10)
    {
        if A[i]≠B[i]   return false          // A is not equal to B
        i ← i + 1
    }
    return true                              // A is equal to B
}
```

22. Algorithm S11.22 shows the routine in pseudocode that reverses an array.

Algorithm S11.22 *Exercise 22*

```
Algorithm: ReverseArray(A, n)
Purpose: Reverse the elements of an array
Pre: Arrays A with n elements
Post: None
Return:
{
    i ← 1
    j ← n
    while (i < j)
    {
        Temp ← A[j]
        A[j] ← A[i]
        A[i] ← Temp
        i ← i + 1
        j ← j – 1
    }
}
```

23. Algorithm S11.23 shows the 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
        }
}
```

24. Algorithm 11.24 shows the sequential search routine in pseudocode (see Chapter 8). Note that we perform sequential search on unsorted arrays. In the beginning, we set the flag to *false*. If the search finds the target in the array, it sets the flag to *true*, moves out of the loop. It returns ***i*** (location of the target) and the value of the flag (*true*). After checking the entire list, if the search does not find the value, it moves out of the loop and returns value of the ***i*** as $n + 1$ and the value of the flag (*false*).

Algorithm S11.24 *Exercise 24*

```
Algorithm: SequentialSearchArray (A, n, x)
Purpose: Apply a sequential 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
        i ← 1
        while ((i ≤ n) or (flag = false))
        {
                if (A[i] = x)          flag ← true
                i ← i + 1
        }
        return (flag, i)                        // If x is not found, flag is false
}
```

25. Algorithm S11.25 shows the binary search routine in pseudocode (see Chapter 8). Note that we perform 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)
        {
                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.
```

26. Algorithm 11.26 shows the insert routine in pseudocode. Note that this algorithm first calls BinarySearch algorithm (Algorithm S11.25) and then ShiftDown algorithm (Algorithm S11.26b).

Algorithm S11.26a *Exercise 26*

```
Algorithm: InsertSortedArray(A, n, x)
Purpose: Insert an element in a sorted array
Pre: A, n, x                                   // x is the value we want to insert
Post: None
Return: A
{
        {flag, i} ← BinarySearch (A, n, x)     // Call binary search algorithm
        if (flag = true)                       // x is already in A
        {
                print (x is already in the array)
                return
        }
        ShiftDown (n, A, i)                    // Call shift down algorithm
        A[i] ← x
        return
}
```

Algorithm S11.26b *Exercise 26*

```
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
        }
}
```

27. Algorithm S11.27a shows the 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: A
{
        j ← i
```

Algorithm S11.27b *Exercise 27*

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

28. Algorithm S11.28 shows the routine in pseudocode that multiplies each element of an array by a constant.

Algorithm S11.28 *Exercise 28*

```
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
        }
}
```

29. Algorithm S11.29 shows the 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)
}
```

30. Algorithm S11.30 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 (Algorithm S11.29).

Algorithm S11.30 *Exercise 30*

```
Algorithm: SubtractFraction(Fr1, Fr2)
Purpose: Subtract two fractions (Fr2 − Fr1)
Pre: Fr1, Fr2                          // Assume denominators have nonzero values
Post: None
Return: Fr3
{
        Fr1.num ← − Fr1.num            // Change the sign of the first fraction
        Fr3 ← AddFraction (Fr1, Fr2)   // Call AddFraction
        return (Fr3)
}
```

31. Algorithm S11.31 shows the 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
        Fr3.num ← Fr3.num / z
        Fr3.denom ← Fr3.denom / z
        return (Fr3)
}
```

32. Algorithm S11.32 shows the routine in pseudocode that divide two fractions. First, we inverse the second fraction and then multiply them by calling MultiplyFraction algorithm (Algorithm S11.31)

Algorithm S11.32 *Exercise 32*

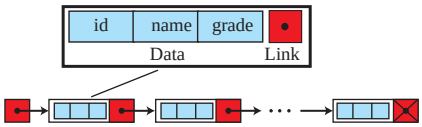
```
Algorithm: DivideFraction(Fr1, Fr2)
Purpose: Divide two fractions (Fr1 ÷ Fr2)
Pre: Fr1, Fr2                          // Assume nonzero values
Post: None
Return: Fr3
```

Algorithm S11.32 *Exercise 32*

```
{
        Temp ← Fr2.denom
        Fr2.denom ← Fr2.num
        Fr2.num ← Temp
        Fr3 ← MultiplyFraction (Fr1, Fr2)          // Call MultiplyFraction
        return (Fr3)
}
```

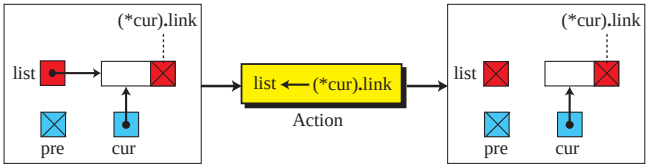
33. Figure S11.33 shows the linked list of records.

Figure S11.33 *Exercise 33*



34. **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 S11.34.

Figure S11.34 *Exercise 34*



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

36. Algorithm S11.36 shows the routine in pseudocode for building a linked. The routing uses the InsertLinkedList algorithm.

Algorithm S11.36 *Exercise 36*

```
Algorithm: BuildLinkedList(data records)
Purpose: Build a linked list from scratch
Pre: given list of data records
Post: None
Return: list, which is a pointer pointing to the first node
{
        list ← null
```

Algorithm S11.36 *Exercise 36*

```
        while (more data records)
        {
                InsertLinkedList (list, next record.key, next record)
        }
        return (list)
}
```

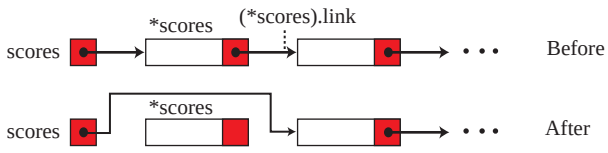
37. Algorithm S11.37 shows the routine for finding the average of a linked list.

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
}
```

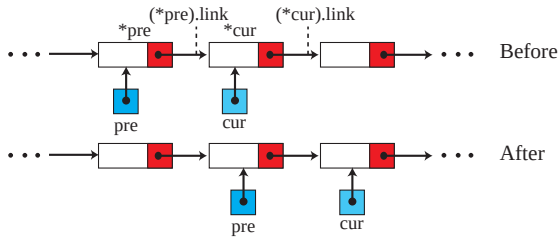
38. Figure S11.38 shows the list of scores before and after **scores ← (*scores).link** statement. The statement drops the first node. This exercise shows that we should never move the head pointer. If we do so, we lose a part of our list.

Figure S11.38 *Exercise 38*



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.

Figure S11.39a *Exercise 39*



However, the statement **pre** ← **(*pre).link** does not work when **pre** is null 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*