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. Algorithm P11-1 shows the routine in pseudocode that compares two arrays.

Algorithm P11-1 *Comparing two arrays*

```
Algorithm: Compares array A with array B
Purpose: Test if the corresponding elements in two arrays are equal
Pre: Arrays A and B of 10 integers
Post: None
Return: true or false
{
    i ← 1
    while (i ≤ 10)
    {
        if A[i]≠B[i]                    // A is not equal to B
            return  false
        i ← i + 1
    }
    return true                         // A is equal to B
}
```

P11-3. Algorithm P11-3 shows the routine in pseudocode that prints the elements of a two dimensional array.

Algorithm P11-3 *Printing elements of a two-dimensional array*

```
Algorithm: PrintArray (A, r, c)
Purpose: Print the contents of a two-dimensional array
Pre: Array A
Post: Printed elements
Return:
{
    i ← 1
    while (i ≤ r)
    {
        j ← 1
        while (j ≤ c)
        {
            print A[i][j]
            j ← j + 1
        }
        i ← i + 1
    }
}
```

P11-5. Algorithm P11-5 shows the binary search routine in pseudocode (see Chapter 8). Note that we perform the binary search on sorted array. 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.

Algorithm P11-5 *Binary search*

```
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)
}
```

P11-7. The algorithm that insert an element in a sorted array has two parts. Part a shows the main algorithm. Part b shows the algorithm named shiftup called by the insert algorithm.

a. Algorithm P11-7a shows the main algorithm.

Algorithm P11-7a *Main algorithm for insertion*

```
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
}
```

b. Algorithm11-7b shows the auxiliary algorithm used by the main algorithm.

Algorithm P11-7b *The shift-up algorithm used by the insert algorithm*

```
Algorithm: ShiftUp (A, n, i)
Purpose: Shift up all elements one place up from index i.
Pre: A, n, i
Post: None
Return: A
{
    j ← i
    while (j ≤ n + 1)
    {
        A[j] ← A[j + 1]
        j ← j + 1
    }
    return
}
```

P11-9. Algorithm P11-9 shows the routine that adds two fractions.

Algorithm P11-9 *Fraction add*

```
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 Chapter 8)
    y ← (Fr1.denom × Fr2.denom) / x      // y is 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)
}
```

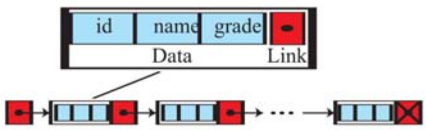
P11-11. Algorithm P11-11 shows the routine in pseudocode that multiplies two fractions.

Algorithm P11-11 *Fraction multiply*

```
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)
}
```

P11-13. Figure P11-13 shows the linked list of records.

Figure P11-13 *Linked list of records*



P11-15. Since *list* = *null*, the **SearchLinkedList** algorithm performs *new* ← *list.* This creates a list with a single node.

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

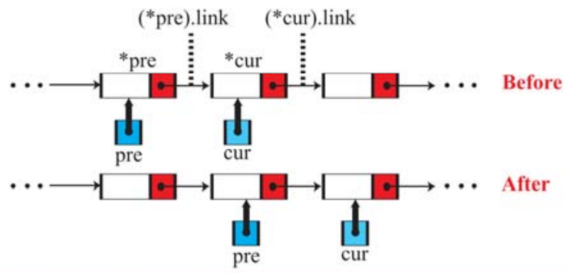
Algorithm P11-17

```
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
}
```

P11-19.

a. Figure P11-19a shows that if **pre** is not null, 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 P11-19a *Moving cur and pre pointers to the right when none is null*



b. However, the statement **pre** ← **(*pre).link** does not work when **pre** is null because, in this case, **(*pre).link** does not exist (Figure P11-19b). For this reason, we should avoid using this method.

Figure P11-19b *Moving pre and cur pointers to the right when pre is null*