Chapter 3: Lists, Stacks, and Queues

3.2 The comments for Exercise 3.4 regarding the amount of abstractness used apply here. The running time of the procedure in Fig. 3.1 is $O(L + P)$.

```
void
PrintLots( List L, List P )
{
        int Counter;
        Position Lpos, Ppos;

        Lpos = First( L );
        Ppos = First( P );
        Counter = 1;
        while( Lpos != NULL && Ppos != NULL )
        {
                if( Ppos->Element == Counter++ )
                {
                        printf( "%? ", Lpos->Element );
                        Ppos = Next( Ppos, P );
                }
                Lpos = Next( Lpos, L );
        }
}
```

Fig. 3.1.

3.3 (a) For singly linked lists, the code is shown in Fig. 3.2.

```
/* BeforeP is the cell before the two adjacent cells that are to be swapped. */
/* Error checks are omitted for clarity. */

void
SwapWithNext( Position BeforeP, List L )
{
        Position P, AfterP;

        P = BeforeP->Next;
        AfterP = P->Next;     /* Both P and AfterP assumed not NULL. */

        P->Next = AfterP->Next;
        BeforeP->Next = AfterP;
        AfterP->Next = P;
}
```

Fig. 3.2.

(b) For doubly linked lists, the code is shown in Fig. 3.3.

```
/* P and AfterP are cells to be switched.  Error checks as before. */

void
SwapWithNext( Position P, List L )
{
        Position BeforeP, AfterP;

        BeforeP = P->Prev;
        AfterP  = P->Next;

        P->Next = AfterP->Next;
        BeforeP->Next = AfterP;
        AfterP->Next = P;
        P->Next->Prev = P;
        P->Prev = AfterP;
        AfterP->Prev = BeforeP;
}
```

Fig. 3.3.

3.4 *Intersect* is shown on page 9.

```
/* This code can be made more abstract by using operations such as          */
/* Retrieve and IsPastEnd to replace L1Pos->Element and L1Pos != NULL.      */
/* We have avoided this because these operations were not rigorously defined. */

List
Intersect( List L1, List L2 )
{
        List Result;
        Position L1Pos, L2Pos, ResultPos;

        L1Pos = First( L1 ); L2Pos = First( L2 );
        Result = MakeEmpty( NULL );
        ResultPos = First( Result );
        while( L1Pos != NULL && L2Pos != NULL )
        {
                if( L1Pos->Element < L2Pos->Element )
                        L1Pos = Next( L1Pos, L1 );
                else if( L1Pos->Element > L2Pos->Element )
                        L2Pos = Next( L2Pos, L2 );
                else
                {
                        Insert( L1Pos->Element, Result, ResultPos );
                        L1 = Next( L1Pos, L1 ); L2 = Next( L2Pos, L2 );
                        ResultPos = Next( ResultPos, Result );
                }
        }
        return Result;
}
```

3.5 Fig. 3.4 contains the code for *Union.*

3.7 (a) One algorithm is to keep the result in a sorted (by exponent) linked list. Each of the MN multiplies requires a search of the linked list for duplicates. Since the size of the linked list is $O(MN)$, the total running time is $O(M^2N^2)$.

(b) The bound can be improved by multiplying one term by the entire other polynomial, and then using the equivalent of the procedure in Exercise 3.2 to insert the entire sequence. Then each sequence takes $O(MN)$, but there are only M of them, giving a time bound of $O(M^2N)$.

(c) An $O(MN \log MN)$ solution is possible by computing all MN pairs and then sorting by exponent using any algorithm in Chapter 7. It is then easy to merge duplicates afterward.

(d) The choice of algorithm depends on the relative values of M and N. If they are close, then the solution in part (c) is better. If one polynomial is very small, then the solution in part (b) is better.

```
List
Union( List L1, List L2 )
{
        List Result;
        ElementType InsertElement;
        Position L1Pos, L2Pos, ResultPos;

        L1Pos = First( L1 ); L2Pos = First( L2 );
        Result = MakeEmpty( NULL );
        ResultPos = First( Result );
        while ( L1Pos != NULL && L2Pos != NULL ) {
                if( L1Pos->Element < L2Pos->Element ) {
                        InsertElement = L1Pos->Element;
                        L1Pos = Next( L1Pos, L1 );
                }
                else if( L1Pos->Element > L2Pos->Element ) {
                        InsertElement = L2Pos->Element;
                        L2Pos = Next( L2Pos, L2 );
                }
                else {
                        InsertElement = L1Pos->Element;
                        L1Pos = Next( L1Pos, L1 ); L2Pos = Next( L2Pos, L2 );
                }
                Insert( InsertElement, Result, ResultPos );
                ResultPos = Next( ResultPos, Result );
        }
        /* Flush out remaining list */
        while( L1Pos != NULL ) {
                Insert( L1Pos->Element, Result, ResultPos );
                L1Pos = Next( L1Pos, L1 ); ResultPos = Next( ResultPos, Result );
        }
        while( L2Pos != NULL ) {
                Insert( L2Pos->Element, Result, ResultPos );
                L2Pos = Next( L2Pos, L2 ); ResultPos = Next( ResultPos, Result );
        }
        return Result;
}
```

Fig. 3.4.

3.8 One can use the *Pow* function in Chapter 2, adapted for polynomial multiplication. If P is small, a standard method that uses $O(P)$ multiplies instead of $O(\log P)$ might be better because the multiplies would involve a large number with a small number, which is good for the multiplication routine in part (b).

3.10 This is a standard programming project. The algorithm can be sped up by setting $M' = M \ mod \ N$, so that the hot potato never goes around the circle more than once, and then if $M' > N/2$, passing the potato appropriately in the alternative direction. This requires a doubly linked list. The worst-case running time is clearly $O(N\ min(M, N))$, although when these heuristics are used, and M and N are comparable, the algorithm might be significantly faster. If $M = 1$, the algorithm is clearly linear. The VAX/VMS C compiler's memory management routines do poorly with the particular pattern of *free*s in this case, causing $O(N \log N)$ behavior.

3.12 Reversal of a singly linked list can be done nonrecursively by using a stack, but this requires $O(N)$ extra space. The solution in Fig. 3.5 is similar to strategies employed in garbage collection algorithms. At the top of the *while* loop, the list from the start to *PreviousPos* is already reversed, whereas the rest of the list, from *CurrentPos* to the end, is normal. This algorithm uses only constant extra space.

```
/* Assuming no header and L is not empty. */

List
ReverseList( List L )
{
        Position CurrentPos, NextPos, PreviousPos;

        PreviousPos = NULL;
        CurrentPos = L;
        NextPos = L->Next;
        while( NextPos != NULL )
        {
                CurrentPos->Next = PreviousPos;
                PreviousPos = CurrentPos;
                CurrentPos = NextPos;
                NextPos = NextPos->Next;
        }
        CurrentPos->Next = PreviousPos;
        return CurrentPos;
}
```

Fig. 3.5.

3.15 (a) The code is shown in Fig. 3.6.

(b) See Fig. 3.7.

(c) This follows from well-known statistical theorems. See Sleator and Tarjan's paper in the Chapter 11 references.

3.16 (c) *Delete* takes $O(N)$ and is in two nested for loops each of size N, giving an obvious $O(N^3)$ bound. A better bound of $O(N^2)$ is obtained by noting that only N elements can be deleted from a list of size N, hence $O(N^2)$ is spent performing deletes. The remainder of the routine is $O(N^2)$, so the bound follows.

(d) $O(N^2)$.

```
/* Array implementation, starting at slot 1 */

Position
Find( ElementType X, List L )
{
        int i, Where;

        Where = 0;
        for( i = 1; i < L.SizeOfList; i++ )
                if( X == L[i].Element )
                {
                        Where = i;
                        break;
                }

        if( Where )             /* Move to front. */
        {
                for( i = Where; i > 1; i-- )
                        L[i].Element = L[i-1].Element;
                L[1].Element = X;
                return 1;
        }
        else
                return 0;       /* Not found. */
}
```

Fig. 3.6.

(e) Sort the list, and make a scan to remove duplicates (which must now be adjacent).

3.17 (a) The advantages are that it is simpler to code, and there is a possible savings if deleted keys are subsequently reinserted (in the same place). The disadvantage is that it uses more space, because each cell needs an extra bit (which is typically a byte), and unused cells are not freed.

3.21 Two stacks can be implemented in an array by having one grow from the low end of the array up, and the other from the high end down.

3.22 (a) Let E be our extended stack. We will implement E with two stacks. One stack, which we'll call S, is used to keep track of the *Push* and *Pop* operations, and the other, M, keeps track of the minimum. To implement *Push(X,E)*, we perform *Push(X,S)*. If X is smaller than or equal to the top element in stack M, then we also perform *Push(X,M)*. To implement *Pop(E)*, we perform *Pop(S)*. If X is equal to the top element in stack M, then we also *Pop(M)*. *FindMin(E)* is performed by examining the top of M. All these operations are clearly $O(1)$.

(b) This result follows from a theorem in Chapter 7 that shows that sorting must take $\Omega(N \log N)$ time. $O(N)$ operations in the repertoire, including *DeleteMin*, would be sufficient to sort.

```
/* Assuming a header. */

Position
Find( ElementType X, List L )
{
        Position PrevPos, XPos;

        PrevPos = FindPrevious( X, L );
        if( PrevPos->Next != NULL )  /* Found. */
        {
                XPos = PrevPos ->Next;
                PrevPos->Next = XPos->Next;
                XPos->Next = L->Next;
                L->Next = XPos;
                return XPos;
        }
        else
                return NULL;
}
```

Fig. 3.7.

3.23 Three stacks can be implemented by having one grow from the bottom up, another from the top down, and a third somewhere in the middle growing in some (arbitrary) direction. If the third stack collides with either of the other two, it needs to be moved. A reasonable strategy is to move it so that its center (at the time of the move) is halfway between the tops of the other two stacks.

3.24 Stack space will not run out because only 49 calls will be stacked. However, the running time is exponential, as shown in Chapter 2, and thus the routine will not terminate in a reasonable amount of time.

3.25 The queue data structure consists of pointers *Q->Front* and *Q->Rear,* which point to the beginning and end of a linked list. The programming details are left as an exercise because it is a likely programming assignment.

3.26 (a) This is a straightforward modification of the queue routines. It is also a likely programming assignment, so we do not provide a solution.