Solutions to Exercises in Chapter 4

4.1 To access the *k*th element of the SLL headed by *first*, counting the first element as 0:

1. Set *curr* to *first*.
2. Repeat *k* times:
   2.1. If *curr* is null, terminate with answer *none*.
   2.2. Set *curr* to node *curr*'s successor.
3. Terminate with answer *curr*.

This algorithm follows between 1 and *n* links, i.e., $(n+1)/2$ links on average. Its time complexity is $O(n)$.

4.2 To access the *k*th element of the DLL headed by (*first*, *last*), counting the first element as 0:

1. Let *n* be the length of the DLL headed by (*first*, *last*).
2. If $2k < n$:
   2.1. Set *curr* to *first*.
   2.2. Repeat *k* times:
      2.2.1. If *curr* is null, terminate with answer *none*.
      2.2.2. Set *curr* to node *curr*'s successor.
3. If $2k \geq n$:
   2.1. Set *curr* to *last*.
   2.2. Repeat $n–1–k$ times:
      2.2.1. If *curr* is null, terminate with answer *none*.
      2.2.2. Set *curr* to node *curr*'s predecessor.
4. Terminate with answer *curr*.

If the DLL's length is immediately available, step 1 follows 0 links. Either step 2 or 3 follows between 0 and $(n–1)/2$ links, i.e., $(n–1)/4$ links on average. This algorithm's time complexity is $O(n)$.

If the DLL's length is not immediately available, step 1 would have to follow *n* links, so it would be better just to mimic the algorithm of Exercise 4.1.

4.3 To reverse the elements of the SLL headed by *first*:

1. Set *curr* to *first* and set *pred* to null.
2. While *curr* is not null, repeat:
   2.1. Let *succ* be node *curr*'s successor.
   2.2. Set node *curr*'s successor to *pred*.
   2.3. Set *pred* to *curr*.
   2.4. Set *curr* to *succ*.
3. Set *first* to *pred*.
4. Terminate.

This algorithm follows *n* links, so its time complexity is $O(n)$. Its space complexity is $O(1)$.

4.4 To reverse the elements of the DLL headed by (*first*, *last*):

1. Set *curr* to *first*.
2. While *curr* is not null, repeat:
   2.1. Let *succ* be node *curr*'s successor.
   2.2. Swap node *curr*'s predecessor and successor links.
   2.2. Set *curr* to *succ*.
3. Swap *first* and *last*.
4. Terminate.

This algorithm follows *n* links, so its time complexity is *O*(*n*). Its space complexity is *O*(1).

4.5 To test whether the SLL headed by *first* is a palindrome:

1. Let *n* be the length of the SLL headed by *first*.
2. Copy characters in reverse order from the first *n*/2 nodes of the SLL headed by *first* into another SLL headed by *prefix*, and let *suffix* be a link to the next node of the SLL headed by *first*.
3. If *n* is odd, set *suffix* to node *suffix*'s successor.
4. Let *matched* be the result of testing whether the SLL headed by *prefix* matches the SLL headed by *suffix*.
5. Terminate with answer *matched*.

To copy characters in reverse order from the first *k* nodes of the SLL headed by *first* into another SLL headed by *prefix*, and let *suffix* be a link to the next node of the SLL headed by *first*:

1. Set *curr* to *first*, and set *prefix* to null.
2. Repeat *k* times:
   2.1. Insert node *curr*'s character before the first node of the SLL headed by *prefix*.
   2.2. Set *curr* to node *curr*'s successor.
3. Set *suffix* to *curr*.
4. Terminate with answers *prefix* and *suffix*.

To test whether the SLL headed by *prefix* matches the SLL headed by *suffix*:

1. Set *p* to *prefix*, and set *s* to *suffix*.
2. While *p* and *s* are not null, repeat:
   2.1. If node *p*'s character ≠ node *s*'s character, terminate with answer *false*.
   2.2. Set *p* to node *p*'s successor, and set *s* to node *s*'s successor.
3. Terminate with answer *true*.

The main algorithm performs *n*/2 character comparisons. Step 1 follows either 0 or *n* links, depending on whether the SLL's length is immediately available or not. Step 2 follows *n*/2 links. Step 4 follows *n*/2 links in each of two SLLs. In total, the algorithm follows either 3*n*/2 or 5*n*/2 links.

4.6 To test whether the DLL headed by (*first*, *last*) is a palindrome:

1. Set *p* to *first*, and set *s* to *last*.
2. While *p* and *s* are not the same node, repeat:
   2.1. If node *p*'s character ≠ node *s*'s character, terminate with answer *false*.
   2.2. If node *p* is node *s*'s predecessor, terminate with answer *true*.
   2.2. Set *p* to node *p*'s successor, and set *s* to node *s*'s predecessor.
3. Terminate with answer *true*.

The algorithm performs *n*/2 character comparisons. It follows *n*/2 successor links and *n*/2 predecessor links. In total, it follows about *n* links.

4.8 If a sorted DLL contains words in alphabetical order, it would be advantageous to search the DLL right-to-left when the target word's initial letter is in the second half of the alphabet.

4.9 To find which if any node of the unsorted DLL headed by (*first*, *last*) contains an element equal to *target* (version that searches simultaneously from both ends):

1. If *first* and *last* are null, terminate with answer *none*.
2. Set *p* to *first*, and set *s* to *last*.
3. Repeat:
   3.1. If *target* is equal to node *p*'s element, terminate with answer *p*.
   3.2. If *target* is equal to node *s*'s element, terminate with answer *s*.
   3.3. If *p* and *s* are the same node, or node *p* is node *s*'s predecessor, terminate with answer *none*.
   3.4. Set *p* to node *p*'s successor, and set *s* to node *s*'s predecessor.

On a successful search, this algorithm performs between 1 and *n* comparisons, i.e., (*n*+1)/2 comparisons on average. On an unsuccessful search, it performs *n* comparisons. Thus it is no better than the original unsorted DLL linear search algorithm.

This algorithm's time complexity is $O(n)$.

4.10 To find which if any node of the sorted DLL headed by (*first, last*) contains an element equal to *target* (version that searches simultaneously from both ends):

1. If *first* and *last* are null, terminate with answer *none*.
2. Set *p* to *first*, and set *s* to *last*.
3. Repeat:
   3.1. If *target* is equal to node *p*'s element, terminate with answer *p*.
   3.2. If *target* is equal to node *s*'s element, terminate with answer *s*.
   3.3. If *p* and *s* are the same node, or node *p* is node *s*'s predecessor, or *target* is less than node *p*'s element, or *target* is greater than node *s*'s element, terminate with answer *none*.
   3.4. Set *p* to node *p*'s successor, and set *s* to node *s*'s predecessor.

On a successful or unsuccessful search, this algorithm performs between 1 and *n* comparisons, i.e., (*n*+1)/2 comparisons on average. Thus it is no better than the original sorted DLL linear search algorithm.

This algorithm's time complexity is $O(n)$.

4.11 To find which if any node of the unsorted SLL headed by *first* contains an element equal to *target* (version that moves the node to the front of the SLL):

1. Set *pred* to null.
2. For each node *curr* of the SLL headed by *first*, repeat:
   2.1. If *target* is equal to node *curr*'s element:
      2.1.1. If *pred* is not null:
         2.1.1.1. Set node *pred*'s successor to node *curr*'s successor.
         2.1.1.2. Set node *curr*'s successor to *first*.
         2.1.1.3. Set *first* to *curr*.
      2.1.2. Terminate with answer *curr*.
   2.2. Set *pred* to *curr*.
3. Terminate with answer *none*.

If the same *x* is searched for 50 times out of the next 100 searches, *x* will be the first or second element in the SLL for most of the time, so each of the 50 searches for *x* will perform only 1 or 2 comparisons. Each of the remaining 50 searches (if successful) will perform about *n*/2 comparisons on average. The total number of comparisons for the 100 searches will be about 100 + 25*n*.

If we use the original unsorted SLL search algorithm, each of the 100 searches (if successful) will perform about *n*/2 comparisons on average. The total number of comparisons will be about 50*n*. Thus the above algorithm is faster for all but small values of *n*.

4.12 To delete the node containing element *elem* in the SLL headed by *first*:

1. Set *pred* to null.
2. For each node *curr* of the SLL headed by *first*, repeat:
   2.1. If *target* is equal to node *curr*'s element:
      2.1.1. Let *succ* be node *curr*'s successor.
      2.1.2. If *pred* is null, set *first* to *succ*.
      2.1.3. If *pred* is not null, set node *pred*'s successor to *succ*.
      2.1.4. Terminate.
   2.2. Set *pred* to *curr*.
3. Terminate.

To delete the node containing element *elem* in the DLL headed by (*first*, *last*):

1. For each node *curr* of the DLL headed by (*first*, *last*), repeat:
   2.1. If *target* is equal to node *curr*'s element:
      2.1.1. Let *pred* and *succ* be node *curr*'s successor and predecessor, respectively.
      2.1.2. If *pred* is null, set *first* to *succ*.
      2.1.3. If *pred* is not null, set node *pred*'s successor to *succ*.
      2.1.4. If *succ* is null, set *last* to *pred*.
      2.1.5. If *succ* is not null, set node *succ*'s predecessor to *pred*.
      2.1.6. Terminate.
3. Terminate.

4.14 To sort the SLL headed by *first* (selection sort version):

1. For each node *curr* of the SLL headed by *first*, repeat:
   1.1. Set *p* such that node *p* contains the least element in the SLL headed by *curr*.
   1.2. If $p \neq curr$, swap node *p*'s element and node *curr*'s element.
2. Terminate.

To sort a DLL, a similar algorithm can be used.

4.15 To sort the SLL headed by *first* (quick-sort version):

1. If neither *first* nor *first*'s successor is null:
   1.1. Partition the SLL headed by *first* into three separate SLLs, such that the SLL headed by *center* contains a single element *pivot*, the SLL headed by *left* contains only elements less than or equal to *pivot*, and the SLL headed by *right* contains only elements greater than or equal to *pivot*.
   1.2. Sort the SLL headed by *left*.
   1.3. Sort the SLL headed by *right*.
   1.4. Let *lastleft* be the last node of the SLL headed by *left*.
   1.5. Set node *lastleft*'s successor to *center*, and set node *center*'s successor to *right*.
2. Terminate.

To sort a DLL, a similar algorithm can be used.

4.16 To insert *elem* after node *pred* in the SLL headed by *first*:

1. Let *succ* be node *pred*'s successor.
2. Make *ins* a link to a newly-created node with element *elem* and successor *succ*.
3. Set node *pred*'s successor to *ins*.
4. Terminate.

(*Note:* If we wish to insert *elem* before the SLL's first node, *pred* will be the dummy node.)

To delete node *del* in the nonempty SLL headed by *first*:

1. Let *succ* be node *del*'s successor.
2. Let *pred* be node *del*'s predecessor.
3. Set node *pred*'s successor to *succ*.
4. Terminate.

(*Note:* Like the corresponding step of Algorithm 4.17, step 2 must find the predecessor by traversing the SLL from its first node.)

The above algorithms are neater than the original algorithms, but they have the same time complexities, $O(1)$ and $O(n)$ respectively.