Exercises 6.4

2. Write an algorithm to determine the average of a linked list of real numbers with first node pointed to by `first`.

Solution:

```
template <class T>
double List<T>::getAverage() {
   double total = 0.0;
   int count = 0;

   Node * ptr = _first;

   while (ptr != NULL) {
      ++count;
      total += ptr->data;
      ptr = ptr->next;
   }

   if (count==0) {
      return 0;
   } else {
      return (total/count);
   }
}
```

4. Write an algorithm to determine whether the data items in a linked list with first node pointed to by `first` are in ascending order.

Solution:

```
template <class T>
bool List<T>::isAscendingOrder() {
   Node * ptr = _first;

   if (_first==NULL || _first->next==NULL) {
      return truie;
   }

   while (ptr->next != null) {
      if (ptr->data>ptr->next->data) {
         return false;
      }
      ptr = ptr->next;
   }

   return true;
}
```

9. The *shuffle-merge* of two lists $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_m$ is the list

$$z = \begin{cases} x_1, y_1, x_2, y_2, \ldots, x_n, y_n, y_{n+1}, y_{n+2}, \ldots, y_m & n < m \\ x_1, y_1, x_2, y_2, \ldots, x_m, y_m, x_{m+1}, x_{m+2}, \ldots, x_n & n > m \\ x_1, y_1, x_2, y_2, \ldots, x_n, y_n & n = m \end{cases}$$

Write an algorithm to shuffle-merge two linked lists with first nodes pointed to by `first1` and `first2`, respectively. The items in these two lists should be copied to produce the new list; the original lists should not be destroyed.

Solution:

```
Node * mergeLists(Node * list1, Node * list2) {
   if (list1 == NULL) {
      Node * newList;
      copyList(list2, newList);

      return newList;
   }

   Node * newListFirst = new Node(list1->data);
   Node * newListPtr = newListFirst;;

   Node * ptr1 = list1;
   Node * ptr2 = list2;

   while (ptr1 != NULL && ptr2 != NULL) {
      // create a new node for ptr2 data
      Node * newNode2 = new Node(ptr2->data);

      // create a new node for ptr1 data
      Node * newNode1 = new Node(ptr1->data);

      // add the new nodes to the merged list
      newListPtr->next = ptr2;
      newListPtr = newListPtr->next;
      newListPtr->next = ptr1;
      newListPtr = newListPtr->next;

      // advance the input lists
      ptr1 = ptr1->next;
      ptr2 = ptr2->next;
   }

   while (ptr1 != NULL) {
      // create a new node for ptr1 data
      Node * newNode = new Node(ptr1->data);
      // add the new node to the merged list
      newListPtr->next = newNode;
```

```
            newListPtr = newListPtr->next;t;
         }

         while (ptr2 != NULL) {
            // create a new node for ptr1 data
            Node * newNode = new Node(ptr1->data);
            // add the new node to the merged list
            newListPtr->next = newNode;
            newListPtr = newListPtr->next;
         }

         // free memory
         delete ptr1;
         delete ptr2;
         delete newListPtr;

         return newListFirst;
      }
```

10. Proceed as in Exercise 9, but do not copy the items. Just change links in the two lists (thus destroying the original lists) to produce the merged list.

Solution:

```
Node * mergeLists(Node * list1, Node * list2) {
   if (list1 == NULL) {
      return list2;
   }

   // the start of this list points to the first element of list1
   Node * newListFirst = list1;
   Node * newListPtr = newListFirst;

   // to walk over each list
   Node * ptr1 = list1->next;
   Node * ptr2 = list2;

   // shuffle-merge the two lists
   while (ptr2 != NULL && ptr1 != NULL) {
      // add the element from list 2
      newListPtr->next = ptr2;
      newListPtr = newListPtr->next;

      // advance the list2 pointer
      ptr2 = ptr2->next;

      // add the element from list 2
      newListPtr->next = ptr1;
      newListPtr = newListPtr->next;

      // advance the list2 pointer
      ptr1 = ptr1->next;
   }

   // add remaining element from list 2
   it (ptr1 == NULL && ptr2!=NULL) {
      newListPtr->next = ptr2;
   }

   // free memory
   delete ptr1;
   delete ptr2;
   delete newListPtr;

   return newListFirst;
}
```

Exercises 6.6

1. An ordered linked list of characters has been constructed using the array-based implementation described in this section. The following diagram shows the current contents of the array that stores the elements of th linked list and storage pool:

Node	Data	Next
[0]	J	3
[1]	Z	6
[2]	C	0
[3]	P	-1
[4]	B	2
[5]	M	1
[6]	K	7
[7]	Q	8
[8]	?	9
[9]	?	-1

`first` = 4 `free` = 5

(a) List the elements of this list.

Solution:
B, C, J, P

(b) List the nodes in the storage pool in the order in which they are linked together.

Solution:
M, Z, K, Q, ?, ?

2. Assuming the contents of the array node pictured in Exercise 1, show the contents of node and the values of `first` and `free` after the letter F is inserted into the list so that the resulting list is in alphabetical order.

Solution:
`first` = 4 `free` = 1

Node	Data	Next
[0]	J	3
[1]	Z	6
[2]	C	5
[3]	P	-1
[4]	B	2
[5]	F	0
[6]	K	7
[7]	Q	8
[8]	?	9
[9]	?	-1

3. Proceed as in Exercise 2, but for the operation Delete J.

Solution:

`first` = 4 `free` = 0

Node	Data	Next
[0]	J	5
[1]	Z	6
[2]	C	3
[3]	P	-1
[4]	B	2
[5]	M	1
[6]	K	7
[7]	Q	8
[8]	?	9
[9]	?	-1

4. Proceed as in Exercise 2, but for the following sequence of operations: Delete J, Delete P, Delete C, Delete B

Solution:

`first` = -1 `free` = 4

Node	Data	Next
[0]	J	5
[1]	Z	6
[2]	C	3
[3]	P	0
[4]	B	2
[5]	M	1
[6]	K	7
[7]	Q	8
[8]	?	9
[9]	?	-1

5. Proceed as in Exercise 2, but for the following sequence of operations: Insert A, Delete P, Insert K, Delete C

Solution: `first` = 5 `free` = 2

Node	Data	Next
[0]	J	3
[1]	Z	7
[2]	C	1
[3]	K	-1
[4]	B	0
[5]	A	4
[6]	K	8
[7]	Q	9
[8]	?	10
[9]	?	0

6. Assuming the array-based implementation as described in this section, write a function to count the nodes in a linked list.

Solution:

```
template <class T>
int List<T>::nodeCount() {
   int count = 0;
   int ptr = _first;

   while (ptr != NULL) {
      ++count;
      ptr = _items[ptr].next;
   }

   return count;
}
```

7. Assuming the array-based implementation as described in this section, write a boolean-valued function that determines whether the data items in the list are arranged in ascending order.

Solution:

```
template <class T>
bool List<T>::isAscendingOrder() {
   if (getSize()<=1) {
      return true;
   }

   int prevPtr = _first;
   int currPtr = node[_first].next;

   while (currPtr != NULL) {
      if (_items[prevPtr].data > _items[currPtr].data) {
         return false;
      }

      prevPtr = currPtr;
      currPtr = _items[currPtr].next;
   }

   return true;
}
```

8. Assuming the array-based implementation as described in this section, write a function that returns a pointer to the last node in a linked list.

Solution:

```
template <class T>
int List<T>::getLastNode() {
   int currentPointer = _first;
   int lastPointer = _first;

   while (currentPointer != NULL) {
      lastPointer = currentPointer;
      currentPointer = _items[currentPointer].next;
   }

   return lastPointer;
}
```

9. Assuming the array-based implementation as described in this section, write a function to reverse a linked list in the manner described in Exercise 12 of Section 6.4.

Solution:

```
template <class T>
void List<T>::reverseList() {
   int currentPointer = _first;
   int previousPointer = NULL;
   int nextPointer = NULL;

   while (currentPointer != NULL) {
      nextPointer = _items[currentPointer].next;
      _items[currentPointer].next = previousPointer;
      previousPointer = currentPointer;
      currentPointer = nextPointer;
   }

   _first = previousPointer;
}
```

Worksheet Questions

1. Give asymptotic bounds on the worst-case time complexity of your algorithms for Exercises 6.4 questions 2., 4., 9. and 10 and for Exercises 6.6 questions 7., 8., and 9. *Justify your answer.*

Solution:

Each of the functions must walk over the linked list one time. If the linked list has n items, then this takes $O(n)$ time.

In the shuffle-merge algorithms (questions 9 & 10), we must walk over two different linked lists in the worst-case. These lists may have different lengths and we do not know which is longer. Therefore the worst-case time is $O(n + m)$ where n is the length of the first list and m is the length of the second list.