Exercises 9.4

For Exercises 1-11, assume that the following declarations have been made:

```
vector<int> number,
            v(10,20),
            w(10);
int num;
```

Assume also that, for exercises that involve input, the following values are entered:

```
99 33 44 88 22 11 55 66 77 -1
```

Describe the contents of the given vector after the statements are executed.

3.

```
for (;;) {
   cin >> num;
   if (num < 0) break;
   number.push_back(num);
}
```

Solution:

number[0] = 99;
number[1] = 33;
number[2] = 44;
number[3] = 88;
number[4] = 22;
number[5] = 11;
number[6] = 55;
number[7] = 66;
number[8] = 77;

For Exercises 5-11, assume that the loop in Exercise 3 has been executed.

10.

```
vector<int>::iterator iter = number.begin();
while (*iter > 25) {
   number.erase(iter);
   iter++;
}
```

Solution:

number[0] = 22;
number[1] = 11;
number[2] = 55;
number[3] = 66;
number[4] = 77;

11.

```
for (vector<int>::iterator iter = number.begin();
                           iter != number.end();
                           iter++)
     w.push_back(*iter+1);
```

Solution:

w[0] = 0;
w[1] = 0;
w[2] = 0;
w[3] = 0;
w[4] = 0;
w[5] = 0;
w[6] = 0;
w[7] = 0;
w[8] = 0;
w[9] = 0;
w[10] = 100;
w[11] = 34;
w[12] = 45;
w[13] = 89;
w[14] = 23;
w[15] = 12;
w[16] = 56;
w[17] = 67;
w[18] = 78;

Exercises 11.1

2. Write an algorithm or code segment for searching a circular linked list for a given item.

Solution:

```
template <class T>
Node * List<T>::search(T item) {
   if (_first==0) {
      return 0;
   }

   if (_first->data==item) {
      return _first;
   }

   Node * ptr = _first->next;

   while (ptr != _first) {
      if (ptr->data == item) {
         return ptr;
      } else {
         ptr = ptr->next;
      }
   }

   return NULL;
}
```

3. Proceed as in Exercise 2, but assume that the list is ordered so that the elements are in ascending order.

Solution:

```
template <class T>
Node * List<T>::search(T item) {
   if (_first==0) {
      return 0;
   }

   if (_first->data==item) {
      return _first;
   }

   Node * ptr = _first->next;

   while (ptr != _first && ptr->data <= item) {
      if (ptr->data == item) {
         return ptr;
      } else {
         ptr = ptr->next;
      }
   }

   return NULL;
}
```

4. Write an algorithm or code segment for locating the *n*th successor of an item in a circular linked list (the *n*th item that follows the given item in the list).

Solution:

```
template <class T>
Node * List<T>::getSuccessor(const T& item, const int& n) {
   Node * itemLocation = search(item);

   if (itemLocation == NULL) {
      return NULL;
   }

   if (n == 0) {
      return itemLocation;
   }

   for (int i=0; i<n; ++i) {
      itemLocation=itemLocation->next;
   }

   return itemLocation;
}
```

6. The *shuffle-merge* operation on two lists was defined in Exercise 9 of Section 6.4. Write an algorithm to shuffle-merge two circular-linked lists. The items in the lists are to be copied to produce the new circular-linked lists; the original lists are not to be destroyed.

Solution: Assume that the linked lists have head nodes. If there are no head nodes, then you can supply temporary ones.

```
template <class T>
Node * List<T>::shuffleMerge(Node * list1, Node * list2) {
   // keep track of the head node of each circular linked list
   Node * head1 = list1;
   Node * head2 = list2;

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

   // for the new list
   Node * newListHead = new Node();
   Node * newListPtr = newListHead;

   while (ptr1!=head1 && ptr2!=head2) {
      // create a new node for the current list1 node
      Node * newNode1 = new Node(ptr1->data);
      // add the new node to the merged list
      newListPtr->next = newNode1;
      newListPtr = newListPtr->next;
      // advance ptr1
      ptr1=ptr1->next;

      // create a new node for the current list2 node
      Node * newNode2 = new Node(ptr2->data);
      // add the new node to the merged list
      newListPtr->next = newNode2;
      newListPtr = newListPtr->next;
      // advance ptr2
      ptr2 = ptr2->next;
   }

   while (ptr1!=head1) {
      // create a new node for the current list1 node
      Node * newNode1 = new Node(ptr1->data);
      // add the new node to the merged list
      newListPtr->next = newNode1;
      newListPtr = newListPtr->next;
      // advance ptr1
      ptr1=ptr1->next;
   }
   while (ptr2!=head2) {
      // create a new node for the current list2 node
      Node * newNode2 = new Node(ptr2->data);
      // add the new node to the merged list
      newListPtr->next = newNode2;
      newListPtr = newListPtr->next;
      // advance ptr2
      ptr2 = ptr2->next;
   }

   newListPtr->next = newListHead;

   // free memory - code omitted for posted solutions

   return newListHead;
}
```

7. Proceed as in Exercise 6, but do not copy the items. Just change links in the two lists (thus destroying the original lists) to produce the merged list.

Solution: Assume that the linked lists have head nodes. If there are no head nodes, then you can supply temporary ones.

```
template <class T>
Node * List<T>::shuffleMerge(Node * list1, Node * list2) {
   // keep track of the head node of each circular linked list
   Node * head1 = list1;
   Node * head2 = list2;

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

   // check if either list is empty
   if (ptr1 == head1) {
      return head2;
   }
   if (ptr2 == head2) {
      return head1;
   }

   // for the new list
   Node * newListHead = new Node();
   Node * newListPtr = newListHead;

   // shuffle-merge the two lists
   while (ptr1!=head1 && ptr2!=head2) {
      // add the new node to the merged list
      newListPtr->next = ptr1;
      newListPtr = newListPtr->next;
      // advance ptr1
      ptr1=ptr1->next;

      // add the new node to the merged list
      newListPtr->next = ptr2;
      newListPtr = newListPtr->next;
      // advance ptr2
      ptr2 = ptr2->next;
   }

   if (ptr1!=head1) {
      newListPtr->next = ptr1;
      newListPtr = newListPtr->next;
   }

   while ((newListPtr->next != head1) && (newListPtr->next !=head2)) {
      newListPtr = newListPtr->next;
   }

   newListPtr->next = newListHead;

   // free memory - code omitted for posted solutions

   return newListHead;
}
```