1. [20 pts] For this question, you will design and implement a new member function for the DListT template given in the specification for Minor Project 1. The member function must conform to the following interface:

```cpp
template <typename T>
void DListT<T>::alternatingSplit(DListT<T>& L1, DListT<T>& L2) const {
    L1.Clear();                  // clear parameters of old data
    L2.Clear();
    DNodeT<T>* Curr = Fore->Next;  // get handle on first data node
    bool Even = true;              // keep track of even/odd count
    while ( Curr != Aft ) {        // stop when rear sentinel is reached
        if ( Even ) { // append element to first split
            DNodeT<T>* Copy = new DNodeT<T>(Curr->Element, L1.Aft->Prev, L1.Aft);
            L1.Aft->Prev->Next = Copy;
            L1.Aft->Prev       = Copy;
            if ( L1.Fore->Next == L1.Aft )
                L1.Fore->Next = Copy;
        } else { // append element to second split
            DNodeT<T>* Copy = new DNodeT<T>(Curr->Element, L2.Aft->Prev, L2.Aft);
            L2.Aft->Prev->Next = Copy;
            L2.Aft->Prev       = Copy;
            if ( L2.Fore->Next == L2.Aft )
                L2.Fore->Next = Copy;
        }
        Even  = !Even;       // reset even/odd count
        Curr = Curr->Next;   // step to next source node, if any
    }
}
```

The function will initialize the two DListT objects passed to it as parameters so that the first contains the even-numbered elements of the original list, and the second contains the odd-numbered elements of the original list. That is, if the original list contains:

\{4, 17, 3, 12, 5, 21, 23, 41, 7\}

then L1 and L2 will be the following lists, respectively:

\{4, 3, 5, 23, 7\} and \{17, 12, 21, 41\}

The order of the elements in the two splits of the original list does matter. Your solution should make no assumptions about the contents of the original list or about the contents of L1 and L2 when they are passed in.
2. [10 pts] Write an implementation for a `DListT` client function (i.e., not a member or a friend) that will determine whether the elements in a given `DListT` object are stored in non-strict ascending order.

Your solution must conform to the following interface:

```cpp
template<typename T> bool isAscending(const DListT<T>& List) {  
    if ( List.isEmpty() ) return true;  // empty list is vacuously ascending  
    typename DListT<T>::const_iterator Pos = List.begin();  
    while ( Pos != --List.end() ) {  // quit when last data node is reached  
        if ( *Pos > *(++Pos) ) {  // compare current elem to next one  
            return false;  
        }  
    }  
    return true;  
}
```
3. [10 pts] Write an implementation for a DListT client function (i.e., not a member or a friend) that will merge two ascending-ordered DListT objects into a single ascending-order DListT object, without modifying the original objects. For example, if L1 and L2 are the following lists, respectively:

\[\{4, 5, 11, 17\}\] and \[\{2, 7, 11, 41\}\]

then the returned list should be:

\[\{2, 4, 5, 7, 11, 11, 17, 41\}\]

If either of the given lists is not in ascending order, an empty list should be returned. Your solution must conform to the following interface:

```
template <typename T>
DListT<T> mergeAscending(const DListT<T>& L1, const DListT<T>& L2) {
    DListT<T> Merger; // create list object to return
    // make sure parameters are strictly ascending
    if ( !isAscending(L1) || !isAscending(L2) ) return Merger;
    // get handles on elements in source lists
    typename DListT<T>::const_iterator nextL1 = L1.begin();
    typename DListT<T>::const_iterator nextL2 = L2.begin();
    while ( nextL1 != L1.end() && nextL2 != L2.end() ) {
        // determine which list head gets merged next, and do so
        if ( *nextL1 < *nextL2 ) {
            Merger.Insert( Merger.end(), *nextL1 );
            ++nextL1;
        } else {
            Merger.Insert( Merger.end(), *nextL2 );
            ++nextL2;
        }
    }
    // merge in tail of any non-empty lists
    while ( nextL1 != L1.end() ) {
        Merger.Insert( Merger.end(), *nextL1 );
        ++nextL1;
    }
    while ( nextL2 != L2.end() ) {
        Merger.Insert( Merger.end(), *nextL2 );
        ++nextL2;
    }
    return Merger;
}
```