CS14 Fall 2015 Lab 3
Using Mergesort to Efficiently Sort a Linked List

January 28, 2016

DUE: Submit by iLearn by 11:59pm Sunday, February 7, 2015
LATE DEADLINE (50% penalty) 11:59pm Monday, February 8, 2015

1 Introduction

The purpose of this assignment is to give you practice using templates and manipulating linked lists.

Recall that either a vector (array) or linked list can be used as the ADT to implement a sequence of identical objects. However, since a vector implementation supports direct access to any individual object in the sequence, it is preferred over a linked-list implementation when the sequence of objects must be sorted.

In this assignment, however, you will see how this disadvantage of linked lists can be avoided by using the mergesort algorithm, which was originally used for sorting data stored on magnetic tapes because it only needs to access the data sequentially.

`https://en.wikipedia.org/wiki/Merge_sort#Use_with_tape_drives`

2 The Underlying Linked List Structure

Represent the sequence of objects by the STL implementation for doubly-linked lists, without any changes. Test your program when the list element data type is a simple `int` as well as the more complex user-defined structure `class Fraction`, which is defined here.

`http://www.cs.ucr.edu/~mart/CS14/frac.cpp`

Your driver program should accept *one* input filename as a command-line argument and generate *one* similarly-named output file with the original filename extension (if any) changed to "`.out`". For example, if the input file were `sample_input.txt`, then the output file would be `sample_input.out`.

The input file is a plain ASCII text file, consisting of a variable number of test cases (with two rows of data per test case), terminating with end-of-file. For each test case, the first row contains a single character, where "i" indicates that the next row contains a sequence of integers each separated by a single white space, and "f" indicates that the next row contains a sequence of fractions, where pairs of integers represent the numerator and denominator of each fractional value, and all integers are once again separated by a single space.

For each test case, your driver program must use the Standard Template library to construct a doubly-linked list from the input sequence, then pass the list as a reference parameter to your sort function, and finally print out the sequence after sorting. Similar to the input format, the output file should contain two rows per test case, the first indicating the problem type (i.e., "i" or "f"), and the second row giving the sequence of values from the sorted list. Include the output method given with the `Fraction` class in your program so the fractional values will print with "/" between the numerator and denominator to make it easier to see that the results have actually been sorted correctly.

3 The Recursive Mergesort Function

You must write your own `mergesort` function. **Do not use the sort() member function from the STL list.** Your `mergesort` function should have two parameters:

- An *STL iterator*, pointing to the left-most node in section of the linked list that must be sorted by this specific function call, and
- An *integer*, giving the total number of nodes in the chosen section of the linked list.

For example, if you started with the following list of numbers `2 8 10 3`, then your main program would make an initial call to `mergesort` with an iterator pointing to the node storing the value `2` and a count of `4`. This initial call would then make two recursive calls with iterators pointing to the nodes storing values `2` and `10` and counts of `2`.

The second parameter provides a hint to your function that saves a little time but does not change the fundamental complexity of the algorithm. Without a node count parameter , the initial call to your function would need to step through the entire list to count the nodes (an $O(N)$ operation). In either case, it would still need to start from the beginning and step half-way through the list to find the center (another $O(N)$ operation) to split the list in preparation for the two recursive calls. However, since merging the two sorted half-lists generated by those two recursive calls to `mergesort` is an $O(N)$ operation anyway, the time spent stepping through the list to perform the initial node count does not affect the overall complexity of the algorithm.

4 The Merge Function

Whenever two recursive calls to `mergesort` applied to adjacent sections from the linked list return, the parent function must call `merge` to combine the two individually-sorted sections they produce into one completely-sorted section. Thus, for compatibility with the `mergesort` function defined above, your `merge` function should have the following four parameters:

- An *STL iterator*, pointing to the left-most node in the left section of the linked list,
- An *integer*, giving the total number of nodes in the left section,
- An *STL iterator*, pointing to the left-most node in the right section of the linked list, and
- An *integer*, giving the total number of nodes in the right section.

The value returned will be an STL iterator pointing to the leftmost object in the combined list section. There is no need to return the size of this combined list section since the caller already knows its value.

The task of your `merge` function is to move objects, one-at-a-time in non-decreasing order, from the two sorted sub-sections of the list directly to their correct location in the combined list section. When it is finished, you should be left with one STL iterator pointing to the left-most node in the combined section. The count of the total number of nodes in that combined list section should already be available as an input parameter to the copy of `mergesort` that made this call to the `merge` function, or by adding together the sizes of the two sub-sections of the linked list in its parameter list.

Your code will carry out these object moves using the STL list's `erase` and `insert` methods where:

- `iterator erase (const_iterator position);` deletes the node pointed to by this iterator. The value returned is an iterator to the node *after* the one you just deleted, or to the *end-of-list* if you deleted the last node. **Be sure to obtain a pointer to the object being moved away from this location before erasing its container!**
- `iterator insert (const_iterator position, const value_type& val);` inserts a new node into the list *before* the node at `position`. **The second parameter is a pointer to the object being relocated to this location.** The value returned is an iterator that points to the newly-inserted node.

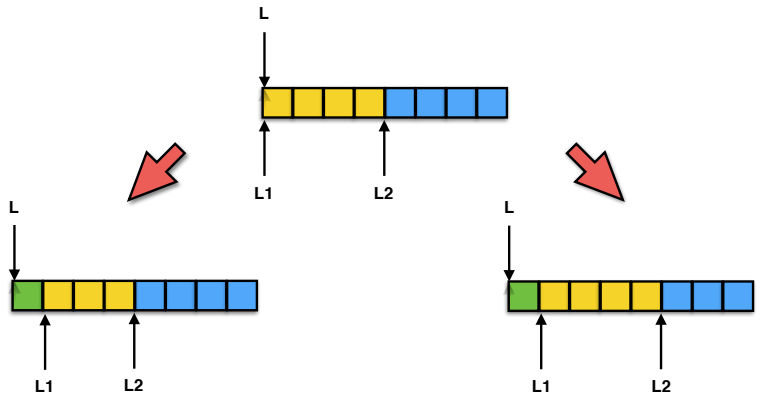

Figure 1: List updates during `merge` function. Starting from center top, we have left sub-segment starting from `L1` shown in yellow, and right sub-segment starting from `L2` shown in blue. Merged output starts from `L` and includes nodes shown in green, when they exist. Each time the left sub-segment has the smallest element, we simply re-assign it to the output segment by incrementing `L1` and changing its color from yellow to green (see left arrow). Each time the right sub-segment has the smallest element, we must erase the node from the front of the right sub-segment, and insert its value into a new node added to the left of `L1` (see right arrow). In this case, the output segment expands to the right by one (green) node, while `L1`, `L2` and the entire left sub-segment shift one step to the right, and the right sub-segment loses its first (blue) node.

Your code should follow the same basic principles for merging two vectors that are stored in vectors, instead of linked lists. In other words, with reference to Figure 1 there must be one pointer `L` to the start of the combined list section, two pointers `L1` and `L2` to the smallest remaining node (if any) in each of the two subsection, plus counters indicating the number of objects currently in each section of the list. Initially, we set `L=L1`, since the output section should cover both of the input sections. However, we might need to move `L` one step towards the left if `L2` happens to contain the smallest value overall, which therefore must be come *before* `L1` in the output section.

At each iteration, the smallest object is moved from one the two input-list sections to the output list section. This next-smallest object must be inserted after all the objects already added to output list section, which is between `L1-1` and `L1`. Thus, at every iteration where the left sub-section holds the smallest value, we simply increment the pointer `L1` to indicate that the first element from the left input section has been reassigned to the output list section. (These changes are illustrated by the left arrow in Figure 1.) Conversely, at every iteration where the right sub-section holds the smallest value, we must remove the first element from the right input section and insert it in front of `L1`, which marks the start of the left sub-section. (These changes are illustrated by the right arrow in Figure 1.)

Notice that in the first case, the object's initial position and final position are the same, so the pointer `L1` changes but the linked list remains exactly the same. Your code must be able to identify this situation, so it can skip the `erase` and `insert` steps in this case.