Project 2: List::Sort

Implementing List::Sort as an in-place, stable, O(n log n) sort.

Version 08/28/18

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Fully implement complex pointer-manipulation algorithms
- Describe in pseudocode top-down and bottom-up merge sort
- Fully test new sort implementatons for runtime using drop-in comparison counters

Background Knowledge Required: Be sure that you have mastered the following material before beginning the assignment:

1. Lectures on Sorts.
2. Lectures Generic Set Algoithms (particularly `g_set_merge`).
3. Details of implementing `fsu::List` found in `LIB/tcpp/list*`.

Operational Objectives: Implement the two versions of List::Sort as member functions of List. List::Sort must be in-place and stable and have worst case runtime O(*n* log *n*), where *n* is the size of the list.

Deliverables: Two files:

```
list_sort.cpp     # slave file of list.h
makefile          # builds all your tests
log.txt           # your project work log and test diary
```

Bottom-Up MergeSort

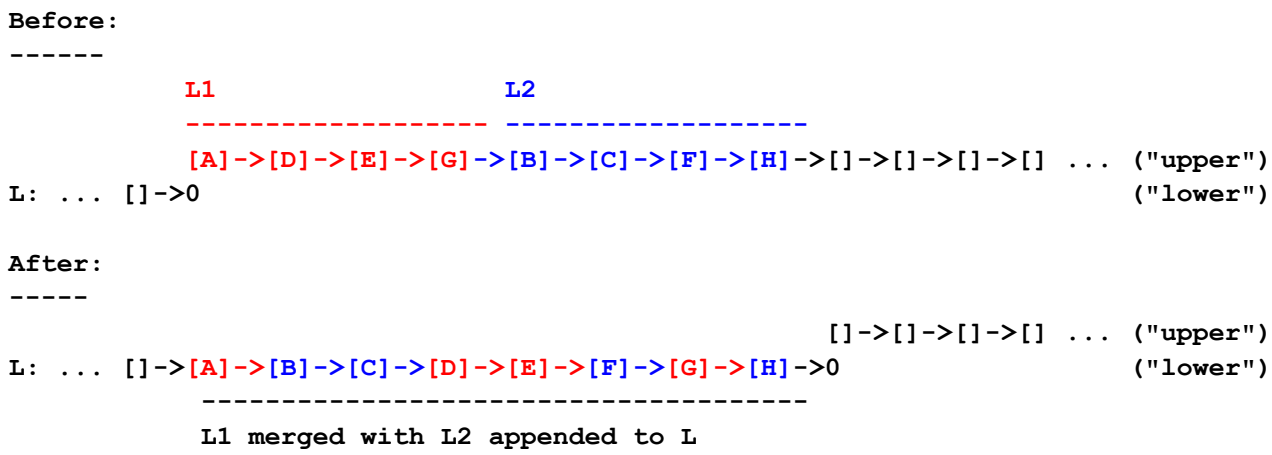
There three instances of the bottom-up merge sort algorithm discussed in the lecture notes. All versions of the algorithm require a "merge" utility. In most cases, a target space to hold the merge of two subsets is temporarily required because data cannot be mapped directly back without potentially over-writing other data. The situation for linked lists is better: by unlinking and re-linking data nodes (instead of copying the data inside the nodes) the need for temporary extra space goes away. Here is pseudo-code for a merge operation for subranges in a linked list.

Let L1, L2 be the "input" subranges in the carrier of a linked list and L the "output". The idea is to move entire nodes from the inputs to the output.

```
initialize pointers p1 to the first node in L1 and p2 to the first node in L2
while (L1 and L2 are not empty)
{
  if (p2->t < p1->t)
  {
    q = p2;
    p2 = p2->next; // "advance" p2
    unlink *q from L2 and append it to L
  }
```

```
  else
  {
    q = p1;
    p1 = p1->next; // "advance" p1
    unlink *q from L1 and append it to L
  }
}
if (L1 is not empty) append L1 to L;
if (L2 is not empty) append L2 to L;
// at most one of the last two statements will activate
```

(The reason for promoting L2 before L1 is that when the values are equal L1 takes precedence over L2.) Here is an illustration:



This is at a stage when you are merging subranges of length 4. You are essentially migrating the links from the top to the bottom. The upper collection gets shorter and the lower collection gets longer as you move along, until at the end all links have migrated from upper to lower picture, and you have completed the step "merge successive pairs of subranges of length 4".

This version of merge facilitates bottom-up merge sort on a linked list as follows:

```
Loop 1: merge all n/2 successive pairs of subranges of length 1
Loop 2: merge all n/4 successive pairs of subranges of length 2
Loop 3: merge all n/8 successive pairs of subranges of length 4
...
Loop k: merge all n/2^k pairs of subranges of length 2^(k-1)
...
```

Proceed until only one sublist remains ($n \leq 2^k$) - which is sorted in place.

There are some loose ends to tie up:

1. At the end of the kth merge loop, there are 3 cases to deal with:
    i. The left-hand subrange is shorter than $n/2^{k-1}$ (and hence the right-hand subrange is empty)
    ii. The right-hand subrange is shorter than $n/2^{k-1}$

iii. Both left- and right-hand subranges are full length

2. Careful counting during the inner loop is necessary (taking time to count separately would increase runtime). Count as you advance pointers, and be sure you never retreat pointers and that you never advance the same pointer twice. That will ensure that the total number of pointer advances in the inner loop is at most n. (A "pointer advance" is a code snippet "`ptr = ptr->next_`" for some link pointer `ptr`.) Because the outer loop runs at most log n times, this will ensure that the total runtime is O(n log n).
3. A control mechanism for the outer loop needs to be invented - "do{with break statements} while(1);" has seen success.
4. The head and tail nodes should be excluded from the sort process.
5. The "unlink/re-link" process only needs to deal with forward ("next") pointers. Then one traversal can be used to repair all of the backward ("prev") pointers.
6. After the whole list is sorted and has its local pointers all re-established, put the head and tail nodes back in place.

Procedural Requirements

1. The official development/testing/assessment environment is specified in the Course Organizer. Code should compile without warnings or errors.

2. In order not to confuse the submit system, create and work within a separate subdirectory `cop4531/proj2`.

3. Begin by copying the entire contents of the directory `LIB/proj2` into your `cop4531/proj2/` directory. Then copy the file `LIB/tcpp/list_sort.cpp` into `cop4531/proj2/`. At this point you should see these files in your directory:

```
list_sort.cpp
sortspy.cpp
deliverables.sh
```

4. Test your sort thoroughly, including collecting and analysing comparision-count data, until you have strong computational evidence that the implementation (1) is correct and (2) runs in time O(*n* log *n*).

5. Be sure that your `log.txt` contains test results and convincing arguments.

6. Submit the assignment using the standard submit procedure.

Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit assignments. If you do not receive the second confirmation with the contents of your assignment, there has been a malfunction.*

Code Requirements and Specifications

1. The file `list_sort.cpp` that you copied from LIB/tcpp is the starting point for your code. The deliverable has the same name as this file ... so be careful not to re-copy the file in LIB/tcpp over your work.

2. The sort algorithm implemented in LIB/tcpp/list_sort.cpp is InsertionSort. You are required to

replace that implementation the bottom-up Merge Sort algorithm.

3. Use a drop-in comp-counting predicate LessThanSpy<T>, as seen in previous courses, to collect data from your List::Sort on data sets ranging in size from small (10) to large (1,000,000). Use these data and clock-time data to provide evidence of the runtime of your implementation.

Hints

- Most of the sorts tested by sortspy.cpp are in LIB - the exception being the one you are developing.
- Attempting to use calls to List::Merge may be problematic, because the subranges being merged are not List objects.