CSCI 132 Data Structures—Lab #8

Introduction

In today's lab we will work with algorithmic efficiency and sorting, and get more practice using pointers and linked chains. Copy the code for the lab:

```
cp -ri ~csci132/labs/lab8 labs/ && cd labs/lab8/
```

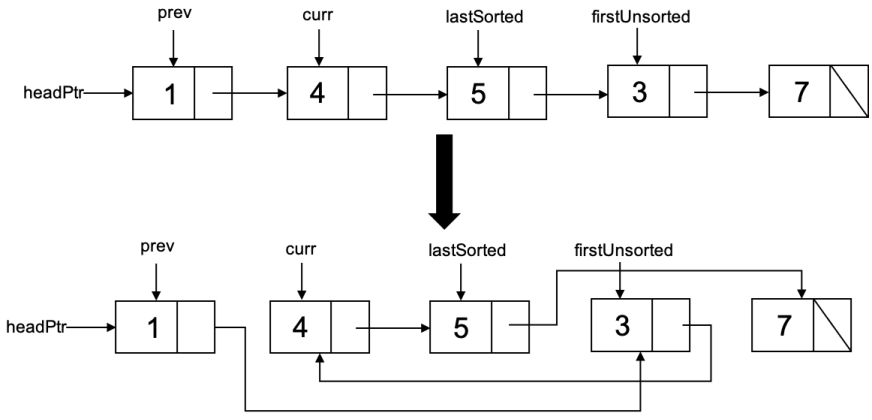
Note that there is no autograder for today's lab, because the testing file tests for correctness for you.

Insertion Sort for Linked Lists

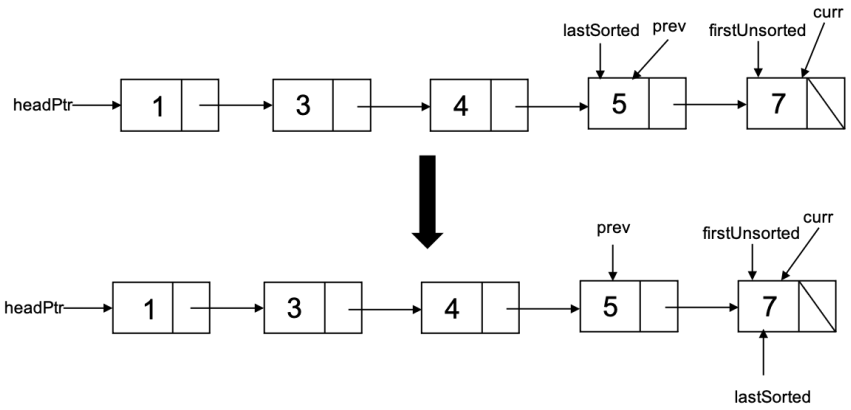
We studied the insertion sort algorithm in class, using an array of integers. Your task is to complete the implementation of insertion sort for a linked chain of `Node<ItemType>` objects. Starter code is provided in `InsertionSort.cpp`. And a test program provided in `SortingLab.cpp` will test your insertion sort implementation.

Performing insertion sort on a linked chain is conceptually no different than on an array. However, since we cannot iterate through a linked chain of nodes in reverse, we will search for the location to insert into the sorted sublist starting at the the beginning of the list. The algorithm works as follows for a linked chain of nodes pointed to by a pointer `headPtr`.

1. Initialize a pointer, `lastSorted` to `headPtr`. This pointer will keep track of the end of the sorted sublist. If `lastSorted` is `nullptr`, we're done.
2. While `lastSorted` is not the last node in the linked chain, set another pointer, `firstUnsorted` to the next node in the list.
   (a) Check if the node pointed to by `firstUnsorted` should be inserted into the beginning of the sorted sublist, and if so, update the pointers to insert this node at the beginning of the list.
   (b) Otherwise, use two pointers, `prev` and `curr` to iterate through the sorted sublist starting at `headPtr` until the location to insert the node pointed to by `firstUnsorted` has been found (i.e., `firstUnsorted->getItem()` is less than or equal to `curr->getItem()`). And then make the necessary updates to the pointers to insert this node in its correct location in the sorted sublist.
   Note that there are two cases to handle here.
      i. Either the node will be inserted between the nodes pointed to by `prev` and `curr` (shown below)



ii. *or* if `firstUnsorted == curr` then the node is inserted at the end of the sorted sublist (shown below).



Checkpoint 1:
The setup for this algorithm as well as Cases 1 and 2a have already been implemented for you. Your task is to complete the implementation of Case 2b above.

After completing your implementation of Case 2b, test your code by compiling the `sorting` program using the commands found in `Makefile`, or type `make sorting`. Test using `./sorting 10` and various other larger array sizes.

The runtime for our linked insertion sort should be $O(n^2)$ except for the case of an array in reverse sorted order. Notice that this is the opposite of the best case for the insertion sort for arrays that we saw in class since we are iterating through our sorted sublist from smallest to largest to find the location to insert the next entry (instead of iterating through the sorted sublist from largest to smallest).

Note: The test program for this lab does not measure the actual "wall clock" time to run the `insertionSort` function. Instead, it uses a special `Record` item type as the data to be sorted. This `Record` type keeps track of how many comparisons and copies are made, so that exact statistics can be printed. The Big-O efficiency analysis applies to these operations just as it does to "wall clock" time. The runtime for insertion sort should be $O(n^2)$ in the worst case, and $O(n)$ in the best case. Verify your implementation's efficiency by trying several chain lengths.

Checkpoint 2:
`InsertionSort.cpp` **already contains an implementation of Insertion Sort for Array Lists for you. In** `SortingLab.cpp`**'s main function, uncomment the corresponding line to test for this implementation too. Look at the number of comparisons and copies done by Insertion Sort for both the Linked Chain and ArrayList implementations. How do the best cases and worst-cases differ? Discuss in** `lab8.txt`**.**

Merge Sort for Array Lists

Merge sort is a classic example of a divide-and-conquer algorithm, similar to the approach we used in class for finding the maximum value in an array. In that problem, we divided the array into two halves, recursively computed the maximum value for each half, and then combined the results by taking the larger of the two. Merge sort follows a similar strategy for sorting an array.

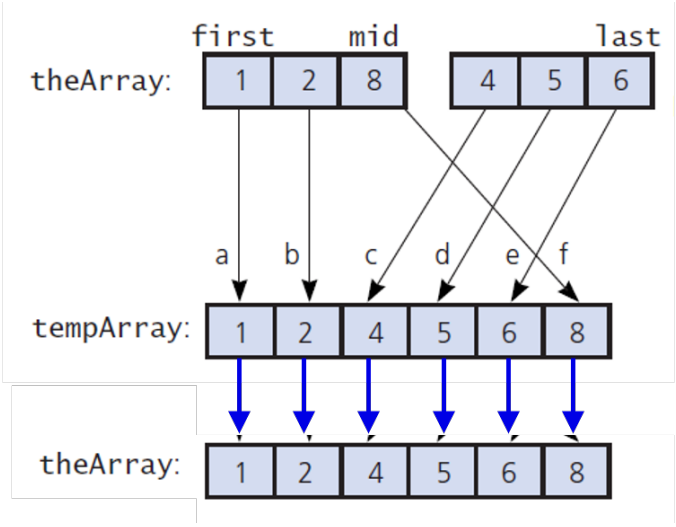
The algorithm works by recursively dividing an ArrayList into two smaller sublists until each sublist contains only a single element (which is inherently sorted). Then, in the merging phase, the sorted sublists are combined in a way that preserves order, ultimately reconstructing a fully sorted array. This merging

step is where the actual work happens and is the function you will implement. In `MergeSort.cpp`, the `mergeSort` function is implemented for you, and you have to finish implementation for the `merge` function.

To efficiently merge two sorted sublists, we use a two-finger approach that processes both lists in a single pass. This should remind you of the two-finger approach of finding common elements in two sorted arrays, that you used in Lab 7. This technique avoids unnecessary comparisons and ensures that merging runs in $O(n)$ time. The approach works as follows:

- Use two indices, one for each temporary sublist, initialized at their respective starts.
- Compare the elements at these indices, selecting the smaller one and adding it into the temporary list. Move the corresponding index forward.
- Continue until one sublist is fully processed.
- Copy any remaining elements from the other sublist back into the temporary list.
- Copy over items from the now-sorted temporary list to appropriate positions in the original list.

An example of the merge process is shown below.



Once you are confident in your implementation of `merge`, uncomment the line of code in `SortingLab.cpp` to test Merge Sort for ArrayLists. Test correctness and efficiency of your code for various array sizes. Then finally, compare the number of operations for Merge Sort and Insertion Sort for the same array size. Is one significantly more efficient than the other? **Discuss in `lab8.txt`.**

What to Turn In

Submit your source code and your `lab8.txt` file using the following command:

```
~csci132/bin/submit
```

Be sure you add your name and date to the prologue for each file you modify.