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/
```

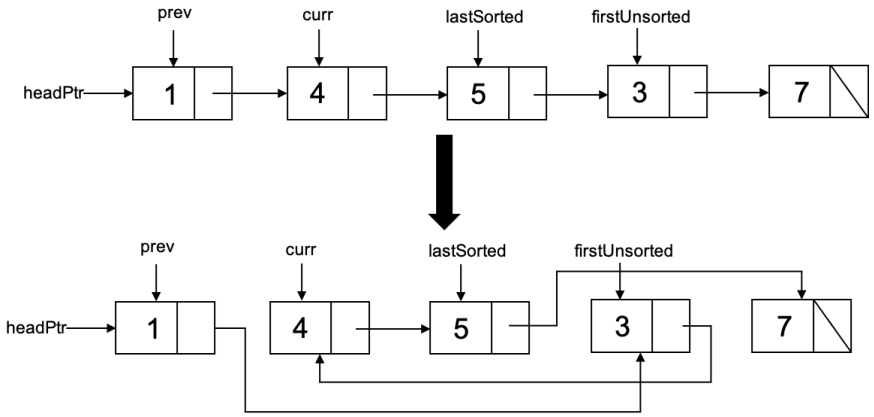
Note: Before we start, note that only the last part of the lab (Counting Inversions) will be auto-graded today. The rest is about completing sorting functions, which you should test by yourself for correctness. Once you're sure of correctness, then move on to the analysis of run-times. Your code will be graded manually to make sure the algorithm is correct.

Insertion Sort for Linked Lists

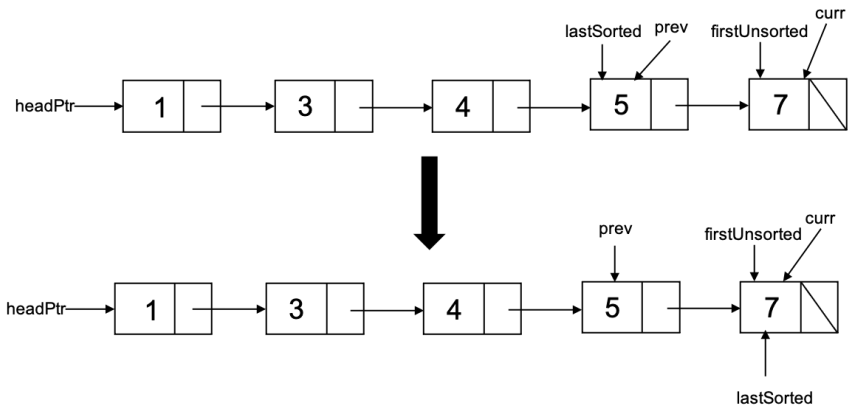
We studied the insertion sort algorithm in class, using an array of integers. Your task is to complete the implementation of insertion sort for a linked chain of `Node<ItemType>` objects. Starter code is provided in `InsertionSort.cpp`. And a test program provided in `SortingLab.cpp` will test your insertion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



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.

To test that your implementation of Case 2b is correct, test your code by compiling the `check_sort` program using the commands found in `Makefile`, or type `make check_sort`. `check_sort` takes the length of a node chain as input, geneates a random node chain and tests your sorting code.

After checking the correctness of your implementation, test the run-time of 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. Using `./sorting n` tests your code for different types of node chains (random, already-ordered, and reverse-ordered). It generates 100 trials and then reports the average number of comparisons and copies.

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). Test using `./sorting` for different node chain lengths and report the expected number of operations for different lengths.

- Report in `lab8.txt` which type of node chain runs the fastest (has the smallest number of comparisons and copies).

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.

Find Pairs, Revisited

In a previous lab we solved the problem of finding a closest pair of integers in a list of numbers. The file `Pairs.cpp` contains an implementation of a $O(n^2)$ algorithm to find a closest set of pairs in a given `ArrayList`. Compile this as the `pairs` program using the commands in `Makefile`, or run `make pairs`. Then run the program with a large list, e.g. `./pairs 50000`. This should remind you again of just how slow $O(n^2)$ can be in practice.

- Copy `Pairs.cpp` as a new program, `FastPairs.cpp`, then modify `Fastpairs.cpp` so it is more efficient. Your task is to write a $O(n \log n)$ algorithm to solve the closest-pair-finding problem.

  You will find it useful to first sort the given list `nums` by calling the mergesort function provided with this lab. You can include this in your file by adding `#include "MergeSort.h"` to the top of your `FastPairs.cpp`. Note that the `mergeSort` function expects a pointer to a list as a parameter. You can create a copy of the list in the following way to then call mergeSort.

  ```
  ArrayList<int> *list = new ArrayList<int>(nums);
  ```

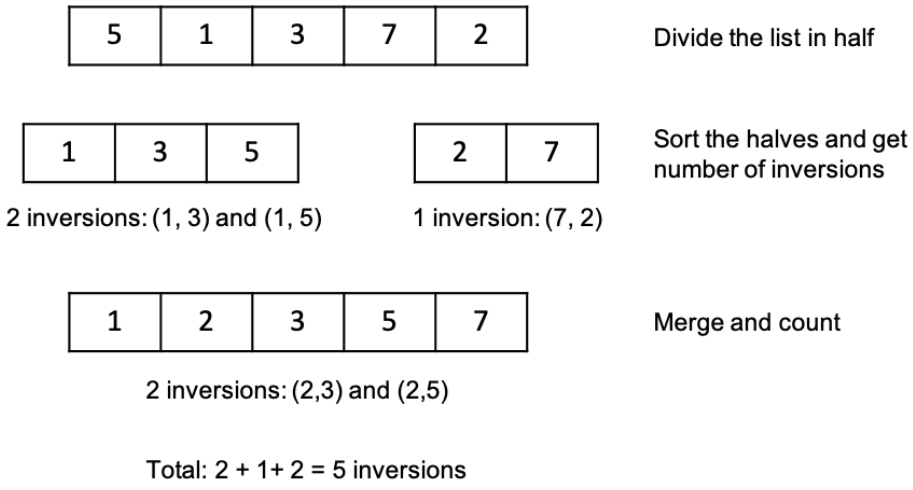
- Briefly explain your approach and why your solution has a $O(n \log n)$ running time in your `lab8.txt` file.

Compile and test your new program, and verify that it really is orders-of-magnitude more efficient than the original implementation. The `Makefile` has the commands needed to compile the program, or type `make fastpairs`, then run using `./fastpairs 50000` or other sizes.

Counting Inversions in a List

We can measure how close a list is to being in order by counting the number of inversions. An inversion is a pair of indices `i` and `j` where `i < j` and `list[i] > list[j]`. For example, the list `[1, 4, 3, 2]` has three inversions since `4` and `3`, `4` and `2`, and `3` and `2` are each out of order.

`CountInversions.cpp` contains an implementation of a $O(n^2)$ algorithm, `slowCountInversions()`, which finds the number of inversions in a given `ArrayList` of integers. However, we can solve this problem with a *faster* algorithm by modifying the algorithm for merge sort in the following way. We split our given list in half, recursively call our function to compute the number of inversions and sort the first half, recursively call our function to compute the number of inversions and sort the second half, and then compute the inversions when merging the two sorted sublists together. (See the figure below.)



We have provided the implementation of the function `fastCountInversions()` which performs the recursive calls on each half of the list and calls a function to merge the two sorted sublists. Your task is to complete the implementation of `mergeCount`, (which initially contains an implementation of the merge function discussed in lecture), so that it computes and returns the number of inversions made when merging the two given sublists.

Hint 1: Think about the example above. How to count inversions made when merging? Consider all elements in the left half, as shown in the image.

- 1: 1 is smaller than all elements in the right half. So, no inversions counted during merging.
- 3: 3 is larger than the element 2 in the right half. So, one inversion.
- 5: 5 is larger than the element 2 in the right half. So, one more inversions.

Hint 2: This might be useful in making sure your mergeAndCount function runs in O(n). How many elements are there in an array between index i and j? For example, if `i = 3, j = 5`, there are 3 elements, `array[3], array[4], array[5]`. If `i = 7, j = 20`, there are 14 elements. In general, there are `(j-i+1)` elements between `array[i]` and `array[j]` including `array[i]` and `array[j]`.

Test your code by compiling the program using the commands found in `Makefile`, or type `make count`, then run using: `./count` You can run the problem with a single commandline argument (`slow/fast`) and then give a list of distinct numbers (see Example 1) or run it with two commandline arguments (`slow/fast`) and a given list size (`n`), which tests the program with a list of the values `1` to `n` in a random order (see Example 2).

- Briefly explain in `lab8.txt` why the run-time of the new faster algorithm is $O(nlgn)$.
- Also, run `./count` with both `slow/fast` options for the following `n` values and report the run-time of both algorithms in `lab8.txt`:

  $1000, 2000, \ldots, 10000$

Sample Output

Example 1:

```
./count slow
Enter a number of values greater than 0: 5
Enter a space-delimited sequence of 5 distinct integers: 2 3 1 7 4
Finding the number of inversions...

Number of inversions: 3

Elapsed time for length 5: about 0.000019 seconds
```

Example 2:

```
./count fast 10
Initializing a list of the numbers 1 to 10 in random order...
list = [ 7, 1, 4, 6, 8, 9, 5, 2, 3, 10 ]
Finding the number of inversions...
Number of inversions: 19

Elapsed time for length 10: about 0.000006 seconds
```

What to Turn In

Submit your source code using the following command:

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

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