PROGRAM TWO / CSC 1310

DOUBLY LINKED LIST

IMPORTANT DATES

Assignment Date: Tuesday, September 26, 2023 in class
Due Date: Thursday, October 12, 2023

DESCRIPTION OF PROGRAM – WHAT DOES THIS PROGRAM DO?

This program is a doubly-linked list that will read a list of counties in the US from a csv file and add each to a linked list, then the linked list will use merge sort to sort the counties by population.

FILES YOU WILL SUBMIT IN YOUR ZIP FILE

This program contains multiple files as described below:

- **List.h** – header file for a doubly-linked list, you will need to implement this entire class yourself.
- **Driver.cpp** – **given to you** – the driver will read the county list from the csv, add them all to the linked list, sort the list, and then print the sorted list. You will need to write everything that interacts with your list.
- **County.h** – **given to you** – header file for a county object. Everything here is written for you.
- **County.cpp** – **given to you** – source file for the functions in county. Everything here is written for you.
- **Counties_list.csv** – **given to you** – a complete list of every county in the US and their populations.
- **counties_ten.csv** – **given to you** – a list of the first ten counties on the previous list. Used for debugging.
- **Reflection.pdf** – write a short reflection on your work for this assignment.

SPECIFICATIONS

DOCUMENTATION

Documentation is part of your grade on all four of the major programs. Your driver.cpp file should have comments at the top that include your name, the course and section number, the due date, and the name of your partner if you're working in pairs. **If you do not include your partner's name your assignment may be flagged for cheating**.

Every class and function should have a high-level description of the purpose of that class or function, and there should be inline comments throughout the body of your program to explain how your program is operating. **You will have points taken off for not commenting your code**.

DRIVER.CPP

All the input from the csv files is done for you. You will need to create a list of county pointers, call the append function to add every new county to the list, call merge sort, and print the sorted list. This makes a total of four lines you'll need to add.

There are two lines written to open a file, one for each file. This way you can comment one out and leave the one you want to read from, this is to make your debugging easier.

LIST CLASS

Your list class must be a doubly-linked list, and it **must be a template class**. You'll notice that some of the functions appear twice on this document, that's because they are overloaded functions and have different uses. Remember that overloaded functions have different parameters/return types.

When you use an operator, since T is an object pointer, if you try to compare T it's going to compare pointers. That means you need to dereference the object. Normally you could do this with *value, but since you're accessing it through a node reference you'll need to use *(node->value).

In the functions that return a node, you'll need to use **`typename List<T>::listNode*`** as the return type.

ATTRIBUTES

- **listNode** – a struct to define a node. It should contain a **value** of type T, and two pointers to another node, **next** and **prev**.
- **head** – a reference to the first node in the list.
- **tail** – a reference to the last node in the list.

PUBLIC FUNCTIONS

Function name: **List** constructor

- Parameters: no parameters
- Purpose: creates an empty list by setting head and tail to null

Function name: **~List** destructor

- Purpose: called when the list is removed from memory
- Specifications: deletes every node in the list by traversing through each node and deleting them as it goes along

Function name: **print**

- Parameters: no parameters
- Returns: void
- Purpose: prints the value of every element in the list
- Specifications: calls the other print function, which takes the starting node as a parameter, and passes head as an argument

Function name: **append**

- Parameters: the new value we are appending to the end
- Returns: void
- Purpose: create a new node and add that node as the last element of the list
- Specifications: this function begins by dynamically allocating a new node, and filling that node with the value we got as a parameter. If the list is empty it sets head and tail to the new node. If the list is not empty it adds the node node after the tail node and moves the tail pointer.

Function name: **mergesort**

- Parameters: no parameters
- Returns: void
- Purpose: this is the public-facing function call to begin sorting the list in **descending** order.
- Specifications: this function calls to other merge sort function, passing head and tail as arguments. Resetting head and tail is tricky to do recursively, so when mergesort is finished this function sets the new head and tail reference. The other merge sort function returns the head of it's merged list partitions. Since the last iteration returns the front, that node needs to be our new head. From the head we can traverse the list to find where the new tail is.

PRIVATE FUNCTIONS

Function name: **print**

- Parameters: a node pointer
- Returns: void
- Purpose: this version of print will print every value in the list after the parameter node. **This is useful for debugging your mergesort**.
- Specifications: this function will traverse through every node in the list, starting at the node sent as the parameter. It will cout the current node at every step in the list traversal.

Function name: **mergesort**

- Parameters: two node pointers, the first and last node of the list partition
- Returns: a reference to a node, the first in the list partition that has been merged
- Purpose: manages the recursive calls that mergesort needs to make
- Specifications: the base case is when the list partition is either zero or one node, in which case the function will return the first parameter (which is either the only element or null). In the recursive case, you will need to make two node references for a placeholder, which will be the first element of each list partition. The first of these two will be the first parameter for the function, and the second will be the node that's returned by the split function. Then both of the recursive calls are made, passing the lower and upper bound of both list partitions, and sending the node that's returned to our two placeholder nodes. Finally, the function calls merge and returns the node that's passed from the merge function.

Function name: **merge**

- Parameters: two node pointers, the first node in the two list partitions being merged.
- Returns: a reference to a node, the first in the list partition that has been merged
- Purpose: merge two list partitions together
- Specifications: The approach to a merge function we covered in class used loops, but you should use recursion here because it will save you some headache. To start, if either of our parameters are null that means we've exhausted one of the two partitions, so we should return whatever node reference is remaining. We then compare the two parameter nodes to see which should be merged next. We follow the same set of steps in both cases, but applied to whichever node has the larger value. The next pointer for the larger node should be the return value of a recursive call to merge, the argument should pass the next node of the list partition. The previous pointer for that node should be set to the selected node. The selected node's previous pointer should be set to null. Then return the selected node.

Function name: **split**

- Parameters: two node pointers, the first and last node in the list partition being split
- Returns: a node reference to the middle node
- Purpose: subdivide a list partition into two more partitions
- Specifications: There are a handful of valid ways to find the middle node. The strategy you should take for this function is to do a double traversal, one from each end, and they will meet at the middle node. This means that you'll have a reference to the middle node that you ca return, **but first** you must set the next pointer for that node to null before you return it. This will sever the two partitions from each other, which you need for checking your base case. Once you have severed these nodes, return the first node after where the list was severed (the node that would otherwise have been pointed to by the middle node).

EXTRA CREDIT OPPORTUNITY – SELECTION SORT

You have the opportunity to get some bonus points on this program by implementing selection sort and a helper function for it to sort successfully. Make sure merge sort works correctly before implementing selection sort (merge sort is worth more points). This section is worth up to ten points.

Function name: **selectionSort**

- Parameters: no parameters
- Returns: void
- Purpose: sorts the list in **ascending** order using selection sort.
- Specifications: Selection sort will follow the same process as what we've covered in class, but will do node traversals in the two loops instead of incrementing a counter. When it swaps the two designated nodes, it does so by calling the swap function (passing min as the second argument, since it will come later in the list).

Function name: **swap**

- Parameters: two node pointers, the two nodes being swapped.
- Returns: void
- Purpose: swaps the position of two nodes in the list
- Specifications: For the sake of simplicity you should assume that the first parameter node appears before the second in the list. Like swapping the position of two variables, you will need a placeholder variable. You'll need to figure out which node links need to move to where and in what order. There is a slightly different process if the nodes are neighbors than if they have nodes between them.