Samantha Lee (sll155), Matthew Notaro (myn7)

Professor Francisco

Systems Programing

1 March 2020

# Assignment 0

Description:

FileSort.c is a program that accepts a text file as one of three command-line arguments, detects the type of data it stores, and lexicographically sorts its items. It can implement either insertion sort or quicksort using a doubly linked list built from the file's tokens.

How to use:

The user must enter a command-line prompt that follows the following format:

*<executable> <type of sort> <text file>*

The type of sort can be quicksort (-q) or insertion sort (-i). An example prompt is:

*./a.out -q ./exampleFile.txt*

The text file can contain values of type int or of type string (char), but not both. Data items must be separated by commas. After tokenizing the data, the program stores the items into a linked list, sorts it based on the user's selected algorithm, and prints the resulting list.

Implementation:

- typedef struct Node{ void* data; struct Node* prev; struct Node* next; } Node;

  Node is the struct used to implement a doubly linked list with a void pointer as the data field.

- main:

  In main(), the type of sorting algorithm and file name are extracted from the input. The function readFromFile() reads the file into a string called file_string, and extractAndBuild extracts the individual tokens from the string, inserts them into a linked list, and determines the type of data in the file. Next, depending on the algorithm chosen, either insertionSort() or quickSort(). Both functions take in a void pointer and comparator as parameters; in our case, the void pointer is casted to a Node pointer and the comparator is either integer-specific or character-specific, depending on the data type.

- char* readFromFile(char* file);

  readFromFile() opens the text file and determines its size via creating a stat struct and accessing its st_size element. Then, the entirety of the file is read into a string called file_buffer and is returned.

- Node* extractAndBuild(char* file_string);

  extractAndBuild() iterates through each character of the given string to find the starts (beginning of file for first token, one char after last comma for all others) and ends (indicated by commas) of the tokens. For each token, a loop examines

each of its characters to parse out all white space before inserting it into a linked list using insert(). The linked list is returned.

- Node* insert(void* token, Node* head);

  insert() creates a node with the given void* token, prepends it to the given doubly linked list pointer, and returns the resulting doubly linked list.

- int insertionSort(void* head, int (*comparator)(void*, void*));

  insertionSort() sorts a doubly linked list through the given pointer to the head using the given comparator function pointer. insertionSort() maintains a sorted portion of the linked list, starting from the front, and incorporates new elements into the sorted portion by comparing each element to the previous ones. If the previous element is greater than the new element, then the previous element shifts over to the right by one, and then the previous element to that is then compared until an element that is less than the desired one is found.

- int quickSort(void* head, int (*comparator)(void*, void*));

  quickSort() takes in an item to sort (in our case, a linked list) and a function pointer to a comparator. It determines the front and rear of the given linked list and calls recursiveQuickSort, passing the front, rear, and comparator. The recursive function returns the front of the sorted linked list, and quickSort() prints the result into STDOUT.

- Node* recursiveQuickSort(Node* left, Node* right, int (*comparator)(void*, void*));

recursiveQuickSort() partitions the list by calling partition(), which sorts the first Node of the list (left) and returns it in its correct position. From there, the function recurses on the remainders of the list to the left and right of the node.

- Node* partition(Node* left, Node* right, int (*comparator)(void*, void*));

  partition() takes in the leftmost and rightmost nodes (called left and right, respectively) of the part of the linked list to be sorted, as well as a comparator. The pivot (node to be sorted) and the Node pointer fromLeft are set to point to left, the Node pointer fromRight to right. A do-while loop iterates for as long as the pointers do not "cross," and, within it, two for loops determine the first item from the left that is greater than the pivot (using the comparator) and the first item from the right that is less than the pivot — fromLeft and fromRight are updated accordingly. Still in the do-while loop, the function swaps fromLeft's data with fromRight's since the pointers have not yet crossed. Once the pointers do cross, the program exits the loop and swaps the pivot's data with that of fromLeft->prev, effectively putting the pivot in its correct location. The Node containing the pivot data (fromLeft->prev) is returned.

- int stringCmp(void* thing1, void* thing2);

  stringCmp() takes two void pointers (thing1, thing2) as parameters and casts them as character pointers. It compares the strings character by character for as long as they are lexicographically the same, and then returns the numerical difference between them. If the difference is less than 0, thing1 is less than thing2; if the

difference is greater than 0, thing1 is greater than thing2; if the difference is 0, the strings are equal.

- int intCmp(void* thing1, void* thing2);

  intCmp() takes two void pointers (thing1, thing2) as parameters, casts them as character pointers, and uses atoi() to convert them to integers. It then returns the difference between them. If the difference is less than 0, thing1 is less than thing2; if the difference is greater than 0, thing1 is greater than thing2; if the difference is 0, the strings are equal.

- void printLL(Node* head);

  printLL() prints the given linked list into STDOUT by traversing its length and casting the data of each Node depending on the values in the original file.

- void freeLL(Node* head);

  freeLL() traverses the linked list, freeing each node's data and then the node itself until the whole list is freed.