CS 214 : Systems Programming
Assignment 0 : File Sorter

Authors :

Jonathan Konopka

NETID : jk1549

RUID : 178005220

Nicholas Rytelewski

NETID : nr548

RUID : 197000952

Project Description :

The goal of this project was to tokenize the contents of a file given as an input parameter and sort the tokens in either alphabetical or numerical order. Our program takes in two parameters, a sorting flag (denoted as "-q" to perform quick sort, and "-i" to perform insertion sort), and a file name. A few prerequisites to the input file include the fact that :

- Tokens must be separated by a ',' character
- Whitespace characters are to be ignored
- Null tokens are possible
- String tokens will only contain lowercase letters
- Files will contain either integers or strings, but never both

Our program uses the Linked List data structure to organize tokens.

Functions :

void insert_string(char *str, int stringlength)

insert_string() takes a char* and int parameter. This function is used exclusively when the program is tokenizing *strings*. It creates a new node for the linked list *tempNode* and assigns the input string and the length of that string to it. It will place this new node to the back of the list. This function returns nothing upon completion.

void insert_digit(int input)

insert_digit() takes an int as a parameter, and much like insert_string(), this function is exclusively used when the program is tokenizing *integer values*. This function is identical to insert_string(), however it sets the token value of the node to NULL rather than assigning it a value. Once again, the *tempNode* will be inserted to the back of the present list. This function returns nothing upon completion.

void printlist(int mode)

printlist() is a function that prints out the contents of the existing linked list beginning from the head. The mode parameter is passed and can either be 0 or 1 (for integers and strings, respectively). Depending on what mode is selected, this function will either print each node of the linked lists string or its integer. This function returns nothing upon completion.

void freelist(node* h)

freelist() is a function that takes the head node of the linked list as a parameter. It will head down the contents of the list and perform the free() operation on each node so that the memory can correctly be deallocated. This function is called at the end of main() after sorting has occurred. This function returns nothing upon completion and the program is to exit afterwords.

int intComp(void* thing1, void* thing2)

intComp() takes two void* parameters, thing1 and thing2. In the function, it casts each "thing" into a node. From the node structs, two integers are created that hold the values of thing1->value and thing2-> value. This function is passed into the insertionSort() and quickSort() functions. It returns 1 if thing1's value is greater than thing2's, and returns 1 if thing1's value is less than thing2's value.

int strComp(void* thing1, void* thing2)

strComp(), much like intComp, takes two void* parameters, thing1 and thing2. This function casts each thing into a node and assigns to char* to the token of the nodes. From there, the functions loops until the null terminator '\0' is found, and increments through the characters of the tokens. If both characters at the first position are equal, the function loops again. If thing1's token is greater than thing2's token, the function returns 1. Likewise, if thing1's token is less than thing2's token, the function returns 0. In the case that the two tokens are identical, the function will return with 1.

int insertionSort(void* toSort, int (*comparator)(void*, void*))

The insertion sort function that has the required prototype so that we could perform the insertion sort algorithm on our data. It casts the void* into a node* and uses whatever given comparator to compare elements. If the iterating curr node happens to be greater than or equal to the front of our sorted linked list, then it will be inserted in front with the rest of the list modified to accommodate it. If the curr node happens to be less than the front, then its next node will be the front. It will return 0 on success.

int quickSort(void* toSort, int (*comparator)(void*, void*))

The quickSort() utilizes our given prototype to perform the quicksort algorithm. It takes a void* and function pointer to the comparator as the parameters. It is simply used to cast the void* into a node* and find the last element in that linked list. The starting and ending nodes are used in our two helper functions, partition and _quickSort. It will set the global head to the start after performing quicksort and then return 0 upon success.

void _quickSort(node* start, node* end, int (*comparator)(void*, void*))

This _quickSort() is the recursive element of our quicksort algorithm. It is called in the given quicksort function. The arguments are two node* and the function pointer to a comparator. The function does not return a value but instead sorts the given linked list. When a pivot from the partition is found, it is called to take all elements less than the pivot and items greater than the pivot in two separate calls.

node* partition(node* start, node* end, int (*comparator)(void*, void*))

The partition() function features the same parameters as _quickSort(). Takes the start and end of the linked list and compares it to the pivot element, where items less than the pivot are to its left and the items greater than the pivot are to its right. This is done by looping through the left until the end, then looping backwards from the right. The new pivot to be inserted into the recursive quicksort algorithm is returned at the end of the function. This is found by swapping the pivot with the right node at the end.