EE488A

Homework 1

Michał Gorywoda

20204787

1. Introduction

The program described in the following document is an implementation of a inverted index based search engine written in C with help of most the basic I/O libraries. The program has to browse through all the text files in specified directory and list all the words that have occurred in these documents. The list, along with the file identification number and occurrence line, shall be stored in memory belonging to the program. Next, the user will be prompted for searched word and the program will browse through the record. If searched word was in one of the documents, a line number(s) and the document name(s) will be displayed.

The program demands operations on character arrays and pointers. However, use of string.h library is not allowed. Because of that, a separate library using pointer arithmetic operations was created.

The project was divided into logical source and header file sets: “main”, “searcher”, “record”, “files” and “mystrings”. In order to achieve fast and efficient compilation process, a makefile was created. With just one shell command, an automated compilation process is possible.

Along with the source code, a set of example text documents was provided. These include excerpts from Shakespeare’s “Julius Caesar” play.

2. Program flow

Assuming the project was built correctly, it is possible to run the program using *./search* command with proper argument being the path to the target directory. For instance, in order for the program to browse the example directory, the command would be *./search docs/*

If the program returns directory error, please try make sure the directory is specified according to format above.

Program starts with displaying the chosen path and browses the directory in search for files with .txt extensions. An array is filled with the names of these files. Next, each file is opened and each line is separated into words. Each word is then checked with word record by a linear search. If there is no such word in the record, it is added to a next free cell. The line number, as well as document ID is saved in the same cell number, but in different arrays. If a word already exists in the record, a line number and document ID is added to a next position in corresponding array, and a number indicating the frequency of occurrence is increased by 1.

After browsing through all the text files, the whole record can be displayed, but this can be disabled by removing the display command.

Lastly, the user is prompted to input a searched word. If it is not found in the record, the user will be informed with a text message. By contrast, if by means of linear search the word is found in the record, the program will display information about the line, documents and the place in the record in the following format:

*Word **searched_word** found under index n, occurred x times*

File_name.txt: line #y

3. Function description

a. Main

int main(int argc, char *argv[])

The main program function, contains a high-level overview of the file browsing, indexing and search procedures.

@param *argc* The number of character sequences in *argv* array.

@param **argv[]* A 2D array containing the program name and arguments passed during the start.

@return An error information returned after the program finishes. By default it returns 0.

b. Record

typedef struct{

char wordRecord[RECORD_SIZE][CELL_SIZE];

int lineRecord[RECORD_SIZE][CELL_SIZE];

int docRecord[RECORD_SIZE][CELL_SIZE];

int frequencyRecord[RECORD_SIZE];

int index;

*}**record_t;***

A structure type that contains 2D record arrays for word, line, document ID as well as an array for frequency. Index is the value of next free record position.

void Record_Display_All(record_t *record);

Displays contents of a record with words, lines, document IDs and frequencies.

@param **record* Pointer to a record_t type structure which contents are to be displayed.

void Record_Display_Single(record_t *record, char files[FILES_MAX][CELL_SIZE], int index);

Displays a single position of a record. This function will also display the name of a text file.

@param **record* Pointer to a record_t type structure which single position is to be displayed.

@param *files[FILES_MAX][CELL_SIZE]* Pointer to an array containing names of a file.

@param *index* An address of the record position to be displayed.

void Record_Scan_File(char *fileName, char *path, record_t *record, int doc);

Scans a file in a path with a specific name and updates the record with words found in that file.

@param **fileName* Pointer to an array containing the name of the file with its extension.

@param **path* Pointer to a path in which file is located.

@param **record* Pointer to a record_t type structure which will be updated.

@param *doc* ID number of currently browsed document.

void Record_Add_New(record_t *record, char *word, int line, int doc);

Adds new word and information about line and document ID to the record.

@param **record* Pointer to a record_t type structure to which the information will be added.

@param **word* Pointer to the array which contains new word.

@param *line* Number of line in which the new word occurred.

@param *doc* ID number of document in which the new word occurred.

void Record_Update_Existing(record_t *record, int matchIndex, int line, int doc);

Updates existing position in the record with information about line and document ID.

@param **record* Pointer to a record_t type structure in which one of positions will be updated with new information.

@param *matchIndex* Number of address in the record in which the word was already saved. It can be obtained from *Record_Search* function.

@param *line* Number of line in which the known word occurred.

@param *doc* ID number of document in which the known word occurred.

int Record_Search(record_t *record, char *word, int *matchIndex);

Searches the record for a given word, returns information about a matched position.

@param **record* Pointer to a record_t type structure which will be browsed.

@param **word* Pointer to an array that contains word which will be searched for in the record.

@param **matchIndex* Pointer to an integer which will contain the address of a position in the record in case of a match.

@return Information about the match. 0 – no match, word needs to be added to the record. 1 – match, the word already exists in the record.

c. Searcher

int Searcher_Get_Word(char *word);

Prompts user for word to be searched in the prepared record. Has to take input in the following format: search *word*

@param **word* Pointer to an array in which the word input by the user will be stored for further processing.

@return Information about input data validity. 0 – proper data format. 1 – unknown command.

d. Files

void Files_Search_Extension(char *extension, char *fileName);

Separates the extension from the file name and saves it for further processing.

@param **extension* Pointer to an array in which the extension will be stored.

@param **fileName* Pointer to an array in which the file name with the extension is stored.

int Files_Search_TXT(char files[FILES_MAX][FILE_NAME_SIZE], char *dir);

Searches for all of the .txt files in a specific direction and saves its names.

@param *files[FILES_MAX][FILE_NAME_SIZE]* Pointer to a 2D array to which the file names will be saved.

@param **dir* Pointer to an array that stores the path to a directory which will be searched for the files.

@return Number of .txt files found in the directory.

e. Mystrings

void String_Flush(char *array, int size);

Fills the array with \0 characters up to a specific point.

@param **array* Pointer to the array which will be flushed.

@param *size* Amount of array bytes to be flushed

void String_Copy(char *dst, char *src, int length);

Copies a specific amount of bytes from source array to destination array.

@param **dst* Pointer to a destination array, to which the data will be copied.

@param **src* Pointer to a source array, from which the data will be copied.

@param *length* Amount of bytes to be copied.

void String_Merge(char *dst, char *src);

Merges source array into destination array at the end of destination array.

@param **dst* Pointer to a destination array, to which the data from source will be merged.

@param **src* Pointer to a source array, which will be merged into destination array.

int String_Compare(char *str1, char *str2);

Compares contents of two character arrays.

@param **str1* String 1 for comparison.

@param **str2* String 2 for comparison.

@return Information about identity. 0 – arrays are the same. 1 – arrays are different from each other.

int String_Get_Length(char *str);

Gets information about string length.

@param **str* Pointer to the array which length is being checked.

@return The length of an array – the number of address before the \0 character counting from 0.

char * String_Tokenize(char *src, char *delimiter);

Splits array into words separated by delimiters.

@param **src* Pointer to the array that will be split.

@param **delimiter* Pointer to the array that contains delimiter characters.

@return Pointer to an address of the next word in source array, after previous delimiter character.

4. Summary

The project was completed successfully according to instructions.

The program searches for text files in directory, passed as an argument before running, creates an inverted index array and asks for word to search. After the user inputs the word, it prints lines and document names it appeared in – all according to required format.

In this project, a set of personal functions for operating on strings was used. However, due to issues related with String_Tokenize function it could not be finished and strtok function from string.h library is used to extract words in a line.

To compile the project in a simple way, use $ *make all* command in the main directory.

To check with sample directory, after compilation run $ *./search docs/* and after prompt, search for a sample word, like Caesar or Rome.

Detailed instructions on how to compile the program are stored in Readme.md file.