Assignment #3

20190741 김 지수

Q1) KMP delete

Note:

1) If the input string is fully deleted by matched pattern(s), the program will print “Empty string. The string is deleted by matched pattern(s)” in order to separate the empty string case with the other possible error case that does not print result.

2) When input string or pattern has more than 30 characters, it will not be added and the program will print the error message and repeat the input process until the valid input is given. And it is obvious that ‘0’ length character can not be inputted physically as the system needs at least one character to enter! If only set of blank or enter is given, the system does not consider this as a input. Even though in this situation, when the valid input is given, the program will be correctly operated.

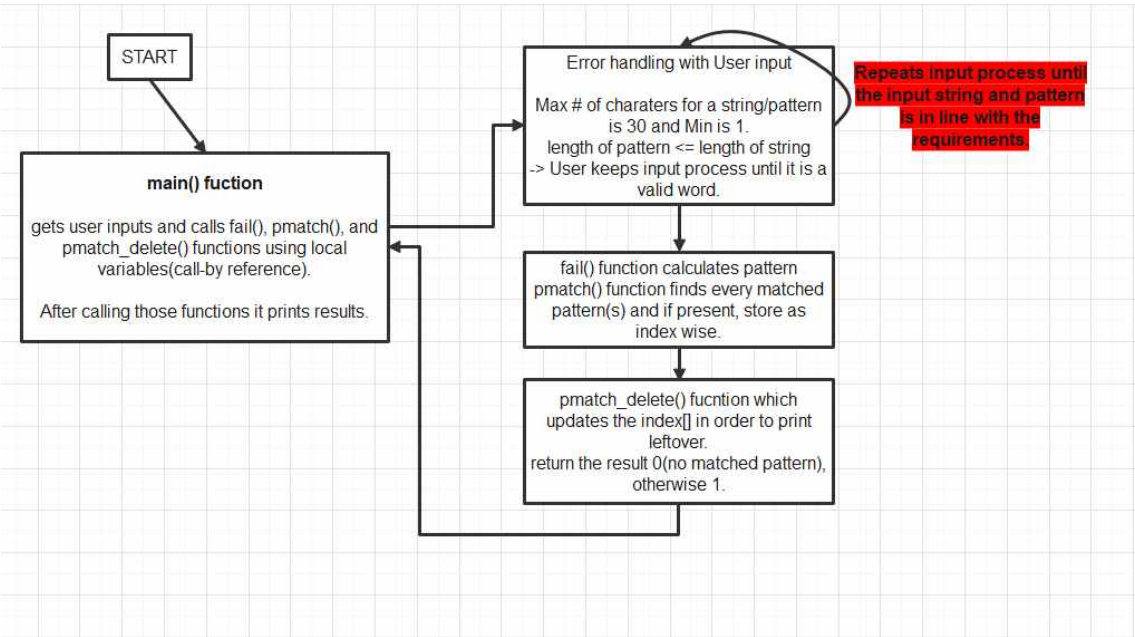
3) It is assumed by Q&A about the Q1, that the word spacing is not considered as a string. So, it is assumed that there is no word spacing in the input string.

4) Failure function is same with that of the class materials. It is obvious that, if it is changed, something strange will happen.

5) The program thinks the pattern, which has same a length compared to the given string, is valid. Thus the input pattern requirement is strlen(pattern)≤strlen(string). When the equality holds and a input pattern is exactly same as the given string, the string is deleted.

Q1-1. Code Description with Flow chart

The flow chart of the Q1 is as follows. When the program starts, the main() function gets user inputs which repeats its process until the input string and pattern is in line with the requirements as noted in flow chart. Next, 3 functions, fail(), pmatch(), and pmatch_delete() function is called in main function. Note that the specific algorithm will be introduced in Q1-2. After the 3 functions being operated, the final result is printed.



<Flow chart, Q1>

Q1-2. Main Algorithms with Pseudo code

The main algorithm, in need of description, is pmatch_delete() function. As we studied in class about the fail() function and pmatch() function, the algorithm of those two functions will not be introduced. pmatch() function in the program simply stores all the starting index values that the matching starts, every time when the pattern is found at the string until the string ends.

Meanwhile, pmatch_delete() function is something different. The Q1 said the string is deleted all of the matched parts by index wise. For example, if the string is bbbbbabbbbbc and patern is bbb, the output is ac because all matched parts in index wise is deleted. That means, it is clever not to impatiently delete the string when we find a matched part. Rather, the program updates to 0 of each index that is deleted by pattern matched. Note that each value of the index[] array was previously initialized to 1.

Pmatch_delete() function outputs a integer value that is stored in an integer variable result in main function to notice whether there is any deleted part. Then, the main function prints the deleted string result finally. More specifically, if the result variable gets 0 from pmatch_delete() function, there is no matched pattern, otherwise it will be 1.

Finally, note that the dynamically allocating technique is used to allocate char string*, *pat. The reason why is to avoid core dumped(Segmentation fault) situation. When the input string is out of bound, the allocated memory is freed and reallocated to control segmentation fault situation while not wasting the memory. The Pseudocode of Q1 is as follows.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void fail(char*, int failure[]);
void pmatch(char*, char*, int ans[], int failure[], int *);
int pmatch_delete(int ans[], int index[], int *count, int lenS, int lenP);

int main() {
        int failure[100], ans[100], count = 0, index[100], result;
        char *string, *pat;
        int lenS, lenP;

        while (1) {
            dynamically allocate string;
                input string until that is valid in line with the requirements.
                if it has error, free string and continue.
        }
        while (1) {
            dynamically allocate pat;
                input pat until that is valid in line with the requirements.
                if it has error, free pat and continue.
        }
        call fail(pat, failure); function.
        call pmatch(string, pat, ans, failure, &count); function.
        call pmatch_delete(ans, index, &count, lenS, lenP); function and store the result on variable result.

        // print result
```

```
        if (no matched) {
                print the whole string as there is no matched result.
        }
        else {
            if (string remains after deletion) print the string which the matched pattern is deleted in index wise.
                if (string fully deleted) print the Empty string message.
        }
    free memory
    return 0;
}
void fail(char* pat, int failure[]) {
        int i, j, n = strlen(pat);
        failure[0] = -1;
        for (j = 1; j < n; j++) {
            make failure[] in line with the definition of fail as follows.
```

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

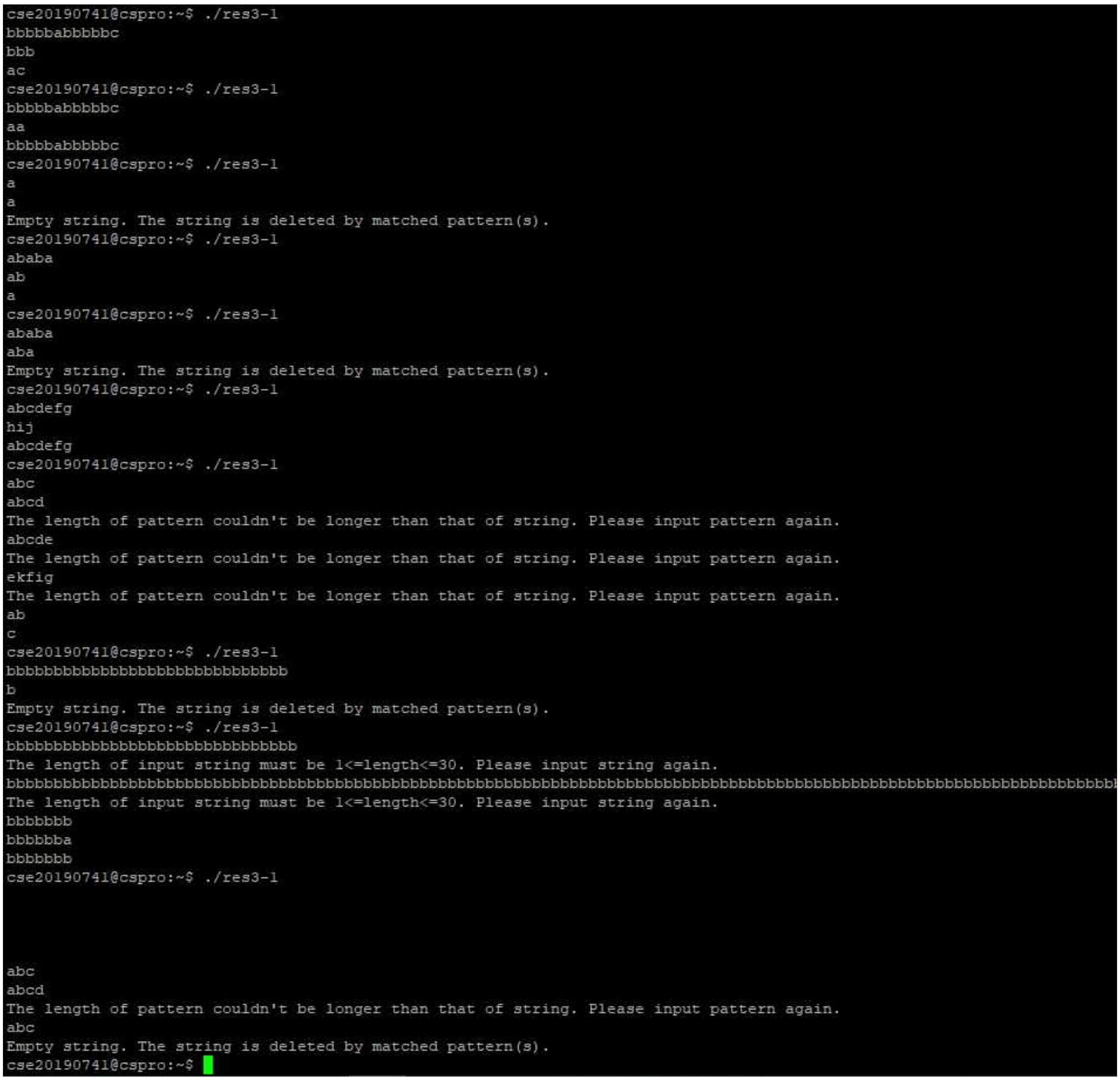
note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j)))$.

```
        }
}
void pmatch(char* string, char* pat, int ans[], int failure[], int *count) {
        int i = 0, j = 0, temp;
        int lens = strlen(string);
        int lenp = strlen(pat);

        Using following while loop, update ans array each time when the patern matched.
        That is put all starting indexs where the pattern matches.
    Used logic is same as that of pmatch() studied in class.

        while (i is lower than lens) {
                if (j is equal to lenp) {
                        ans[*count] = i - lenp;
                        temp = j;
                        (*count)++;
                        j = failure[temp - 1] + 1;
                }
                else if (string[i] is equal to pat[j]) {
                        i++; j++;
                }
                else if (j == 0) i++;
                else j = failure[j - 1] + 1;
        }
        if (j is equal to lenp) {
                ans[*count] = i - lenp;
                (*count)++;
        }
}
int pmatch_delete(int ans[], int index[], int *count, int lenS, int lenP) {
        int i, j, k;
        initialize every index[] value to 1 for lenS(string) size.
```

```
        if (*count is equal to 0) {
                there is no matched pattern
                return 0;
        }
    update matched indexes to 0 using for loops.
        for (i = 0; i < *count; i++) {
                j = ans[i];
                k = ans[i];
                for ( ;j < k+lenP; j++) {
                        index[j] = 0;
                }
        }
        return 1 to notice that there is at least one pattern matching.
}
```

<Pseudocode, Q1>

Q1-3. Test cases simulated



```
cse20190741@cspro:~$ ./res3-1
bbbbbabbbbbc
bbb
ac
cse20190741@cspro:~$ ./res3-1
bbbbbabbbbbc
aa
bbbbbabbbbbc
cse20190741@cspro:~$ ./res3-1
a
a
Empty string. The string is deleted by matched pattern(s).
cse20190741@cspro:~$ ./res3-1
ababa
ab
a
cse20190741@cspro:~$ ./res3-1
ababa
aba
Empty string. The string is deleted by matched pattern(s).
cse20190741@cspro:~$ ./res3-1
abcdefg
hij
abcdefg
cse20190741@cspro:~$ ./res3-1
abc
abcd
The length of pattern couldn't be longer than that of string. Please input pattern again.
abcde
The length of pattern couldn't be longer than that of string. Please input pattern again.
ekfig
The length of pattern couldn't be longer than that of string. Please input pattern again.
ab
c
cse20190741@cspro:~$ ./res3-1
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
b
Empty string. The string is deleted by matched pattern(s).
cse20190741@cspro:~$ ./res3-1
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
The length of input string must be 1<=length<=30. Please input string again.
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
The length of input string must be 1<=length<=30. Please input string again.
bbbbbbb
bbbbbba
bbbbbbb
cse20190741@cspro:~$ ./res3-1




abc
abcd
The length of pattern couldn't be longer than that of string. Please input pattern again.
abc
Empty string. The string is deleted by matched pattern(s).
cse20190741@cspro:~$
```

<Test results, Q1>

As simulating the many test cases like the upper image, could make the program more reliable by finding the corner cases. As a result, there is no program error or incorrect result found for several test cases by comparing the test cases results with other students.

Q2) Dictionary

notes)

1. Only “exit” input is set to terminate the program in line with the given pdf requirement.
2. When input word is same, it will not be added and the program says “the dictionary has same word, and input again.” and get another input.(Until the correct input is given, the total number of words unchanged of course.)
3. When input word has more than 100 characters, it will not be added and the program says “The number of characters of input word is over than 100. Please input less than or equal to 100 characters.” and get another input.(Until the correct input is given, the total number of words unchanged of course.)
4. The program does not count blank or just enter. It is not recorded as a input until valid input(even if it has more than 100 characters, it is valid in input wise but is not valid input word) is input(ted). When the valid input is given next on the blank or just enter, it operates well.
5. Each time when the new word is input(ted), the program dynamically allocates the memory. If memory allocation problem occurs(may happen when the computer has a very low memory, for example 1mb, although this is a very extreme case.), the program terminates with the error message.
6. When the total input number of words exceeds 100 words, the program says “The number of input words is now over than 100. Program terminates.” It can only store up to 100 words. A corner case is tested.
7. As it is English dictionary, it is assumed that only alphabet characters will be given to program. Program does not handle this because it is a “English word dictionary” that is in line with the Q2 said(i.e. Jinho enters a English word).

Q2-1. Code Description with Flow chart

The flow chart of the Q2 is as follows. When the program starts, main function calls liveSort function which gets a english word and prints the sorted result each time when the input is given by using compare function. Compare function, which compares two given strings, has similar functionality with strcmp function. As the Q2 restricts the use of string.h library the program implements the compare function. At liveSort() function, 3 steps are as follows.

1. Error handling with User input.

As, noted earlier the maximum number of characters for a input word is 100 and same words are not allowed. So, program says the error message and keeps the User input process until the valid english word is given.

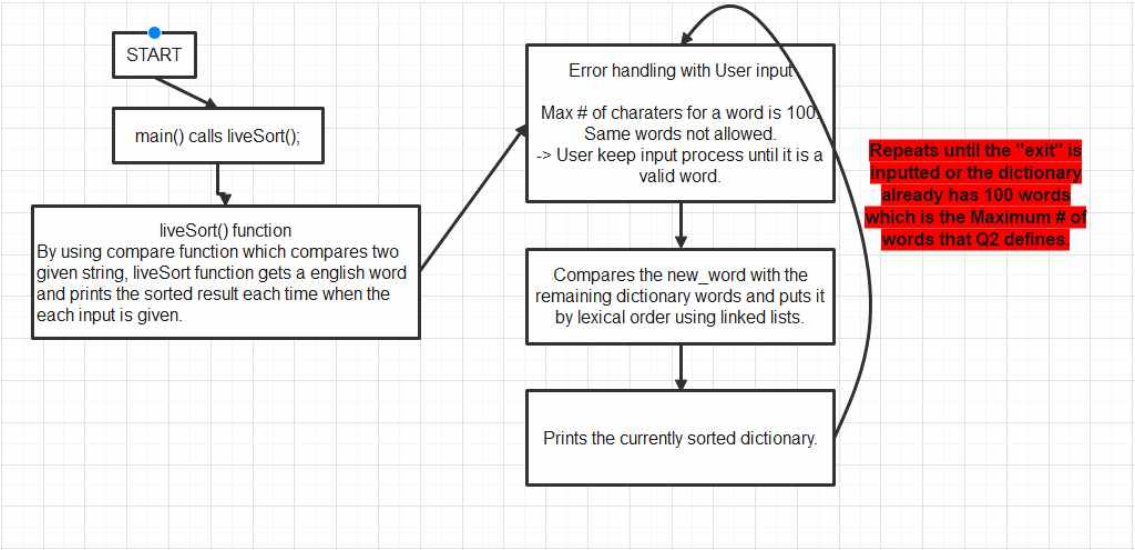
2. Comparing and Sorting process

Considering time consumptions, the program uses linked lists to store the words in dictionary because sorting becomes very time consuming when we use arrays in implementation. As a result, in this process, the new_word(new input english word in code) is compared with the remaining dictionary and put by lexical order at linked lists.

3. Prints the currently sorted dictionary.

The sorted dictionary is printed in line with the output form of Q2.

Finally, this process is repeated until the exit is inputted or the dictionary already has 100 words which is the maximum number of words that Q2 defines.



<Flow chart, Q2>

Q2-2. Main Algorithms with Pseudo code

Frankly speaking, there is no specific “algorithms” in the program because it uses the linke d lists in sorting, that is mainly a simple implemetation. So, it is thought that a big flow of th e program is important. However, it is better to note that in the error handling process, ever y time when the input word is invalid, the memory allocated to the new_word is freed and re allocated in order to avoid “Segmentation fault”, while not wasting the memory. In error hand ling process, countW counts the number of characters at given words and countT counts the total number of words.

The pseudocode of Q2 is as follows.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

/* struct and struct pointer definition is not a global variable*/
struct data {
    char word[100000];
    struct data* link;
};


int compare(char* str1, char* str2);
void liveSort();

int main() {
    liveSort();
    return 0;
}

int compare(char* str1, char* str2) {
    int i = 0;
    while (str1[i] is not '\0' and str2[i] is not '\0') {
        if (str1[i] is equal to str2[i]) {
            i++;
```

```
            continue the process becuase i th element of str1 and str2 is equal.
        }
        else if (str1[i] is lower than str2[i]) {
            return -1;
        }
        else {
            return 1;
        }
    }
    // have same characters until the index min(length of str1, length of str2)-1.

    if (str1[i] is '\0' and str2[i] is '\0') {
        return 0;    //equal case
    }
    else if (str1[i] is '\0') {
        return -1;     //str2 is same as str1 until index(i-1) and has more characters
    }
    else {
        return       1;     //str1 is same as str2 until index(i-1) and has more characters
    }
}

void liveSort() {
    int num = 0, countT = 0;
   dynamically allocate struct data* head, tail, and curr.

    while (1) {
        int found = 0, cmp = 0, count = 1;
        struct data* new_word;

        while (1) {
            int countW = 0, same = 0;
            dynamically allocate new_word and handles Memory allocation error.
            get a new word using scanf()

            while (#of characters in words <= 100 && new_word->word[countW] is not '\0') {
                counting number of characters in a new input word.
            }
            if (exit is inputted) {
                exit(1);
            }
            if (new input word has more than 100 characters) {
                Error handling statement.
                free new_word and keep input process.
                continue;
            }
            if (total number of input word is bigger than 100) {
                Error handling statement.
                exit(1);
            }
            if (regular cases) {
                set current linked list to head.
                while (there is no remaining word to compare with in the dictionary) {
                    if (there is same word) {
```

```
                        Error handling statement.
                        set same to 1 in order to recognize there is a same word.
                        free new_word and keep input process.
                        break;
                    }
                    curr = curr->link to chechk next.
                }
            }
            if (there is same word in dictionary) back to input process;
            if (there are more than 100 characters in words) back to input process;
            in the normal cases break; to go on to a next process.
        }
        countT++;
        //count total number of normal input

        if (it is the first input) {
            head = new_word;
            tail = new_word;
        }
        else {
            compare each elements of the dictionary and put the newly inputted word by lexical order.
        }
        num++;
        // to recognize the very first input.
        print the sorted result each time after the valid input following the output form.
    }
    free memory;
}
```

<Pseudocode, Q2>

Q2-3. Test cases simulated

As simulating the test cases could make the program more reliable by finding the corner cas es, following cases are tested.

- Inputting one word
- Inputting two words in lexicographic order
- Inputting two words in reverse lexicographic order
- Inputting multiple words in random order
- Inputting the same word multiple times
- Inputting the exit command
- Inputting a word with more than 100 characters
- corner cases(ex. “a” repeated 101 times is rejected, 100 times accepted)

As a result, there was no program error or incorrect result found for several test cases.