Data Structure Task 3 Report

20200901 이효주

Problem1.

1. Problem definition)

This problem requires finding all parts of a given string that match a given pattern and writing a function pmatch_delete() that outputs the result of deleting the part, and the global variable is not available.

2. Approach)

First, the approach to this problem used the kmp algorithm learned in class to find a part that matches a given pattern in a given string and output the result of deleting the part. The fail() function was used to check how far the pattern string matches, and the calculated failure function was returned as int *.

The pmatch_delete() function finds the part of a given string that matches the given pattern. This function uses the fail() function to find the starting index of any part of the pattern that matches a given string. Then, delete the string as long as the pattern from the index. If you repeat this action for all starting indexes, you can delete all patterns. Finally, output the deleted result string.

To explain in detail, in the pmatch_delete() function, if the length of the pattern is longer than the length of the string, we first found the failure function of the pattern and then compared it by the length of the string. If the characters of the string and the pattern are the same, 1 is stored in the current index of the array called result initialized to 0.

Therefore, for deletion, the array was initialized to 0, so only if it is not 1, it is added to the string, and if the value of the array is 1, the pattern and the string are the same, so it is not added.

3. Pseudo code)

```
fail(pattern):
    lenp = length of pattern
    failure = [0] * lenp

    for i from 1 to lenp - 1:
        j = failure[i - 1]
        while j > 0 and pattern[i] != pattern[j]:
            j = failure[j - 1]
        if pattern[i] == pattern[j]:
            failure[i] = j + 1
        else:
            failure[i] = 0

    return failure

pmatch_delete(string, pattern):
    lenp = length of pattern
    lens = length of string

    if lenp > lens:
        return -1
```

```
    failure = fail(pattern)
    j = 0
    result = [0] * 31
    result2 = ""

    for i from 0 to lens - 1:
        while j > 0 and string[i] != pattern[j]:
            j = failure[j - 1]
        if string[i] == pattern[j]:
            j += 1
            if j == lenp:
                for k from i - lenp + 1 to i:
                    result[k] = 1
                j = failure[j - 1]

    for i from 0 to lens - 1:
        if result[i] == 0:
            append string[i] to result2

    print result2
    free failure
    return 0

main():
```

read string from stdin

read pattern from stdin

call pmatch_delete(string, pattern)

return 0

Problem2.

1. Problem definition)

This problem is a problem of sorting and outputting English words received from users according to Lexical order. The English word is input from the user, and it is repeated until the exit is input. Do not use static assignments or <string.h>

2. Approach)

This problem is that the input string is received, sorted every time, and then outputted. This requires the input string to be stored in the array.

First, an array array S for storing a string and a variable count for storing the number of strings are initialized.

It then goes around the infinite loop and receives a string, and if the input string is "exit", it exits the loop and exits the program. If it is not "exit", the string is stored in the dynamically allocated memory space and the number of strings is increased.

After the new string is stored, alignment is performed with all previously stored strings. The alignment algorithm was implemented by comparing all strings.

When the alignment is completed, the array S is output. At this time, I was careful not to print commas after the last string.

For dynamic memory allocation, malloc and reloc functions were used, and memory was released using free functions to prevent memory leakage. In addition, the function my_strcmp that compares strings was directly implemented and used.

3. Pseudo code)

1. Define function swap(a, b):

```
temp = a
a = b
b = temp
```

2. Define function my_strcmp(str1, str2):

```
i = 0
while str1[i] != '₩0' AND str2[i] != '₩0' do
    if str1[i] < str2[i] then
        return -1
    else if str1[i] > str2[i] then
        return 1
    end if
    i = i + 1

if str1[i] == '₩0' AND str2[i] == '₩0' then
    return 0
else if str1[i] == '₩0' then
    return -1
else
    return 1
```

3. Define function sorting(arrayS, count):

```
for i = 0 to count-1 do
    for j = i + 1 to count-1 do
        if my_strcmp(arrayS[i], arrayS[j]) > 0 then
            swap(arrayS[i], arrayS[j])

for i = 0 to count-1 do
    if i == count-1 then
        print arrayS[i] + "₩n"
    else
        print arrayS[i] + ", "
```

4. Define main function:

```
count = 0
arrayS = NULL

while true do
    string = read string from user input and dynamic allocation
    if string == "exit" then
        for i = 0 to count-1 do
            free(arrayS[i])
        end for
        free(arrayS)
        return 0
```

```
count = count + 1
arrayS[count-1] = string
sorting(arrayS, count)
```