Lab Exam 3

In this lab, you are going to be familiar with "Struct" and "Linked List" in C. As in the preliminary lab, it consists of two parts, which are "Basic Introduction to Struct" and "Linked List Operations".
Note: For convenience, linked lists will be represented as follows in this document:

```
2 -> 3 -> 1 -> 5 -> 1 -> 4
```

Note: As a difference from the preliminary lab, we have shared create_list() and print_list() functions with lab3.h file. You can see their declarations in the lab3_copy.h file, however, you can use them **only for the second part of the lab exam**. While testing your codes with the "Run" option, you can use these functions. Example usage (You can see it also in run.c file):

```
int data[5] = {1, 3, 4, 2, 5};
Node* head = create_list(data, 5);
/* You can call your functions here */
print_list(head);
```

Important Specifications

- You can not use any library other than provided libraries as <stdio.h> and <stdlib.h>.
- DO NOT use keywords static.
- You are NOT allowed to use global variables anywhere.
- You are NOT allowed to change the part above the "Do not change the lines above" line.
- Please submit your implementations in the "lab3.c" file.
- You can define your own helper functions.
- Make sure that functions given to you as tasks can be called "as is". DO NOT change their signature.
- Make sure that your solution works with the examples that are given to you in the description first.
- Your codes will be compiled with "-ansi -Wall -g -O0 -pedantic-errors" flags. Please follow ANSI standards otherwise your code will get errors.
- As in the previous labs, a copy of the header file that contains the function prototypes is also provided as "lab3_copy.h":
    - It is an exact copy of lab3.h, which will be used while evaluating your codes.
    - This file is not used in the evaluation, it was shared with you to inform what is in lab3.h file.
    - Any change in this file will have no effect in the run, debug or evaluate commands.

- You can use the "run.c" file for testing your functions with the run and debug commands, modify it however you like. When you click the "run" button, main in the "run.c" will be evaluated. This file will not be used in the evaluation.

1. Basic Introduction to Struct

In this part, you are provided two structs which are "Student" and "Teacher", you can see their declaration below and in the lab3_copy.h file. As you can see, in the "Student" struct definition there is a "Teacher* best_teacher" variable which points to a "Teacher" for each "Student".

```
typedef struct Teacher
{
  int teacherID;
} Teacher;

typedef struct Student
{
  int studentID;
  Teacher* best_teacher;
} Student;
```

Question - 1 (15 Points)

Function:

```
Student** create_student_array_with_teachers(int* student_id_list, int*
teacher_id_list, int student_and_teacher_count)
```

Explanation:
You are given three parameters, which are student_id_list, teacher_id_list and student_and_teacher_count and two structs, which are "Student" and "Teacher" structs. For each student, there is only one unique best teacher, therefore, student_id_list and teacher_id_list have the same length and you can use student_and_teacher_count for each array.
student_id_list and teacher_id_list are prepared respectively, which means studentID of first student and teacherID of first student's best_teacher is at 0th index in student_id_list and teacher_id_list respectively. You should take studentIDs one by one from student_id_list array and teacherIDs one by one from the teacher_id_list. Then, create a student and a teacher for each studentID and teacherID by connecting them using best_teacher variable in the "Student" struct.
You should create and return an array whose elements are student pointers (student*). You do not have to create an additional teacher array, connecting each teacher with a proper student is enough.

Important Note:
The order of the studentIDs in student_id_list and your returned array must be the same.
Example: If the student_id_list is [1, 3, 2], then your returned array must be [pointer_to_Student_whose_id_1, pointer_to_Student_whose_id_3, pointer_to_Student_whose_id_2].

Example:

```
    student_id_list[5] = {1, 2, 3, 4, 5}
    teacher_id_list[5] = {2, 8, 5, 6, 1}
    student_and_teacher_count = 5
    returned_array = create_student_array_with_teachers(student_id_list,
teacher_id_list, student_and_teacher_count);
    returned_array: [pointer_to_student_whose_id_1_and_id_of_best_teacher_2,
pointer_to_student_whose_id_2_and_id_of_best_teacher_8,
                     pointer_to_student_whose_id_3_and_id_of_best_teacher_5,
pointer_to_student_whose_id_4_and_id_of_best_teacher_6,
                     pointer_to_student_whose_id_5_and_id_of_best_teacher_1]
```

2. Linked List Operations

In this part, you are going to implement several functions related to linked lists. A linked list is a data structure in which objects refer to the same kind of object and are linked in a linear sequence.
Each node in the linked list is composed of a value whose type is int and a pointer to the next node. The next pointer of the last node in the linked list should be NULL.

Linked list "Node" is defined in lab3.h file, you can see the declaration below and in the lab3_copy.h file.

```
struct Node
{
  int data;
  struct Node *next;
};
typedef struct Node Node;
```

Question - 2 (10 Points)

Function:

```
Node* insert_tail(Node* head, int new_node_data)
```

Explanation:
With the given new_node_data, you should create a new node and add it to the end of the given linked list.
You have to return the head of the changed list. You should not create a new list but update the existing list. If the head of the list does not change, after the operation, you can just return the same head value as the given input.

Note:
If the given list is empty (NULL), as you can understand, you should add the new node to the head.

Example 1:

```
list (before): 1 -> 5 -> 3 -> 4
new_node_data = 2
list = insert_tail(list, new_node_data);
list (after): 1 -> 5 -> 3 -> 4 -> 2
```

Example 2:

```
list (before): NULL
new_node_data = 3
list = insert_tail(list, new_node_data);
list (after): 3
```

Question - 3 (15 Points)

Function:

```
Node* remove_kth_node_reversely(Node* head, int index)
```

Explanation:
You should remove the node at the given index in reverse order and connect the previous and next nodes of the removed node. Range of "index" variable is [0, length_of_list-1].
You have to return the head of the changed list. You should not create a new list but update the existing list. If the head of the list does not change, after the operation, you can just return the same head value as the given input.

Note:
If there is only one node in the given list, as you can understand, the new list must be empty.

Example 1:

```
list (before): 1 -> 5 -> 3 -> 4
index = 1
list = remove_kth_node_reversely(list, index);
list (after): 1 -> 5 -> 4
```

Example 2:

```
list (before): 1 -> 5 -> 3 -> 4
index = 0
list = remove_kth_node_reversely(list, index);
list (after): 1 -> 5 -> 3
```

Example 3:

```
list (before): 1
index = 0
```

```
list = remove_kth_node_reversely(list, index);
list (after): NULL
```

Question - 4 (20 Points)

Function:

```
int find_occurrences_of_second_list(Node* head, Node* second_head)
```

Explanation:

You are given two lists which are head and second_head. You should search the entire list and return how many times the second list is found in the first list by comparing the data values of the nodes.

Example 1:

```
list: 1 -> 5 -> 3 -> 2 -> 4
second_list: 3 -> 2
occurrences = find_occurrences_of_second_list(list, second_list);
occurrences: 1
```

Example 2:

```
list: 1 -> 5 -> 3 -> 5 -> 3 -> 5 -> 3 -> 4
second_list: 5 -> 3 -> 5
occurrences = find_occurrences_of_second_list(list, second_list);
occurrences: 2
```

Example 3:

```
list: 1 -> 2 -> 2 -> 2 -> 2 -> 4
second_list: 2 -> 2
occurrences = find_occurrences_of_second_list(list, second_list);
occurrences: 3
```

Example 4:

```
list: 1 -> 4 -> 4 -> 3 -> 2
second_list: 4
occurrences = find_occurrences_of_second_list(list, second_list);
occurrences: 2
```

Example 5:

```
list: NULL
second_list: 1 -> 2
occurrences = find_occurrences_of_second_list(list, second_list);
occurrences: 0
```

Question - 5 (20 Points)

Function:

```
Node* cut_and_paste(Node* head, int index_to_cut, int index_to_paste)
```

Explanation:
You should cut the node at index_to_cut position and paste it to index_to_paste position by shifting the node at paste position to forward. Range of both "index_to_cut" and "index_to_paste" variables is [0, length_of_list-1]. When index_to_paste > index_to_cut, paste index might create an ambiguity. Therefore, index_to_cut ≥ index_to_paste (index_to_cut will always be greater than or equal to index_to_paste.).
You have to return the head of the changed list. You should not create a new list but update the existing list. If the head of the list does not change, after the operation, you can just return the same head value as the given input.

Important Note:
It is **not a copy and paste operation**, it is **a cut and paste operation**. You should **not** create a new node to add, you should **MOVE** the node at the given index_to_cut.

Example 1:

```
list (before): 1 -> 5 -> 6 -> 2 -> 4
index_to_cut: 3
index_to_paste: 0
list = cut_and_paste(list, index_to_cut, index_to_paste);
list (after): 2 -> 1 -> 5 -> 6 -> 4
```

Example 2:

```
list (before): 1 -> 5 -> 2
index_to_cut: 1
index_to_paste: 1
list = cut_and_paste(list, index_to_cut, index_to_paste);
list (after): 1 -> 5 -> 2
```

Example 3:

```
list (before): 1
index_to_cut: 0
index_to_paste: 0
list = cut_and_paste(list, index_to_cut, index_to_paste);
list (after): 1
```

Question - 6 (20 Points)

Function:

```
Node* sum_symmetric_pairs(Node* head)
```

Explanation:
You should create and return a new list by summing the data values of the symmetric node pairs of the given list. You can also edit the given list instead of creating a new list. The given list will not be checked while grading, only returned list will be compared with the expected list.
Example: If the given list is 1 -> 3 -> 4 -> 5, the node count of the new list must be 2.

Data of the first node in the new list must be 1 + 5 = 6 and data of the second node in the new list is 3 + 4 = 7. Then, the new list must be 6 -> 7.

Important Note:
Node count of the given list will always be a multiple of 2 and greater than or equal to 2.

Example 1:

```
list: 1 -> 5 -> 3 -> 4
new_list = sum_symmetric_pairs(list);
new_list: 5 -> 8
```

Example 2:

```
list: 1 -> 4
new_list = sum_symmetric_pairs(list);
new_list: 5
```