Project 1: Implementation of Linked Lists
KECE208 Data Structures and Algorithms
Fall, 2018

Instructor: **Hwangnam Kim**

TA: **Seounghwan OH, Inseop UM**

GOAL

In project1, you will be assigned to use your knowledge with Linked List and Stack in order to fill in the functions that are provided at project files. Students are required to fill in empty functions in order to get outputs and get grades on those outputs. In this project, you are allowed to use “struct linked_list” which is the list structure for this project that contains head, tail and size of the list. Also, you will be given “struct linked_node,” which contains prev, next, and element of the node. Please refer to ‘linked_list.h’ for more detail.

Given Code:

linked_list.h / linked_list.c / project1_main.c

You are responsible for completing implementation of *‘linked_list.c’* in order to make the program run properly. You are not allowed to modify *'linked_list.h'* and *'project1_main.c'*.

- **linked_list.h**: This is the header file of linkLQS.c that declares all the functions and values that are going to be used in linkLQS.c. You do not have to add any more functions or variables in order to complete the project and you are NOT ALLOWED to.

- **linked_list.c**: This contains implementations and definitions of functions and variables and you are required to fill some of the functions that contain comments “Your code starts from here.” By filling and completing implementation of those functions, you will get proper output for this project.

- **project1_main.c**: This file is provided to students in order to debug whether their codes works correctly or not. The usage of this class will be explained later during project1 guide lecture. Time and location will be announced shortly.

- **Makefile:** This file compiles the c files listed on this file. If students type “make”, this will compile all the code. But when students type “make clean”, this will erase executable code and force the user to recompile later.

- **project1.exe:** This is an executable code and provides right results of commands. Students must be having same output as this executable file generates. In order to execute this file, just type “./project1.exe”

Structure of the linked list and the linked node:

```
struct linked_node{
        int value;
        struct linked_node* next;
        struct linked_node* prev;
};

struct linked_list{
        int type_of_list;
        struct linked_node* head;
        struct linked_node* tail;
        int number_of_nodes;
};
```

These are written in *listed_list.h* file and you are not allowed to modify any of these.

```
int list_exist;
```

Also, the above value is declared as a global variable and this value should be equal to 0 when there is no list created, and should be equal to 1 if a list is created. Since this value is used in print_list function, students should modify this value whenever the existence of list changes.

Description of the functions that are provided is listed below:

1. You need to follow exactly what the descriptions say and implement each function according to the description.
2. Each function also has some error cases that it should be dealt with. Each error case always should alert the user with message and exit the function. Descriptions below will show what alarm message each function should generate, and you have to follow exactly the way it is written.
3. Always assume that node value is unique and more than one node with same value cannot be entered to the list. This condition will never be tested so that students do not have to implement this part.
4. Node value can be any integer value and you can assume that only integer will be inserted to the function so that you do not have to worry about handling this kind of error case.
5. When removing or inserting a node to a list, you should always change the *number_of_nodes* value of the list.
6. You are not allowed to use any library nor codes that are available through the web, friends, and etc. You are only required to use the give code and the files that are included within the .h file.
7. If you cannot make your code when TA ask you to, we look on as cheating

struct linked_list* create_list (int number_of_nodes, int list_type)
This function creates a *list* with the number of nodes. Number of nodes should be greater than 0 but if the input is less than 1, it should print the error message "Function create_list: the number of nodes is not specified correctly" and exit the function. Also, if a *list* already exists, then it should print the error message "Function create_list: a list already exists" and exit the function. There are two types of list, normal or stack, but the input value for the type is not 0 or 1, it should print the error message "Function create_list: the list type is wrong". When the nodes are inserted to the *list*, the node value created randomly but no duplication. And value's range is from 1 to number of nodes. After the *list* is generated as specified, the function returns this *list*. This function allows the list to insert new nodes to either the head or the tail according to the input value(list_type).

struct linked_node* create_node (int node_ value)
This function creates a node that can be inserted to a list. *Node value* can be any integer.

void insert_node(struct linked_list* list, struct linked_node* node)
This function inserts a *node* to a *list*. This insertion will always insert a *node* to a head of the *list* which means the head of the *list* should be changed after a *new node* is inserted. For least hints, there is no duplication check in this function.

void print_list(struct linked_list* list)
This function prints a *list* in proper way.
Functions create_list, insert_node and create_node will be provided to students so that those can be used as reference of how to implement the functions and deal with the error cases. Also, print_list function will be provided to students as debugging tool so that students can see whether the output is properly made or not.

The assignments:

void remove_list(struct linked_list* list)
This function removes the *list*. When the *list* is removed, all the memory should go back to the computer so that other programs or values should be capable of using this. While deleting the *list*, every node should be freed separately; free(*list*) will not remove every node in the *list*. To check whether the nodes are removed perfectly, for every deletion of a node, this function should print message "The node with value n (corresponding value) is deleted!" Also, if the whole *list* is deleted, this function should print message "The list is completely deleted: n

nodes are deleted".

void advanced_insert_node(struct linked_list* list, struct linked_node* node, int nth_node, int start_point)
This function inserts a node to a *list*, but different from above insertion function. The *new node* will be inserted to the N-th order of the *list*. N-th order could be from head of list or tail of list. In case of start_point is 0, order would start from head and case of 1, order would start from tail. For instance, if the start_point is 1 and node 8 is inserted at order 3 in the list [1 2 3 4 5], the result would be [1 2 3 8 4 5]. If n-th_node is bigger than number of nodes, the *new node* would be tail/head of the list. However, if the *node* value is not given correctly (less than 0), this function should print the error message "Function advanced_insert_node: The specified order is not proper value to be used" and exit the function. Also, if a same *new node* value exists in the *list*, this function should print the error message "Function advanced_insert_node: A duplicate number exists".

void remove_node(struct linked_list* list, int rm_node_value)
This function removes a *node* with specified value. If there is only one node in the *list*, remove the *node* and remove the *list* also since there is nothing left. While removing a *node*, the *node* should be perfectly freed. If the type of list is stack, print the error message "Function remove_node: The list type is not normal. Removal is not allowed" Also, if there is no such node to remove from the *list*, print the error message "Function remove_node: There is no such node to remove" and exit the function.

void push_Stack (struct linked_list* list, struct linked_node* node)
This function inserts a node in stack manner. If the type of list is not stack, print the error message "Function push_Stack: The list type is not a stack" The new node will be always inserted to tail of the list which means the tail of the list should be changed after a new node is inserted.

void pop_Stack (struct linked_list* list, int number
This function removes some nodes in stack manner; the tail of the list will be removed, repeatedly. The parameter (variable *number_of_nodes*) means the number of nodes which will be removed. When parameter is bigger than 1, popping a node with n times, you do not remove node at one go. If there is only one node in the list, please make sure it frees (de-allocates) both the node and the list. If the list is not a stack type, print the error message "Function pop_Stack: The list type is not a stack" and exit the function. If the *number_of_nodes* parameter is less than 1 or more than the number of nodes in the stack, respectively print the error message "Function popStack: The number of nodes which will be removed is more than 1" and "Function popStack: The number of nodes which will be removed is more than that in the stack", then exit the function. The removed nodes should be freed.

void search_node(struct linked_list* list, int find_node_ value)
This function finds the node from the *list* that value is same with *find_node_value* and count the order of the node. This function should print message "The order of (node_value) is (order)." and error message "Function search_node : There is no such node to search.".

void reverse_range(struct linked_list* list, int order1, int order2)
This function makes the list to be rearranged in reverse order from *order1* to *order2*. For example, if the user input *order1* is 3, *order2* is 7 and the *list* is composed of nodes [1 2 3 4 5 6 7 8 9 10], the output of this function would be [1 2 7 6 5 4 3 8 9 10]. If *order1* is bigger than *order2* or *order2* is bigger than *number_of_nodes* of the *list* or *order1* is smaller than 0, print the error message "Function reverse_range: Input value is invalid" and exit the function.

void modify_node(struct linked_list* list, int modify_node_value, int new_value)
This function modifies the value of a node into a new value. If the new value already exist in the list, this function should print the error message "Function modify_node: A duplicate number exists". If the function executes without any error, it should print the message "The value (modify_node_value) turns into (new_value)." If modify_node_value does not exist in the list, this function should print the error message "Function modify_node: There is no such value in the list".

void odd_sequence_reverse(struct linked_list* list)
This function makes group with its nodes in odd sequence numbers and make the groups rearranged in reverse order. For instance, if the list is composed of nodes [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20], odd sequence number groups would be [(1), (2, 3, 4), (5, 6, 7, 8, 9), (10, 11, 12, 13, 14, 15, 16), (17, 18, 19, 20)]. And the output of this function would be [1 4 3 2 9 8 7 6 5 16 15 14 13 12 11 10 20 19 18 17]. If the size of last group

of list is smaller than its odd number, it reverse the nodes anyway.
There is no error message.
Comments:
1. You should comment within the function in order to make TAs and yourself what you have been tried to do with codes written. If there is some code that is not understandable and there is no comment on that, points will be deducted.
2. Comments should be all in ENGLISH. Sometimes compile error or running error occurs because the written language is not English. This class is in English so we strongly recommend you to write comments in English. If a student did not write comments in English and some error happened because of that, all the responsibility belongs to the student so functionality points will be deducted.

Project Requirements & Scoring Criteria (100 points total):

1. This is an individual project. Making a copy of the content of your friend's HW, books, google, naver, etc., will get an F for this course. You will not get any partial points for redoing the work. Even if you have different code or variable names, using the same function(s) that is not originated from C library which is initially installed with your Linux (or cygwin, etc.) will be regarded as copying. The person who provides the source will also consider as cheating and will be penalized. Also, asking your friends to do your work will definitely be considered as cheating.
2. If your code does not compile; meaning, there are errors and we cannot fix to grade you work, you will automatically get 0 for Project1 operation part. Please make sure to have compiled your code before turning it in.
3. Compiling in Linux (cygwin) environment is strongly recommended. (5 *extra* points) The initially given code may not be compiled in other compilers. If you use a compiler other than gcc compiler, you must be able to show the compiling process directly to the TA.
4. Codes should have neither error nor warning when compiling: 5 points in total
   A. No errors: 3 points
   B. No warnings: 2 points
5. List operations: 80 points in total
   The following functions should be implemented: 80 points
   A. void remove_list(linked_list* list) (10 points)
      i. Pointer Handling: each node is freed separately and list is freed
      ii. Message is printed as instructed
      iii. Comments
   B. void advanced_insert_node(struct linked_list* list, struct linked_node* node, int nth_node, int start_point) (15 points)
      i. Error handlings as instructed
      ii. Case when nth_node is tail or head.
      iii. Connection of previous and next node each other
      iv. Number of nodes increased correctly
      v. Comments
   C. void remove_node(linked_list* list, int rm_node_value) (8 points)
      i. Error handlings as instructed
      ii. Pointer Handling: the removed node is freed well.
      iii. Number of nodes decreased correctly
      iv. Connection of previous and next node each other
      v. Comments
   D. void push_Stack(linked_list* list, int number)(4 points)
      i. Error handlings as instructed
      ii. Number of nodes inserted correctly
      iii. Connection of previous and next node each other
      iv. Comments
   E. void pop_Stack(linked_list* list, int number)(6 points)
      i. Error handlings as instructed
      ii. Number of nodes decreased correctly
      iii. Connection of previous and next node each other
      iv. Comments
   F. void search_node(linked_list* list, int find_node_value) (5 points)
      i. Error handlings as instructed
      ii. Display of result as instructed

    iii. Comments

G. void reverse_range(linked_list* list, int order1, order2) (12 points)
    i. Error handlings as instructed
    ii. Head and Tail are reconfigured correctly
    iii. Connection of previous and next node each other
    iv. Comments

H. void modify_node(linked_list* list, modify_node_value, new_value)(5points)
    i. Error handlings as instructed
    ii. Value is changed correctly
    iii. Comments

I. void odd_sequence_reverse(linked_list* list) (15 points)
    i. Head and Tail are reconfigured correctly.
    ii. Connection of previous and next node each other
    iii. Comments.

6. Submission: 15 points in total

① **Due date is 11:59PM, November 7 (Wednesday)**
- ● Submit as early as possible. No excuse for returned email (send failure).
- ● No late submission is allowed

② Include a README file in your zip file which explains your program: 10 points

③ All files should be compressed in one .zip file: 2 points
- ● The file name should be: [DSA_PROJECT_1]2018170000_홍길동.zip: 3 points
- ● Submit the zip file to ***seounghwan324@korea.ac.kr***
- ● File should include:
    - i. linked_list.c
    - ii. README file(It should include short explanation of each function. Korean can be allowed).