Solution for 2024-2025 DSA Exam, second sitting, Code 2

Write the necessary data structures and algorithms to reverse the order of the nodes in a single linked list.

The program will be run with command line arguments.

The first argument is the operation to be performed and can be either G (generate list) (4 points) or R (reverse list) (up to 4 points).

The original list will be created by reading integer values from a file. The path of the file will be the second command line argument for your program. The values in this file are separated by a space character. The list will be allocated on the heap.

The list will then be reversed using a recursive algorithm.

The values of each node, in the order after the reversal algorithm, will be printed in decimal format to the standard output separated by one space character. The last value will have one space character after it.

If the first argument is G (generate list) your program should read the input file, generate the dynamically allocated single linked list and print to the standard output the values of each node as it is encountered.

If the first argument is R (reverse list) your program should reverse the nodes in the list.

- **iteratively (2 points), or**
- **recursively (4 points)**

Constraints:

- **There should be no array declared in your code**
- **There should be no unnecessary data structures in your code**
- **Your code should not contain any memory leaks**
- **There should be no unnecessary code**
- **Once created, the value (payload) of the nodes should not be changed**
- **Code must be legible, use comments when you think they help**
- **The list must be generated in the order of the values in the file**

Assume the command line arguments exist and are correct both in number and in value(s).

Assume the input file exists, is not empty and has the correct format.

Do not print extra characters from what is required as this will result in tests fail.

The code must compile. Any errors and subsequent failure to compile will result in the submitted solution to be marked with 0 (zero) points.

The code must run correctly. If the code compiles without errors, but does not generate the correct result, it will be marked with 1 (one) point.

Violation of the constraints will bring massive points deductions, up to 9 points.

Up to 2 (two) points will be given based on the code quality and legibility, complexity of solution and constraints.

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct node{
        int payload;
        struct node *next;
}Node;

void printList (Node *head){
        Node *tmp = head;

        while(tmp){
                printf("%d ", tmp->payload);
                tmp=tmp->next;
        }
}

Node* freeList(Node *head){
        Node *tmp = head;
        Node *del = head;

        while(tmp){
                tmp=tmp->next;
                free(del);
                del = tmp;
        }
return NULL;
}
```

Data structure and type

Run of the mill printList and freeList

```c
Node* addNode(Node *prev, int value){
        Node *tmp = (Node *)malloc(sizeof(Node));
        tmp->payload = value;
        tmp->next = NULL;

        if(prev) prev->next = tmp;

return tmp;
}

Node* generateList(char *inputFile){
        FILE *f = fopen(inputFile, "r");
        int val = 0;
        Node *prev = NULL;
        Node *head = NULL;

        while(!feof(f)){
                fscanf(f, "%d", &val);
                if(feof(f)) break;
                prev = addNode(prev, val);
                if(!head) head = prev;
        }

        fclose(f);
return head;
}
```

Add node at the end of the list.

Read values from file and add to list. An extra pointer to the last node added helps with not having to tranverse the whole list to add a node at the end. Given that all nodes are added as they are read from the file, this turns a O(N^2) into a O(n) operation with memory for just an extra pointer.

```
57
58 Node* reverseListIteratively(Node *head){
59         if(!head || !head->next) return head;
60
61         Node* iter = head->next;
62         head->next = NULL;
63         while(iter){
64                 Node *tmp = iter;
65                 iter = iter->next;
66                 tmp->next = head;
67                 head = tmp;
68         }
69
70 return head;
71 }
```

Traverse the list with iter and put each node in front of head node. The new node become the head node.
Test what happenes if you don’t have line 62.

```
Node* reverseListRecursively(Node *current){
        if(current == NULL || current->next == NULL) return current;

        Node *revHead = reverseListRecursively(current->next);
        current->next->next = current;
        current->next = NULL;
        return revHead;
}
```

Put pen on paper and go through these operations.
Remember how recursive calls work.
Assume the nodes are in a stack with the first node at the bottom of the stack and the last at the top.

```c

int main(int argc, char **argv){
        Node *head;
        char *op = argv[1];
        char *inputFilePath = argv[2];

        head = generateList(inputFilePath);

        if(!strcmp(op, "G")){
                printList(head);
                head = freeList(head);
        return 0;
        }

        if(!strcmp(op, "R")){
                head = reverseListRecursively(head);
//              head = reverseListIteratively(head);
                printList(head);
                head = freeList(head);
        return 0;
        }

return 0;
}
```

Not much.