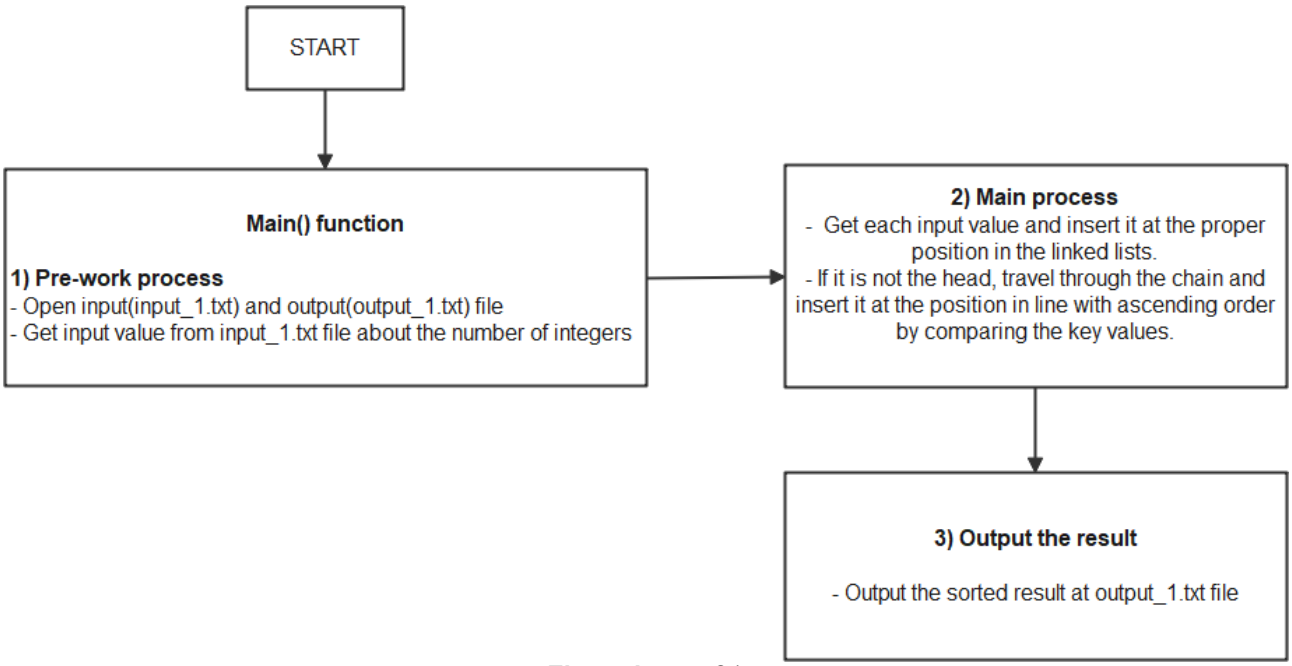
Assignment #5

20190741 김 지수

Q1) Sorting with linked lists

Q1-1. Code Description with Flow chart

As the program starts, Pre-work process begins. The program opens input and output files. Next, it gets the input value from input_1.txt file about the number of integers. On the main process, using the loop, the program gets each input value and insert it at the proper positio n in the linked lists to lively sort the given value in the ascending order. Finally the program outputs the sorted result using linked lists at output_1.txt file.



<Flow chart, Q1>

Q1-2. Main Algorithms with Pseudo code

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

typedef struct list_node *list_pointer;
typedef struct list_node {
        int key;
        list_pointer link;
};

int main() {
        FILE *input, *output;

        int i, num, item;
        int data[3];
        list_pointer temp, head, curr, trail;

        input = fopen("input_1.txt", "r");
        if (input == NULL) {
```

```
                printf("File open error");
                exit(1);
        }

        output = fopen("output_1.txt", "w");
        if (output == NULL) {
                printf("File open error");
                exit(1);
        }
        head = (list_pointer)malloc(sizeof(struct list_node));
        /*malloc function error handling*/
        if (!head) {
                fprintf(stderr, "Memory allocation error! Program terminates");
                exit(1);
        }
        head->link = NULL;

        fscanf(input, "%d\n", &num);
        for (i = 0; i < num; i++) {
                temp = (list_pointer)malloc(sizeof(struct list_node));
                /*malloc function error handling*/
                if (!temp) {
                        fprintf(stderr, "Memory allocation error! Program terminates");
                        exit(1);
                }

                fscanf(input, "%d", &item);
                temp->key = item;
                temp->link = NULL;
                if (!(head->link)) {
                        head->link = temp;
                }
                else {
                        /*searching its correct position based on the size of the key*/
                        curr = head->link;
                        trail = head;
                        while (1) {
                                if (!curr) {
                                /*no number left to compare. input number's position is at the last*/
                                        trail->link = temp;
                                        break;
                                }

                                if (temp->key > curr->key) {
                                        trail = curr;
                                        curr = curr->link;
                                }
                                else if (temp->key <= curr->key) {
                                        trail->link = temp;
                                        temp->link = curr;
                                        break;
                                }
                        }
                }
```

```
        }

        curr = head->link;
        while (curr) {
                fprintf(output, "%d ", curr->key);
                curr = curr->link;
        }
        fclose(input);
        fclose(output);

        return 0;
}
```

<Pseudocode, Q1>

In this program, there is no such specific algorithm to explain. Thus, the explanations about operations are given at the upper <Pseudocode, Q1> using /*explanation*/.

Q1-3. Test cases simulated

input_1.txt	output_1.txt
5 -1 5 3 4 0	-1 0 3 4 5
4 4 2 1 3	1 2 3 4
6 5 6 7 1 2 3	1 2 3 5 6 7
10 0 -999 123 21 1 2 3 3 3 999	-999 0 1 2 3 3 3 21 123 999
1 9	9
0	
5 9 4 7 2 1	1 2 4 7 9
10 42 19 73 58 91 33 64 77 28 55	19 28 33 42 55 58 64 73 77 91

Q2) Maze searching with doubly linked lists

Note:

1) The program uses file input/output with some requirements. Number of input rows and columns both is assumed as 10 in line with the input explanation at PDF file.

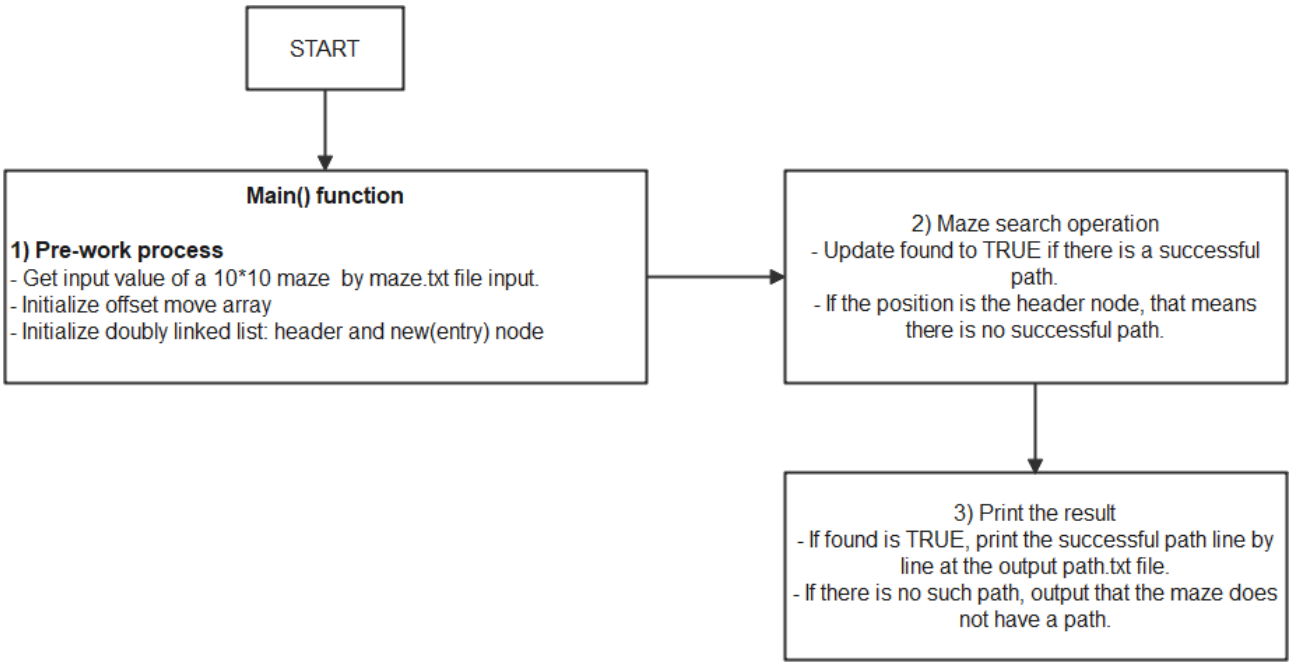
2) Note that the result of path could be different with others. Since the student does not want to allow the inefficiency of algorithm that searches maze, the searching order of direction is modified. It is inefficient to search North and Northeast first in the maze searching as it is opposite way from the destination, the program searches the East position first and Northeast position at last.

3) The main algorithm used in this program is almost the same as that of assignment #3. Only data structure used is different. Thus the explanation is based on the assignment #3.

4) The output requirement said the row and col results should be spaced. As it is hard to see the number of space, the results are outputted with proper space in this program.

Q2-1. Code Description with Flow chart

The flow chart of the Q2 is as follows. When the program starts, the main() function gets file input about each element. In this process, the mark matrix is initialized. After the pre-work process, the program searches maze whether there is a successful path and prints the result accordingly at the 'path.txt' file. Each detailed information about the searching process will be explained at Q2-2 with pseudo code.



<Flow chart, Q2>

Q2-2. Main Algorithms with Pseudo code

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#define MAX_ROW 10
#define MAX_COL 10
#define TRUE 1
#define FALSE 0

typedef struct {
        short int vert;
        short int horiz;
}offset;
offset move[8];

typedef struct {
        short int row;
        short int col;
        short int dir;
}element;

typedef struct node* nodePointer;
typedef struct node {
        nodePointer next_link;
        element data;
        nodePointer before_link;
};

void main() {
        FILE *input, *output;
        int i, j, row = 10, col = 10, nextRow, nextCol, dir, found = FALSE, item;
        int exitRow = 8, exitCol = 8;
        int maze[MAX_ROW][MAX_COL];
        int mark[MAX_ROW][MAX_COL];
        nodePointer head, position, new, print, temp;
        /* Input file open */
        input = fopen("maze.txt", "r");
        if (input == NULL) {
                printf("File open error");
                exit(1);
        }
        /* Output file open */
        output = fopen("path.txt", "w");
        if (output == NULL) {
                printf("File open error");
                exit(1);
        }
        /*maze input process*/
        for (i = 0; i < row; i++) {
            for (j = 0; j < col; j++) {
                fscanf(input, "%d", &item);
                    if (item == 0 || item == 1) {
                        maze[i][j] = item;
                        mark[i][j] = 0;
```

```
                }
                else {
                        fprintf(stderr, "Input maze element is out of bound(0 or 1). Program terminates.\n");
                        exit(1);
                }
        }
}
fclose(input);

/* more greedy version of move */
move[0].vert = 0; move[0].horiz = 1;     // East
move[1].vert = 1; move[1].horiz = 1;     // SE
move[2].vert = 1; move[2].horiz = 0;     // South
move[3].vert = 1; move[3].horiz = -1;     // SW
move[4].vert = 0; move[4].horiz = -1;     // West
move[5].vert = -1; move[5].horiz = -1;     // NW
move[6].vert = -1; move[6].horiz = 0;     // North
move[7].vert = -1; move[7].horiz = 1;     // NE
mark[1][1] = 1;

head = (nodePointer)malloc(sizeof(struct node));
if (!head) {
        fprintf(stderr, "Memory is Full. Program terminates.\n");
        exit(1);
}
new = (nodePointer)malloc(sizeof(struct node));
if (!new) {
        fprintf(stderr, "Memory is Full. Program terminates.\n");
        exit(1);
}

/*initial doubly linked lists setting: header and new(entry) node*/
head->data.row = -1;  head->data.col = -1;  head->data.dir = 10;
new->data.row = 1; new->data.col = 1; new->data.dir = 0;
head->before_link = new; head->next_link = new;
new->before_link = head; new->next_link = head;
position = head->next_link;

while (position is not the header node || the path is not found ) {
        /*delete current position and update position to its before_link*/
        position->before_link->next_link = position->next_link;
        position->next_link->before_link = position->before_link;
        row = position->data.row; col = position->data.col; dir = position->data.dir;
        temp = position;
        position = position->before_link;
        free(temp);

        while (there are more moves from current position) {
                /* move in direction dir */
                nextRow = row + move[dir].vert;
                nextCol = col + move[dir].horiz;
                if (next position is the destination && next position is possible to move on)
                        found = TRUE;
```

```
                    else if (next position is legal move and haven't been there) {
                        mark[nextRow][nextCol] = 1;
                        new = (nodePointer)malloc(sizeof(struct node));
                        if (!new) {
                            fprintf(stderr, "Memory is Full. Program terminates.\n");
                            exit(1);
                        }
                        /*Save current position by inserting new node*/
                        new->data.row = row; new->data.col = col; new->data.dir = ++dir;
                        new->before_link = position; new->next_link = position->next_link;
                        position->next_link->before_link = new;
                        position->next_link = new;
                        position = new;
                        row = nextRow; col = nextCol; dir = 0;
                    }
                    else ++dir;
                }
            }
            if (there is a successful path) {
                output the successful path at output file;
            }
            else output that the maze does not have a path;          fclose(output);
}
```

<Pseudocode, Q2>

```
typedef struct {
        short int row;
        short int col;
        short int dir;
}element;
typedef struct node* nodePointer;
typedef struct node {
        nodePointer next_link;
        element data;
        nodePointer before_link;
};
```

The upper code is the defined data structures used in this program. As the question required to define doubly linked list structure for updating paths, the program defined the node that has before_link, and next_link to move both direction using nodePointer type links and has element data to record node's data of row, col, dir.

As this program is coded based on the maze searching program in the course material, the algorithm explanation is added about the different part compared to the original function. At the original function, the function says there is a successful path before the checking process, that is, whether the maze[nextRow][nextCol] is possible to move on. This could bring a serious problem. Thus, in this program the first if statement in the while loop checks whether the next position is the final destination that we could move on.

In addition, this program is coded with mind that only data structure that is used to implement the path result should be changed from stacks to doubly linked list. As a result, the checking process is independent of data structure but is only implemented with doubly linked list to meet the assignment requirement. Nothing special!

As a result, the program operates correctly and the results are shown in the next part.

Q2-3. Test cases simulated

In order to show the output result within the page, the divided output cell is not in a same line. It means, each path <row, col> element will be output in the file line by line. Note that a path result start at <1, 1> and ends at <8, 8> as the maze size is fixed, if successful path exi sts, in accordance with the output example.

<Test results, Q2>

maze.txt	path.txt
1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 0 0 0 1 0 1 1 1 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 0 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1	1 1 2 2 2 3 2 4 3 5 3 6 4 5 5 6 5 7 5 8 6 7 7 8 8 8
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1	The maze does not have a path.
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 0 1 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1	1 1 1 2 1 3 1 4 1 5 1 6 1 7 2 8 3 7 3 6 3 5 3 4 3 3 3 2 4 1 5 2 6 1 7 2 7 3 8 4 7 5 7 6 8 7 8 8

As simulating the many test cases like upper table, the program could be more reliable by checking the corner cases. As a result, there is no program error or incorrect result found for several test cases.

Q3) SNS friend management program

notes)

1. Question note said to use push(), and pop() function while there is no comment for using s tacks to implement this problem. As everyone knows, if A is friend with B and B is friend wit h C does not means that A is friend with C. The output example for this question is in accor dance with this principle. If not, all of the Sogang University students are my friends! So, the re is no transitivity characteristic in this problem, and thus there is no need of using stacks i n our equivalent classes problem. As a result, the student understood that push() and pop() f unction means, it is required to implement through functions when the program Adds or Rem oves one's friend. The program implemented these features using pushFriend(), and popFriend () functions and it actually works similarly what push(), and pop() does for managing a stack. In short, pushFriend() and popFriend() are push() and pop() function for our data structures.

2. Assume that user inputs less than 50 characters of each name.

Q3-1. Code Description with Flow chart

The flow chart of the Q3 is as follows. When the program starts, main function gets user in puts, string. If the string is out of bound, the program outputs error message and terminates. Otherwise, if the input is valid, 3 Functions are called sequentially as seen in the <Flow chart, Q2> below. Specific information will be introduced at Q3-2 part.

<Flow chart, Q3>

Q3-2. Main Algorithms with Pseudo code

In this program, there are many functions that is purposely made for structured programmi ng. So, in this part, each function is explained below.

Function Explanations
personPointer createPerson(char* name) Function createPerson creates a new person instance that has a person name, and his/her friend list. Friend list, friendPointer friends, is initially set to NULL. This function is classified as Creator function according to our course materials.
personPointer findPerson(personPointer* people, int numPeople, char* name) Function findPerson searches if the given name is already in the user list. A double pointer people is the list of people that has already input. If the given name is already in the list then return the personPointer that has the same name. Else there is no such name, return NULL. This logic is made to check the presence of the input name that is widely used in this program.
void pushFriend(personPointer person, char* name) Function pushFriend pushes friend relationship by linking new friend at the friend list of the given person. In order not to add the same friend, the program handles this problem when it calls the function by using areFriends function.
void popFriend(personPointer person, char* name) Function popFriend pops friend relationship by unlinking the given friend from the friend list of the given person. When two people are not friend, nothing happens in the popFriend process because they are not friend! As they were not friends poping process does not influence the friend list of the given person.
void printFriends(personPointer person, FILE* outputFile)

Function printFriends outputs friend list of the given person at the given output file. Unfortunately, if the given person has no friend, the program outputs that there is no friend for the notice purpose, not teasing him/her.
int areFriends(personPointer person1, personPointer person2) Function areFriends returns the result about whether two people are friends or not. As the function return integer value, this function is widely used for checking their relationship.

The pesudocode of Q3 is the runnable code with /*explanations*/. As the main operation is u sing linked lists and manage their friends list by person, there is no such specific explainable algorithm but there are functions and some operations. As a result, used functions are explai ned before and the explanations about the meaningful operations are explained as belows.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_NAME_LENGTH 50

typedef struct FriendNode {
    char name[MAX_NAME_LENGTH];
    struct FriendNode* next;
} FriendNode, *friendPointer;

typedef struct Person {
    char name[MAX_NAME_LENGTH];
    friendPointer friends;
} Person, *personPointer;

personPointer createPerson(char* name) {
    personPointer newPerson = (personPointer)malloc(sizeof(Person));
    if (!newPerson) {
        fprintf(stderr, "Memory Allocation error. Program terminates.\n");
        exit(1);
    }
    strcpy(newPerson->name, name);
    newPerson->friends = NULL;
    return newPerson;
}

personPointer findPerson(personPointer* people, int numPeople, char* name) {
    for (int i = 0; i < numPeople; i++) {
        if (strcmp(people[i]->name, name) == 0) {
            return people[i];
        }
    }
    return NULL;
}

void pushFriend(personPointer person, char* name) {
    friendPointer newFriend = (friendPointer)malloc(sizeof(FriendNode));
    if (!newFriend) {
        fprintf(stderr, "Memory Allocation error. Program terminates.\n");
```

```
            exit(1);
        }
        strcpy(newFriend->name, name);
        newFriend->next = NULL;

        /*push new friend into the friend list of that person*/
        if (person->friends == NULL) {
            person->friends = newFriend;
            return;
        }
        friendPointer currentFriend = person->friends;
        while (currentFriend->next != NULL) {
            currentFriend = currentFriend->next;
        }
        currentFriend->next = newFriend;
}

void popFriend(personPointer person, char* name) {
        if (person->friends == NULL) {
            return;
        }
        if (strcmp(person->friends->name, name) == 0) {
            friendPointer temp = person->friends;
            person->friends = person->friends->next;
            free(temp);
            return;
        }
        friendPointer prevFriend = person->friends;
        friendPointer currentFriend = person->friends->next;
        while (currentFriend != NULL && strcmp(currentFriend->name, name) != 0) {
            prevFriend = currentFriend;
            currentFriend = currentFriend->next;
        }
        if (currentFriend != NULL) {
            prevFriend->next = currentFriend->next;
            free(currentFriend);
        }
}

void printFriends(personPointer person, FILE* outputFile) {
        friendPointer currentFriend = person->friends;
        if (!currentFriend) {
            fprintf(outputFile, "There is no friend...");
        }
        while (currentFriend != NULL) {
            fprintf(outputFile, "%s ", currentFriend->name);
            currentFriend = currentFriend->next;
        }
        fprintf(outputFile, "\n");
}

int areFriends(personPointer person1, personPointer person2) {
        friendPointer currentFriend = person1->friends;
        while (currentFriend != NULL) {
```

```
            if (strcmp(currentFriend->name, person2->name) == 0) {
                return 1;
            }
            currentFriend = currentFriend->next;
        }
        return 0;
}

int main() {
        FILE* input = fopen("input_3.txt", "r");
        FILE* output = fopen("output_3.txt", "w");

        if (input == NULL) {
            fprintf(stderr, "Failed to open input file. Program terminates.\n");
            exit(1);
        }
        if (output == NULL) {
            fprintf(stderr, "Failed to open output file. Program terminates.\n");
            exit(1);
        }

        int numPeople = 0;
        personPointer people[100000], person1, person2, person;
        char command;
        char name1[MAX_NAME_LENGTH];
        char name2[MAX_NAME_LENGTH];

        while (fscanf(input, "%c", &command) == 1) {
            if (command == 'P') {
                fscanf(input, " %s", name1);
                /*Check if there is a same name*/
                if (findPerson(people, numPeople, name1) != NULL) {
                    fprintf(output, "A person with the same name already exists.\n");
                    continue;
                }
                people[numPeople] = createPerson(name1);
                numPeople++;
            }
            else if (command == 'F') {
                fscanf(input, " %s %s", name1, name2);
                person1 = findPerson(people, numPeople, name1);
                person2 = findPerson(people, numPeople, name2);
                /*Get two people to add their friendship. If there is no such person notice and continue*/
                if (person1 == NULL || person2 == NULL) {
                    fprintf(output, "One or both people do not exist.\n");
                    continue;
                }
                /*push their friendship onto both friendlists*/
                if (!(areFriends(person1, person2))) {
                    pushFriend(person1, name2);
                    pushFriend(person2, name1);
                }
            }
            else if (command == 'U') {
```

```
            fscanf(input, " %s %s", name1, name2);
            person1 = findPerson(people, numPeople, name1);
            person2 = findPerson(people, numPeople, name2);
            /*if there is no such person to unfriend, program tells explicity. When two people are not friend,
              nothing happens in the popFriend process because they are not friend!*/
            if (person1 == NULL || person2 == NULL) {
                fprintf(output, "One or both people do not exist.\n");
                continue;
            }
            popFriend(person1, name2);
            popFriend(person2, name1);
        }
        else if (command == 'L') {
            fscanf(input, " %s", name1);
            person = findPerson(people, numPeople, name1);
            if (person == NULL) {
                fprintf(output, "Person does not exist.\n");
                continue;
            }
            printFriends(person, output);
        }
        else if (command == 'Q') {
            fscanf(input, " %s %s", name1, name2);
            person1 = findPerson(people, numPeople, name1);
            person2 = findPerson(people, numPeople, name2);
            if (person1 == NULL || person2 == NULL) {
                fprintf(output, "One or both people do not exist.\n");
                continue;
            }
            /*check their friendship and output the result*/
            if (areFriends(person1, person2)) {
                fprintf(output, "Yes\n");
            }
            else {
                fprintf(output, "No\n");
            }
        }
        else if (command == 'X') {
            break;
        }
    }
    fclose(input);
    fclose(output);
    return 0;
}
```

<Pseudocode, Q3>

Q3-3. Test cases simulated

As simulating the test cases could make the program more reliable by finding the corner cas es, cases are tested. As a result, there was no program error or incorrect result found for se veral test cases.

input_3.txt	output_3.txt
P Sam P Liza P Mark P Amy F Liza Amy F Liza Mark F Amy Sam L Amy L Sam U Liza Amy L Amy Q Liza Mark X	Liza Sam Amy Sam Yes
P John P Alice P Mike F John Alice F Alice Mike F John Mike L John L Alice L Mike Q John Alice Q John Mike Q Alice Mike U John Alice L John L Alice L Mike X	Alice Mike John Mike Alice John Yes Yes Yes Mike Mike Alice John
P Sam P Liza P Mark P Amy F Liza Amy F Liza Amy F Liza Amy F Liza Amy F Liza Mark F Amy Sam L Amy L Sam U Liza Amy L Amy Q Liza Mark X	Liza Sam Amy Sam Yes

P Sam P Liza P Mark P Amy F Liza Amy F Liza Mark F Amy Sam F Amy Sam P John P Jay P Sam F John Amy F John Mark F Jay Liza L Mark L Amy L Sam U Liza Amy U Liza John U Amy John L Amy L Jay L John L Mark Q Liza Mark Q Amy John Q Jay John Q Liza Jay X	A person with the same name already exists. Liza John Liza Sam John Amy Sam Liza Mark Liza John Yes No No Yes
P Sam P Liza P Mark P Amy F Liza Amy F Liza Mark F Amy Sam F Amy Sam P John P Jay P Sam F John Amy F John Mark F Jay Liza L Mark L Amy L Sam U Liza Amy U Liza John U Amy John L Amy L Jay L John L Mark Q Liza Mark Q Amy John Q Jay John Q Liza Jay U Liza Jay L Jay X	A person with the same name already exists. Liza John Liza Sam John Amy Sam Liza Mark Liza John Yes No No Yes There is no friend...