Data Structure Task 5 Report

20200901 이효주

Problem1.

1. Problem definition)

The user must receive the number of numbers to be stored in the linked list and the number and sort them in ascending order.

2. Pseudo code & Approach)

(1) Main function

// The main function reads integers from a file and stores them in a singly linked list
// The linked list is then sorted in ascending order and written to an output file

```
int main() {
	int n, i, data;
	list_node* head = NULL;
	FILE* in_file, * out_file;

	// Read input from the input file
	in_file = fopen("input_1.txt", "r");
	out_file = fopen("output_1.txt", "w");

	// Number of integers to be entered
	fscanf(in_file, "%d", &n);

	for (i = 0; i < n; i++) {
		fscanf(in_file, "%d", &data);
		// add in node for each integer
		head = add_node(head, data);
	}

	// Sort the linked list in ascending order
	sort_list(head);

	// Write the sorted linked list to the output file
	list_node* curr_node = head;
	while (curr_node != NULL) {
		fprintf(out_file, "%d ", curr_node->data);
		curr_node = curr_node->link;
	}
```

```
    // Free the memory allocated for the linked list
    fclose(in_file);
    fclose(out_file);
    free_list(head);

    return 0;
}
```

(2) Struct of the linked list

```
// Define a structure for a node of the singly linked list
typedef struct list_node {
        int data;
        struct list_node* link;
}list_node;
```

(3) Add node Function

```
// Function to add a node to the end of the linked list
list_node* add_node(list_node* head, int data) {

    // Allocate memory for the new node
    list_node* new_node = (list_node*)malloc(sizeof(list_node));
    new_node->data = data;
    new_node->link = NULL;

    // If the linked list is empty, set the new node as the head
    if (head == NULL) {
        head = new_node;
    }

    else {
        // Traverse the linked list to find the last node
        list_node* curr_node = head;
        while (curr_node->link != NULL) {
            curr_node = curr_node->link;
        }
        // Set the new node as the last node
        curr_node->link = new_node;
    }

    return head;
}
```

(4) Sort Function

```
// Function to sort the linked list in ascending order
void sort_list(list_node* head) {

    // If the given linked list is empty, simply return without doing anything
    if (head == NULL) {
        return;
    }

    // Loop through the linked list twice, comparing each pair of adjacent nodes and
swapping them if they are in the wrong order
    list_node* i, * j;
    int temp;
    for (i = head; i != NULL; i = i->link) {
        for (j = i->link; j != NULL; j = j->link) {
            if (i->data > j->data) {
                // Swap the data values of the nodes, to sort the linked list in ascending
order
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}
```

(5) Free Function

```
// Free the memory allocated for each node in the linked list
void free_list(list_node* head) {

    // Initialize a pointer to the current node as the head node
    list_node* curr_node = head;

    // Continue until the current node becomes NULL
    while (curr_node != NULL) {
        // Create a temporary pointer to the current node
        list_node* temp_node = curr_node;

        // Move the current node pointer to the next node
        curr_node = curr_node->link;

        // Free the memory allocated to the temporary node pointer
        free(temp_node);
```

```
            }
        }
```

add_node function:

Use the head pointer and data received as parameters to add a new node to the end of the connection list.
Allocates dynamic memory for the list_node structure, saves data, and initializes the link on the node to NULL.
If the connection list is empty, set the new node as the head.
Otherwise, it will tour the connection list all the way to find the last node, followed by a new node.
Returns the changed head pointer.

sort_list function:

Use the head pointer received as a parameter to sort the list of connections in ascending order.
If the given link list is empty, do nothing and exit the function.
Use a double iteration statement to tour all nodes and compare adjacent node pairs.
If the data on the next node, j, is larger than the data on the next node, then the data on the two nodes is exchanged to perform an ascending sort.

Free_list function:

Use the head pointer received as a parameter to release the memory allocated to each node in the connection list.
Initializes the pointer curr_node pointing to the current node.
Repeat until curr_node is NULL.
Save the current node to the temporary pointer temp_node, and move curr_node to the next node.
Release the memory allocated to the node that temp_node points to.

- 1. Use the add_node function to add an integer read from the input file to the end of the link list. This configures all integers read from the input file into a linked list.

  2. Sort the list of associations in ascending order using the sort_list function. Use a double iteration statement to compare adjacent node pairs, and to exchange data values to sort them. This process aligns the nodes in the connection list.

  3. The main function reads integers from the input file through file input and output to form a list of connections. Then, call the sort_list function to sort the list of connections.

  4. Finally, write the data values of the nodes in the sorted connection list to the output file to release the connection list. For this purpose, data values for each node are written to the output file while traversing the connection list. It then invokes the free_list function to release the dynamically allocated memory.

This is a simple sorting algorithm that saves integers contained in an input file as a linked list, sorts them, and writes them to an output file.

Input_1.txt	Output_1.txt
6 5 6 7 1 2 3	1 2 3 5 6 7

Problem2.

1. Problem definition)

A maze surrounded by 1 of 10*10 should be double linked list to store information on the path, and if there is an exit, mark and the corresponding path should be printed out.
The maze should be entered as a maze.txt file, and the output of path should be output as path.txt.

2. Pseudo code & Approach)

- This code uses a Depth-First Search (DFS) algorithm to solve the problem of finding a route from a maze to a destination.

  The main function reads maze data from a file and stores it in a maze array. and initialize the mark array and path_list.

  The dfs function is recursively called and finds its path by moving up, down, left, right, and diagonal from its current position.

  Within the dfs function, the current location is indicated by a visit to the mark array. If the current location is the destination, call the addNode function to add the current location to the path_list.

  Determine where you can move up, down, left, and right diagonally from the current position. The moveable location is where the value of the maze array is 0, which is not visited in the mark array.

  If you find a moveable location, call the addNode function to add the current location to the path_list, and recursively call the dfs function to that location.

  If the last node on the path_list reaches its destination (path_list.tail->row == EXIT_ROW and path_list.tail->col == EXIT_COL), abort further navigation.

  Otherwise, call the pop function to remove the last node from the path_list and return to the previous location. Use this to explore other paths.

  When the dfs function ends, path_list stores the path from origin to destination.

The main function traverses the path_list and outputs the path to a file.

The algorithm uses DFS to explore every possible path in the maze and find a path to the destination. Use the mark array to track where you have already visited, and use the path_list to save the path. DFS works by exploring possible paths in a maze to find a route from origin to destination in a depth-first manner, and by returning to the previous position and exploring another route if the wrong path is encountered.

➢

```
#define MAX_ROW 32
#define MAX_COL 32
#define TRUE 1
#define FALSE 0

/* define stack structure */
typedef struct Node {
        int row;
        int col;
        struct Node* prev;
        struct Node* next;
} Node;

typedef struct {
        Node* head;
        Node* tail;
} DoublyLinkedList;

typedef struct {
        int vert;
        int horiz;
} offsets;

offsets move[8] = { {-1,0}, {-1,-1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1}, {-1,-1} };

int EXIT_ROW = 8, EXIT_COL = 8;
int maze[MAX_ROW][MAX_COL];
int mark[MAX_ROW][MAX_COL];
DoublyLinkedList path_list;
```

initializeList: Function that initializes a new double-connect list. Sets the head and tail of the list to NULL.

```
        /* initialize a new doubly linked list */
void initializeList(DoublyLinkedList* list) {
        list->head = NULL;
        list->tail = NULL;
```

```
}
```

addNode: Function that adds a new node to a given dual-connect list. Sets the coordinates (row, col) of the new node and the old node (prev), and updates the association with the tail of the list.

```
/* add a new node to the end of the doubly linked list */
void addNode(DoublyLinkedList* list, int row, int col) {
	Node* new_node = (Node*)malloc(sizeof(Node));
	new_node->row = row;
	new_node->col = col;
	new_node->prev = list->tail;
	new_node->next = NULL;
	if (list->head == NULL) {
		list->head = new_node;
	}
	else {
		list->tail->next = new_node;
	}
	list->tail = new_node;
}
```

pop: Function that removes and returns the last node from the double-connect list. If the list is empty, output an error message and exit the program.

```
/* define function to pop element from the end of doubly linked list */
Node* pop() {
	if (path_list.head == NULL) {
		fprintf(stderr, "List is empty\n");
		exit(EXIT_FAILURE);
	}
	Node* temp = path_list.tail;
	path_list.tail = temp->prev;
	if (path_list.tail == NULL) {
		path_list.head = NULL;
	}
	else {
		path_list.tail->next = NULL;
	}
	return temp;
}
```

dfs: Function that performs Depth-First Search. Locate the path by moving up, down, left, and right diagonally from the current position. When the escape point is reached, addNode is called to store the path and dfs is called recursively. At this point, if the escape point is reached, further exploration is interrupted. If the path is incorrect, use pop to return to the previous location.

```
void dfs(int row, int col) {
    int i, nextRow, nextCol;

    mark[row][col] = 1;
    if (row == EXIT_ROW && col == EXIT_COL) {
        addNode(&path_list, row, col);
        return;
    }

    for (i = 0; i < 8; i++) {
        // move in direction i
        nextRow = row + move[i].vert;
        nextCol = col + move[i].horiz;
        if (!maze[nextRow][nextCol] && !mark[nextRow][nextCol]) {
            addNode(&path_list, row, col);
            dfs(nextRow, nextCol);
            if (path_list.tail->row == EXIT_ROW && path_list.tail->col == EXIT_COL) {
                return;
            }
            pop();
        }
    }
}
```

main: The entry point for the program. Reads maze data from the file and stores it in a maze array. Initialize the mark array and path_list. Then, make sure that the origin and destination are valid paths, and call the dfs function to find the path. Travers the path_list to write the path to the file and prints the results to the file.

```
int main() {
    int i, j;
    FILE* fp = fopen("maze.txt", "r");
    if (fp == NULL) {
        printf("Failed to open file.\n");
        return 1;
    }

    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            fscanf(fp, "%d", &maze[i][j]);
```

```
            }
        }
        fclose(fp);

        // Initialize the mark and path list
        for (i = 0; i < MAX_ROW; i++) {
            for (j = 0; j < MAX_COL; j++) {
                mark[i][j] = 0;
            }
        }
        initializeList(&path_list);

        // Check if the maze has a valid path
        if (maze[1][1] != 0 || maze[EXIT_ROW][EXIT_COL] != 0) {
            printf("The maze does not have a valid path.\n");
            return 0;
        }

        // Find the path using depth-first search (DFS)
        dfs(1, 1);

        // Write the path to the file
        FILE* path_fp = fopen("path.txt", "w");
        if (path_fp == NULL) {
            printf("Failed to create path file.\n");
            return 1;
        }

        Node* current = path_list.head;
        while (current != NULL) {
            fprintf(path_fp, "%d %d\n", current->row, current->col);
            current = current->next;
        }
        fclose(path_fp);

        return 0;
}
```

Maze.txt	Path.txt
1 1 1 1 1 1 1 1 1 1	1 1
1 0 1 1 1 1 1 0 1 1	2 2
1 1 0 0 0 1 0 1 1 1	2 3

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	2 4 3 5 3 6 4 5 5 6 5 7 5 8 6 7 7 8 8 8

Problem3.

1. Problem definition)

You should write a program that manages SNS friends using equivalence classes.
command:
P <name> : Save by specifying a specific name. Care should be taken not to register a person with the same name.
F <name1> <name2>: Record that two specific people are friends.
U <name1> <name2>: Record that a particular two people are no longer friends.
L <name> : Prints a list of friends of a particular person.
Q <name1> <name2>: Determine if a particular two people are friends. If you're a friend, say "Yes" or "No"
I'm going to print it out.
X: Exit the program

2. Pseudo code & Approach)

This code implements a simple social network system that manages friendship between people. The following is a detailed description of the key functions and how this code works:

main: Read the command from the input file and call the appropriate function according to each command. Follow the command to add people, form friendships, print out a list of friends, or check if they are friends. Open and close the input and output files.

```
int main() {
    FILE* inputFile = fopen("input_3.txt", "r");
    FILE* outputFile = fopen("output_3.txt", "w");
    char command[10];
    char name1[MAX_NAME_LENGTH];
    char name2[MAX_NAME_LENGTH];

    // Initialize people array
    for (int i = 0; i < MAX_PEOPLE; i++) {
        people[i] = NULL;
```

```
    }

    while (fscanf(inputFile, "%s", command) == 1) {
        if (strcmp(command, "P") == 0) {
            fscanf(inputFile, "%s", name1);
            push(name1);
        }
        else if (strcmp(command, "F") == 0) {
            fscanf(inputFile, "%s %s", name1, name2);
            makeFriends(name1, name2);
        }
        else if (strcmp(command, "U") == 0) {
            fscanf(inputFile, "%s %s", name1, name2);
            removeFriendship(name1, name2);
        }
        else if (strcmp(command, "L") == 0) {
            fscanf(inputFile, "%s", name1);
            printFriendList(name1, outputFile);
        }
        else if (strcmp(command, "Q") == 0) {
            fscanf(inputFile, "%s %s", name1, name2);
            checkFriendship(name1, name2, outputFile);
        }
        else if (strcmp(command, "X") == 0) {
            break;
        }
    }

    fclose(inputFile);
    fclose(outputFile);

    return 0;
}
```

Node structure: Node for storing a list of friends as a linked list. Each node contains a name (name) and a pointer (next) pointing to the next node.

```
#define MAX_NAME_LENGTH 50
#define MAX_PEOPLE 100

typedef struct Node {
    char name[20];
    struct Node* next;
} Node;

Node* people[MAX_PEOPLE];
```

createNode: A function that creates a new node with a given name. Dynamically allocates memory, copies the name, initializes the node, and returns it.

```
// Create a new node
Node* createNode(const char* name) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    strcpy(newNode->name, name);
    newNode->next = NULL;
    return newNode;
}
```

findPerson: A function that finds a person with a given name and returns that node. Traverses the people array and compares names to find matching nodes.

```
// Find a person by name
Node* findPerson(const char* name) {
    int i;
    for (i = 0; i < MAX_PEOPLE; i++) {
        if (people[i] != NULL && strcmp(people[i]->name, name) == 0) {
            return people[i];
        }
    }
    return NULL;
}
```

push: A function that adds a person with a given name to the list. After verifying that someone already has the same name, create a new node and add it to the list.

```
// Push a person to the list
void push(const char* name) {
    if (findPerson(name) == NULL) {
        Node* newNode = createNode(name);
        int i;
        for (i = 0; i < MAX_PEOPLE; i++) {
            if (people[i] == NULL) {
                people[i] = newNode;
                return;
            }
        }
    }
    else
```

```
            printf("Already exist");
        printf("Error: Maximum number of people reached.\n");
}
```

pop: A function that removes people with a given name from the list. Use the findPerson function to find the node, then release it and remove it from the list.

```
// Pop a person from the list
void pop(const char* name) {
        Node* person = findPerson(name);
        if (person != NULL) {
                int i;
                for (i = 0; i < MAX_PEOPLE; i++) {
                        if (people[i] == person) {
                                people[i] = NULL;
                                free(person);
                                return;
                        }
                }
        }
}
```

makeFriends: A function that registers two given people as friends. Create friendships by adding new friends to each person's node.

```
// Make two people friends
void makeFriends(const char* name1, const char* name2) {
        Node* person1 = findPerson(name1);
        Node* person2 = findPerson(name2);

        if (person1 != NULL && person2 != NULL) {
                Node* newname1 = createNode(name1);
                Node* newname2 = createNode(name2);

                newname1->next = person2->next;
                newname2->next = person1->next;

                person1->next = newname2;
                person2->next = newname1;
        }
}
```

removeFriendship: A function that removes a friend relationship between a given person. Locate the opponent's

name in each person's Friends list and remove the node.

```
// Remove friendship between two people
void removeFriendship(const char* name1, const char* name2) {
    Node* person1 = findPerson(name1);
    Node* person2 = findPerson(name2);
    if (person1 != NULL && person2 != NULL) {
        Node* prev1 = NULL;
        Node* prev2 = NULL;

        Node* current1 = person1->next;
        Node* current2 = person2->next;

        while (current1 != NULL && current2 != NULL) {
            if (strcmp(current1->name, name2) == 0) {
                if (prev1 == NULL) {
                    person1->next = current1->next;
                }
                else {
                    prev1->next = current1->next;
                }
                free(current1);
                current1 = person1->next;
            }
            else {
                prev1 = current1;
                current1 = current1->next;
            }

            if (strcmp(current2->name, name1) == 0) {
                if (prev2 == NULL) {
                    person2->next = current2->next;
                }
                else {
                    prev2->next = current2->next;
                }
                free(current2);
                current2 = person2->next;
            }
            else {
                prev2 = current2;
                current2 = current2->next;
            }
        }
    }
}
```

printFriendList: A function that outputs a list of friends for a given person. After finding that person's node, it will print out each friend's name while touring the friend list.

```
// Print the friend list of a person
void printFriendList(const char* name, FILE* outputFile) {
    Node* person = findPerson(name);
    if (person != NULL) {
        Node* current = person->next;
        while (current != NULL) {
            fprintf(outputFile, "%s ", current->name);
            current = current->next;
        }
        fprintf(outputFile, "\n");
    }
}
```

checkFriendship: A function that determines whether a given person is a friend. Make sure that the first person's name matches the second person's name by touring the first person's friend list.

```
// Check if two people are friends
void checkFriendship(const char* name1, const char* name2, FILE* outputFile) {
    Node* person1 = findPerson(name1);
    Node* person2 = findPerson(name2);
    if (person1 != NULL && person2 != NULL) {
        Node* current = person1->next;
        while (current != NULL) {
            if (strcmp(current->name, name2) == 0) {
                fprintf(outputFile, "Yes\n");
                return;
            }
            current = current->next;
        }
    }
    fprintf(outputFile, "No\n");
}
```

The main algorithm of this code consists of the following steps:

1. push: function that adds people
Use the findPerson function to determine if a person with the same name already exists.
Create a new node to call the createNode function.
Rotates through the people array to find empty spots and insert new nodes.

2. pop: a function that removes people
Use the findPerson function to find the node of a person with a given name.
Removes the node from the people array and releases memory.

3. makeFriends: A function that forms a friendship
Use the findPerson function to find the nodes of the given two people.
Create a new node and add it to your friends list.

4. removeFriendship: Function that removes friend relationships
Use the findPerson function to find the nodes of the given two people.
Spaces each person's list of friends and removes nodes that match their names.

5. printFriendList: a function that outputs a list of friends
Use the findPerson function to find the nodes of a given person.
Prints a friend's name while touring the list of friends on that node.

6. CheckFriendship: A function to check if you are a friend
Use the findPerson function to find the first person's node.
Make sure that the name matches the second person's name by touring the list of friends on that node.

7. main: main execution function
Read the command from the input file and call the appropriate function.
Invokes functions such as push, makeFriends, removeFriendship, printFriendList, checkFriendship, and so on.

Through each function, the algorithm implements a social network system by performing tasks such as adding, removing people, forming friendships, printing out friends lists, and checking friendships.

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	Sam Liza Amy Sam Yes