자료구조 과제 4 보고서

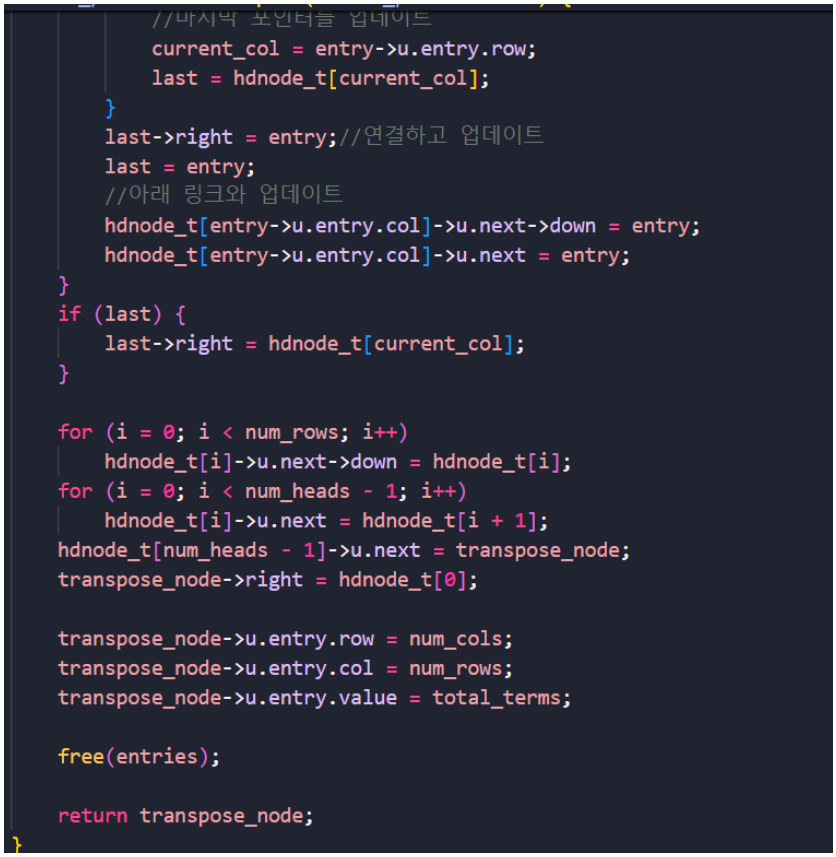
Jumagul Alua, 20231632

1. Matrix transpose

```
matrix_pointer mtranspose(matrix_pointer node) {
    int num_rows = node->u.entry.row;
    int num_cols = node->u.entry.col;
    //number of nodes needed
    int num_heads = (num_cols > num_rows) ? num_cols : num_rows;
    matrix_pointer temp, transpose_node;
    int i, current_col = 0;
    int total_terms = 0;
    matrix_pointer *entries = malloc(num_heads * num_heads * sizeof(matrix_pointer));
    //노드가 0이면 원래꺼 변환
    if (!num_heads){
        return node;
    }
    //새로운 노드 생성
    transpose_node = new_node();
    transpose_node->tag = entry;
    //교환
    transpose_node->u.entry.row = num_cols;
    transpose_node->u.entry.col = num_rows;
    //해더 노드를 초기화
    for (i = 0; i < num_heads; i++) {
        temp = new_node();
        hdnode_t[i] = temp;
        hdnode_t[i]->tag = head;
        hdnode_t[i]->right = temp;
        hdnode_t[i]->u.next = temp;
    }
    matrix_pointer head = node->right;
```

```
    for (i = 0; i < num_rows; i++) {
        for (temp = head->right; temp != head; temp = temp->right) {
            //각 항목에서 행, 열, 값 추출
            int row = temp->u.entry.row;
            int col = temp->u.entry.col;
            int value = temp->u.entry.value;

            matrix_pointer new_entry = new_node();
            new_entry->tag = entry;
            new_entry->u.entry.row = col;
            new_entry->u.entry.col = row;
            new_entry->u.entry.value = value;
            //저강하고 총 항목 수를 업데이트
            entries[total_terms++] = new_entry;
        }
        head = head->u.next;
    }
    //배열을 정렬
    qsort(entries, total_terms, sizeof(matrix_pointer), compare_entries);
    current_col = -1;
    matrix_pointer last = NULL;
    for (i = 0; i < total_terms; i++) {
        matrix_pointer entry = entries[i];
        if (entry->u.entry.row != current_col) {
            //마기막 항모그이 링크를 업데이트
            if (last) {
                last->right = hdnode_t[current_col];
            }
```



```
            //마지막 포인터를 업데이트
            current_col = entry->u.entry.row;
            last = hdnode_t[current_col];
        }
        last->right = entry;//연결하고 업데이트
        last = entry;
        //아래 링크와 업데이트
        hdnode_t[entry->u.entry.col]->u.next->down = entry;
        hdnode_t[entry->u.entry.col]->u.next = entry;
    }
    if (last) {
        last->right = hdnode_t[current_col];
    }

    for (i = 0; i < num_rows; i++)
        hdnode_t[i]->u.next->down = hdnode_t[i];
    for (i = 0; i < num_heads - 1; i++)
        hdnode_t[i]->u.next = hdnode_t[i + 1];
    hdnode_t[num_heads - 1]->u.next = transpose_node;
    transpose_node->right = hdnode_t[0];

    transpose_node->u.entry.row = num_cols;
    transpose_node->u.entry.col = num_rows;
    transpose_node->u.entry.value = total_terms;

    free(entries);

    return transpose_node;
}
```

행렬의 행과 열을 서로 바꾸는 작업을 했다. 예를 들어, m×n 행렬을 전치하면 결과는 n×m 행렬이 된다. 이는 각 원소의 행과 열을 서로 바꿔줬다. 이를 하기 위해 mread 과 mwrite 함수들이 교재에 있었기 때문에 mtronspose 만 찾으면 되었다. 추가적으로 입력은 파일에서 가져왔고 또 다른 파일에 넣었어야 했다.

mtranspose 함수는 희소 행렬을 나타내는 자료구조로서 희소 행렬 표현의 일종인 연결 리스트를 사용하여 전치를 수행한다. 이 함수는 먼저 주어진 행렬을 통해 전치된 행렬을 나타내는 새로운 행렬 노드를 만들고, 원래 행렬의 각 원소를 순회하면서 행과 열을 바꾼 새로운 항목을 생성하여 새로운 행렬에 추가한다.

전치된 행렬을 나타내기 위해 각 행과 열에 대한 헤더 노드를 사용했다. 이 헤더 노드는 각 열의 시작을 가리키고, 각 행은 다음 헤더 노드를 가리킨다.

전체적인 흐름은:

- 주어진 행렬에서 행과 열의 수를 얻는다.

- 전치된 행렬을 나타내는 새로운 행렬 노드를 생성한다.
- 헤더 노드를 초기화하고, 각 열의 시작을 가리키도록 설정한다.
- 원래 행렬을 순회하면서 각 항목을 새로운 행렬에 추가한다. 행과 열을 바꾼다.
- 새로운 행렬의 각 행에 대해 항목을 정렬하고, 헤더 노드를 업데이트하여 각 열의 시작을 설정한다.
- 전치된 행렬 노드의 행과 열 정보를 업데이트한다.
- 동적으로 할당된 메모리를 해제한다.

시간 복잡도는 주어진 행렬의 항목 수에 비례하며, 각 항목을 전치하고 정렬하는 데 O(n log n)의 시간이 소요된다. 여기서 n 은 행렬의 총 항목 수이다. 공간 복잡도는 전치된 행렬의 크기에 비례하여, 일반적으로 희소 행렬에 대한 전치는 전치된 행렬의 항목 수와 유사한 공간을 필요한다.

2. **Polynomials**

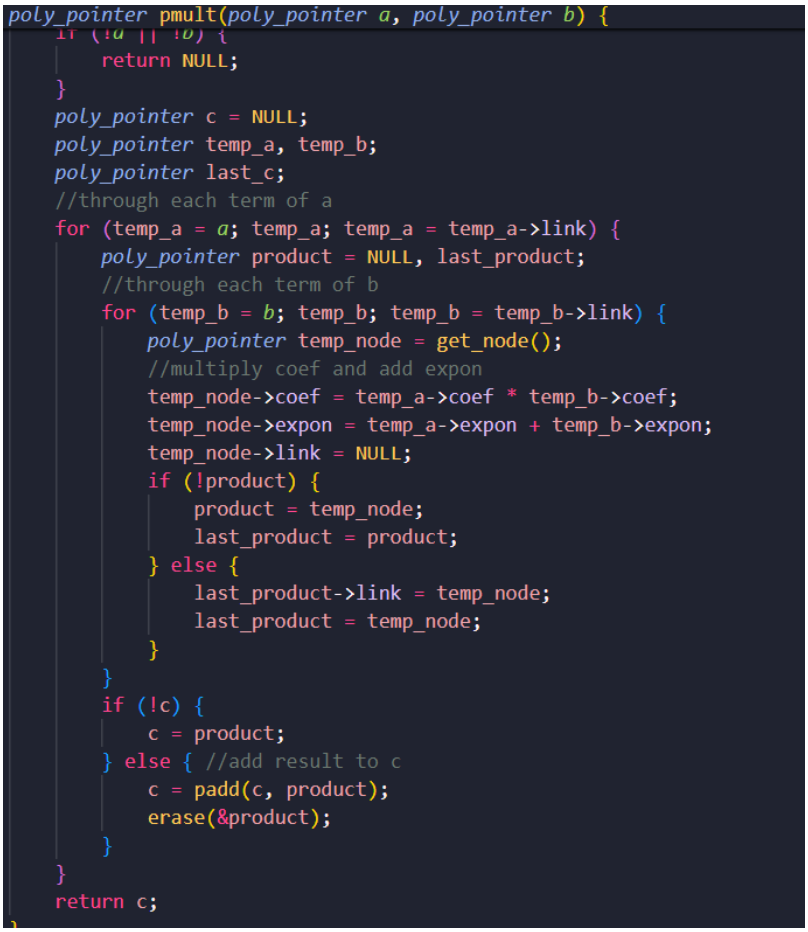
   다항식을 계산하고 다항식의 곱셈을 수행하는 프로그램을 작겅했. 다항식은 연결 리스트를 사용하여 표현된다. 이 연결 리스트는 각 항의 계수와 지수를 저장하고, 다음 항을 가리키는 포인터를 가지고 있다. 교재에는 필요한 나머지 함수가 있기 때문에 pwrite 과 pmult 함수를 추가해서 작성해보았다.

```
////////////////////////////////////////////////////////////
void pwrite(poly_pointer p, const char *filename) {
    FILE *file = fopen(filename, "w");
    int num_terms = 0;
    poly_pointer temp = p;
    //the number of terms
    while (temp) {
        num_terms++;
        temp = temp->link;
    }
    fprintf(file, "%d\n", num_terms);
    //writing each terms to the file
    while (p) {
        fprintf(file, "%d %d\n", p->coef, p->expon);
        p = p->link;
    }

    fclose(file);
}
```

pwrite 함수는 다항식을 파일에 쓰는 기능을 나타낸다.

- 우선 다항식의 각 항의 개수를 세기 위해 다항식을 순회한다.
- 그 후, 파일에 항의 개수를 출력한다.
- 이후, 각 항의 계수와 지수를 파일에 출력한다.



```
poly_pointer pmult(poly_pointer a, poly_pointer b) {
    if (!a || !b) {
        return NULL;
    }
    poly_pointer c = NULL;
    poly_pointer temp_a, temp_b;
    poly_pointer last_c;
    //through each term of a
    for (temp_a = a; temp_a; temp_a = temp_a->link) {
        poly_pointer product = NULL, last_product;
        //through each term of b
        for (temp_b = b; temp_b; temp_b = temp_b->link) {
            poly_pointer temp_node = get_node();
            //multiply coef and add expon
            temp_node->coef = temp_a->coef * temp_b->coef;
            temp_node->expon = temp_a->expon + temp_b->expon;
            temp_node->link = NULL;
            if (!product) {
                product = temp_node;
                last_product = product;
            } else {
                last_product->link = temp_node;
                last_product = temp_node;
            }
        }
        if (!c) {
            c = product;
        } else { //add result to c
            c = padd(c, product);
            erase(&product);
        }
    }
    return c;
}
```

pmult 함수는 두 다항식을 곱하는 기능을 한다.

- 각 항의 곱을 계산하기 위해 중첩된 루프를 사용한다.
- 첫 번째 다항식의 각 항과 두 번째 다항식의 각 항을 곱한 결과를 새로운 다항식에 추가한다.
- 새로운 다항식을 만들기 위해 padd 함수를 호출해 과정을 수행했다.
- padd 함수는 다항식에 새로운 항을 추가하고, 필요한 경우 계수를 더한다.

- 시간 복잡도:

pwrite: 다항식을 한 번 순회하여 파일에 쓰므로 O(n) 시간이 걸린다. (n 은 항의 개수)

pmult: 두 다항식의 곱을 계산하기 위해 중첩된 루프를 사용하므로 O(n^2) 시간이 걸린다. (n 은 각 다항식의 항의 개수)

- 공간 복잡도:

다항식의 각 항을 저장하기 위해 메모리가 필요한다. 따라서, 공간 복잡도는 O(m+n)이 될 것이다. (m 과 n 은 각각 첫 번째 다항식과 두 번째 다항식의 항의 개수)

3. **Maze problem**

미로는 2 차원 배열로 파일 안에 있었고, 1 의 경계로 둘러싸여 있었기 때문에 1, 1 에 시작점을 넣어 도착점까지의 경로를 찾아내는 것이 목표였다.

자료구조:

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

DoublyLinkedList: 이중 링크드 리스트를 나타내는 구조체를 미로의 경로를 저장하는데 사용했다.

미로에서 경로를 찾은 데에 사용된 함수들은:

- read(const char *filename): 파일에서 미로를 읽어들이고, 각 칸의 값을 2 차원 배열인 maze 에 저장한다.
- find(int x, int y, int mark[ROWS][COLS], int path[ROWS * COLS][2], int *pathlen): 재귀적으로 DFS 를 사용하여 미로에서 경로를 찾는다. 현재 위치에서 출구까지의 경로를 path 배열에 저장하고, 경로의 길이를 pathlen 에 저장한다.
- create_list(): 빈 이중 링크드 리스트를 생성하여 반환한다.
- append(DoublyLinkedList *list, int x, int y): 리스트의 끝에 새로운 노드를 추가한다. 노드는 주어진 좌표를 가지고 있다.

- free_list(DoublyLinkedList *list): 동적으로 할당된 메모리를 해제하여 리스트를 삭제한다.
- write(const char *filename, DoublyLinkedList *list): 리스트에 있는 노드들을 파일에 쓴다.
- main(): 미로를 읽어들이고, DFS 를 사용하여 경로를 찾은 후 이를 파일에 쓴다.

<u>시간 및 공간 복잡도:</u>

- 시간 복잡도: DFS 알고리즘에 의해 결정된다. 모든 칸을 방문하며 출구를 찾는 것이 최악의 경우이므로, O(NxM)의 시간이 소요된다. 여기서 N 과 M 은 미로의 행과 열의 수이다.
- 공간 복잡도: 미로를 나타내는 2 차원 배열, 방문 여부를 나타내는 2 차원 배열, 경로를 저장하는 배열, 이중 링크드 리스트에 의해 결정된다. 여기서 N 과 M 은 미로의 행과 열의 수이다. 따라서 전체적으로 O(NxM)의 공간이 필요한다.