자료구조(Data Structure)

Programming Assignment 4

20161603
신민준

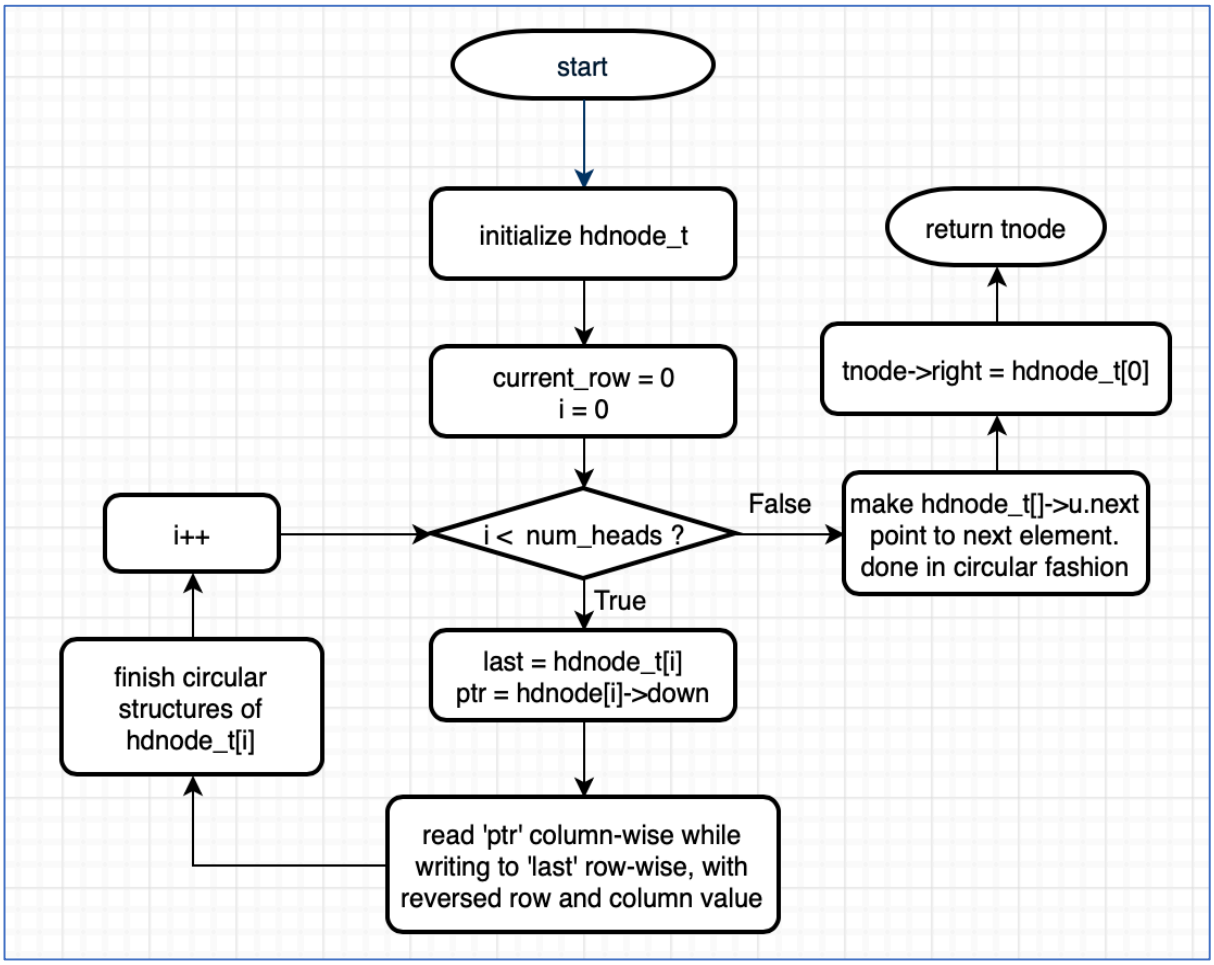
목차

1. 문제1

문제1에서는 circularly linked list를 사용한 sparse matrix 자료구조에서, input.txt 파일을 통해 입력받은 sparse matrix를 transpose 한 행렬을 output.txt 파일에 출력하도록 요구하고 있습니다. 해당 자료구조에서는 header node를 사용해 각 열과 행의 첫 시작 node를 표현하므로, row-major order로 읽은 original sparse matrix를 column-major order로 읽으면 자연스럽게 column-first-row-later 순서로 정렬된 원소를 얻을 수 있습니다. 따라서 이 방식으로 입력받은 sparse matrix를 읽어가며 행과 열의 값만 서로 바꿔 저장하는 방식으로 구현했습니다.

새로운 노드를 메모리에 할당하고 불러오는 함수로 new_node() 함수를 만들어 사용했으며, sparse matrix를 읽고 쓰는데에는 mread()와 mwrite() 함수를 사용했습니다. 이 두 함수는 교재에 있는 방식과 동일하게 작업을 수행합니다. mtranspose() 함수에서 transpose 작업을 담당합니다. 해당 함수의 flowchart는 다음과 같습니다.

1.1 프로그램 구조



Flowchart 1

1.2 세부 설명

Transposed된 sparse matrix의 row, column, number of terms를 보여주는 노드를 먼저 생성했습니다. 이 때, Transposing이 되어야 하기 때문에, 행과 열의 값은 서로 바꾸어 저장해야 합니다.

```
num_rows = header->u.entry.row;
num_cols = header->u.entry.col;
num_terms = header->u.entry.value;

tnode = new_node();
tnode->tag = entry;
tnode->u.entry.row = num_cols;
tnode->u.entry.col = num_rows;
tnode->u.entry.value = num_terms;
```

Code 1

이후, 필요한 수의 header node들을 초기화시킵니다.

```
num_heads = (num_rows > num_cols) ? num_rows : num_cols;

for(int 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;
}
```

Code 2

입력받은 sparse matrix는 hdnode[] 배열을 header node로 갖고 있기 때문에, 각 hdnode[] 원소에 대해 다음 [Code 3]와 같은 코드를 통해 입력받았던 sparse matrix를 column-wise하게 읽고, 그 내용을 hdnode_t[] 배열에 row-wise하게 복사합니다. 이 때, transposing이 일어나야 하므로 행과 열의 값은 서로 뒤바뀐 채로 복사하게 됩니다. 이후, 이 matrix의 행과 열은 circularly linked list 구조를 따라야 하므로, 마지막 행과 열의 원소가 원래의 헤더를 가르키도록 마무리합니다.

```
    for(matrix_pointer ptr = hdnode[i]->down ;
         ptr != hdnode[i] ; ptr = ptr->down) {
      row = ptr->u.entry.row;
      col = ptr->u.entry.col;
      value = ptr->u.entry.value;

      temp = new_node();
      temp->tag = entry;
      temp->u.entry.row = col;
      temp->u.entry.col = row;
      temp->u.entry.value = value;
      last->right = temp;
      last = temp;

      hdnode_t[row]->u.next->down = temp;
      hdnode_t[row]->u.next = temp;
   }
   last->right = hdnode_t[i]; // finishing row-wise circular structure
   for(int i=0 ; i<num_heads ; i++) {
      // finishing column-wise circular structure
      hdnode_t[i]->u.next->down = hdnode_t[i];
   }
```

Code 3

이 과정에서 hdnode_t[]->u.next는 입력받은 마지막 원소의 주소를 가리키는 데 사용되었으므로, 이를 원래 목적인 다음 원소로의 포인터로 사용할 수 있게 수정합니다.

```
// Set u.next to point to next header node, in circular structure
for(int i=0 ; i<num_heads-1 ; i++)
  hdnode_t[i]->u.next = hdnode_t[i+1];
hdnode_t[num_heads-1]->u.next = tnode;
tnode->right = hdnode_t[0];

return tnode;
```

Code 4

이 프로시져의 시간복잡도는 다음과 같습니다.

$r = rows\ in\ original\ sparse\ matrix, c = columns\ in\ original\ sparse\ matrix$일 때,

$$O(c \times \max\{r, c\})$$

1.3 참고 사항

- Header node 의 u.next 원소가 입력받을 때엔 같은 열의 다음 원소를 가르키는데 사용되고, 입력받은 후에는 다음 header node 를 가르키는 새로운 용도로 쓰이는 것이 인상깊었습니다. 따로 배열을 선언해 메모리를 낭비하는 대신, 이러한 방법으로 최적화를 이루는 것이 이상적이었고, 배울 부분이 많은 코드였습니다.

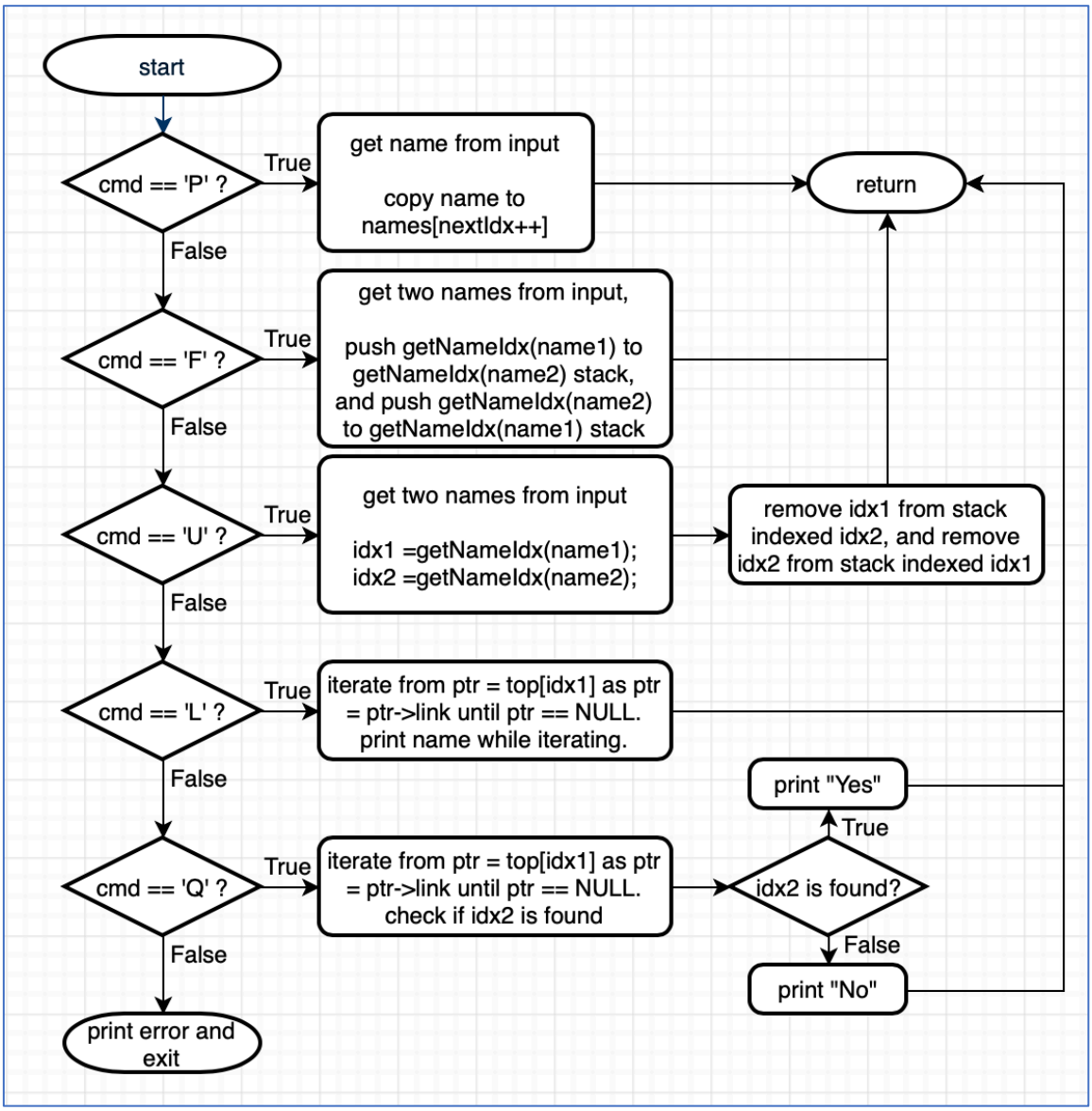
2. 문제2

문제2에서 요구하는 equivalance class는 교재의 방식 그대로, 스택을 사용해 구현할 수 있습니다. 이 때, 종료 명령인 X를 제외한 모든 명령어를 처리하는 commandHandler 함수를 사용해 명령어를 처리했습니다.

각 명령어에 대한 처리는 입력받은 명령어에 따라 알맞은 수의 추가 입력을 받고 이를 곧바로 출력 파일에 쓰는 방식으로 입출력이 동시에 이루어지도록 했습니다.

commandHandler 함수의 flowchart는 다음과 같습니다.

2.1 프로그램 구조



Flowchart 2

2.2 세부 설명

이 함수는 받은 command에 따라 입력받을 이름의 갯수가 달라집니다. 따라서, 입력과 출력을 동시에 하면서 프로세스를 진행하는 것이 간단하고 효율적일거라 생각해 이와 같이 구현

했습니다.

문자열을 처리할 때, 문자열 그대로 프로그램 내에서 사용하는 것은 공간적인 측면에서나 유지보수적인 측면에서 불리한 점이 많을 것이라 판단하여, names 라는 새로운 전역변수를 선언하고, 저장된 이름의 갯수를 nextIdx에 저장해 문자열 대신 names 전역변수의 인덱스 값으로 이름을 처리하도록 구현했습니다.

```
char names[MAX_PEOPLE_NUM][MAX_NAME_LENGTH];
int nextIdx = 0;
/**
 * Gets index number of given name
 * @param  name Character string
 * @return      index of name string, or -1 if such index is not found
 */
int getNameIdx(char* name) {
  int idx;

  for(idx=0 ; idx<nextIdx ; idx++) {
    if(!strcmp(name, names[idx]))
      break;
  }
  if(idx == nextIdx) idx = -1;
  return idx;
}
```

Code 5

[Code 5]에서 구현한 내용을 바탕으로, 새로운 사람을 추가하는 'P' 명령과 친구 관계를 만드는 'F' 명령은 다음과 같이 구현할 수 있었습니다. 이 때 교재에 나온 equivalance class 구현 방법과 마찬가지로 각 이름의 index에 해당하는 스택에 서로의 index를 push 하는 방법을 사용했습니다.

```
    if(cmd == 'P') { // Add new person
      fscanf(in, " %s", name1);
      strcpy(names[nextIdx++], name1);
    } else if(cmd == 'F') { // Befriend
      fscanf(in, " %s", name1);
      fscanf(in, " %s", name2);
      idx1 = getNameIdx(name1);
      idx2 = getNameIdx(name2);

      push(idx1, idx2);
      push(idx2, idx1);
    }
```

Code 6

친구 관계를 끊는 작업은 각자의 이름 index의 스택에서 서로의 이름 index에 해당하는 원소를 삭제하는 방법으로 구현했습니다. 해당 코드는 [Code 7]에 나타나있습니다.

```
    else if(cmd == 'U') { // Unfriend
      fscanf(in, " %s", name1);
      fscanf(in, " %s", name2);
      idx1 = getNameIdx(name1);
      idx2 = getNameIdx(name2);

      // remove idx2 from idx1 stack
      prev = NULL;
```

```
        for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
          if(prev == NULL) {
            if(ptr->index == idx2) {
              top[idx1] = ptr->link;
              free(ptr);
              break;
            }
          } else if(ptr->index == idx2) {
            prev->link = ptr->link;
            free(ptr);
            break;
          }
          prev = ptr;
        }
        // remove idx1 from idx2 stack
        prev = NULL;
        for(node* ptr = top[idx2] ; ptr != NULL ; ptr = ptr->link) {
          if(prev == NULL) {
            if(ptr->index == idx1) {
              top[idx2] = ptr->link;
              free(ptr);
              break;
            }
          } else if(ptr->index == idx1) {
            prev->link = ptr->link;
            free(ptr);
            break;
          }
          prev = ptr;
        }
      }
```

Code 7

모든 친구들의 리스트를 출력하는 명령어 'L'은 간단하게 해당 스택의 모든 원소를 출력해주는 방식으로 구현했습니다. 또한, [Code 5]에서 보인 getNameIdx() 함수의 구현방법에 의해, 친구 목록을 보여주길 원하는 사람의 이름이 등록되어있지 않은 상태라면 -1을 리턴하므로, 이를 이용해 예외처리를 했습니다.

```
      else if(cmd == 'L') { // List all friends
        fscanf(in, " %s", name1);
        idx1 = getNameIdx(name1);

        if(idx1 == -1) {
          fprintf(out, "None\n");
          return;
        }

        for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
          fprintf(out, "%s", names[ptr->index]);
          if(ptr->link != NULL) fprintf(out, " ");
        }
        fprintf(out, "\n");
      }
```

Code 8

두 사람이 서로 친구 관계에 있는지 확인하기 위해서는 각 사람의 stack에 서로의 이름 인덱스가 존재하는지 확인하면 됩니다. 다만, 이 프로그램의 구현 방법 상 한 사람의 스텍에 다른

사람의 인덱스가 존재한다면 그 반대도 참일 수 밖에 없기 때문에, [Code 9]에서처럼 시간을 아끼기 위해 한쪽 방향으로만 확인을 진행했습니다.

```
else if(cmd == 'Q') { // Check if friends
   fscanf(in, " %s", name1);
   fscanf(in, " %s", name2);
   idx1 = getNameIdx(name1);
   idx2 = getNameIdx(name2);

   isFriend = 0;
   for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
      if(ptr->index == idx2) {
         isFriend = 1;
         break;
      }
   }
   fprintf(out, "%s\n", isFriend ? "Yes" : "No");
}
```

Code 9

commandHandler 함수의 시간복잡도는 n = max stack length in top[] array 이라고 할 때, $O(n)$입니다.

2.3 참고 사항

- 이 프로그램에서는 스택의 최대 크기를 처음부터 정적 배열로 선언했습니다. 만약 동적 할당을 통해 스택의 크기를 pop, push에 따라 조정했다면, 메모리를 현저히 적게 사용할 수 있는 방향으로 최적화가 가능했을 것입니다. OOP 언어를 사용한다면 이 과정을 구현하는 것 또한 간단하게 이룰 수 있습니다.
- 이름의 인덱스를 저장하기 위한 names 전역변수도 그 길이를 정적배열로 선언했는데, 이름을 추가할 때 마다 배열의 크기를 증가시키는 방향으로 구현한다면 메모리 낭비를 크게 줄일 수 있을 것입니다.

3. 문제 코드

3.1 문제1 코드

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

#define MAX_SIZE 50
#define INPUT_FILENAME "input.txt"
#define OUTPUT_FILENAME "output.txt"

typedef enum {head, entry} tagfield;
typedef struct matrix_node* matrix_pointer;
struct entry_node {
   int row;
   int col;
   int value;
};
struct matrix_node {
   matrix_pointer down;
   matrix_pointer right;
   tagfield tag;
   union {
      matrix_pointer next;
      struct entry_node entry;
   } u;
};
matrix_pointer hdnode[MAX_SIZE];
matrix_pointer hdnode_t[MAX_SIZE];

/**
 * returns one newly allocated node
 * @return node created
 */
matrix_pointer new_node() {
   matrix_pointer new = malloc(sizeof(struct matrix_node));
   return new;
}
/**
 * Transposes given sparse matrix
 * @param  header pointer to header node for the sparse matrix to transpose
 * @return       pointer to header node of the transposed sparse matrix
 */
matrix_pointer mtranspose(matrix_pointer header) {
   matrix_pointer tnode, last, temp;
   int num_rows, num_cols, num_terms, num_heads;
   int row, col, value;

   num_rows = header->u.entry.row;
   num_cols = header->u.entry.col;
   num_terms = header->u.entry.value;

   tnode = new_node();
   tnode->tag = entry;
   tnode->u.entry.row = num_cols;
   tnode->u.entry.col = num_rows;
   tnode->u.entry.value = num_terms;

   num_heads = (num_rows > num_cols) ? num_rows : num_cols;

   for(int 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;
   }
   for(int i=0 ; i<num_heads ; i++) {
      last = hdnode_t[i];
      // read hdnode[i] column-wise, while writing to hdnode_t[i]
      // row-wise, with the row and column value exchanged.
      for(matrix_pointer ptr = hdnode[i]->down ;
           ptr != hdnode[i] ; ptr = ptr->down) {
         row = ptr->u.entry.row;
         col = ptr->u.entry.col;
         value = ptr->u.entry.value;

         temp = new_node();
         temp->tag = entry;
         temp->u.entry.row = col;
         temp->u.entry.col = row;
         temp->u.entry.value = value;
         last->right = temp;
         last = temp;

         hdnode_t[row]->u.next->down = temp;
         hdnode_t[row]->u.next = temp;
      }
      last->right = hdnode_t[i]; // finishing row-wise circular structure
      for(int i=0 ; i<num_heads ; i++) {
         // finishing column-wise circular structure
         hdnode_t[i]->u.next->down = hdnode_t[i];
      }
   }

   // Set u.next to point to next header node, in circular structure
   for(int i=0 ; i<num_heads-1 ; i++)
      hdnode_t[i]->u.next = hdnode_t[i+1];
   hdnode_t[num_heads-1]->u.next = tnode;
   tnode->right = hdnode_t[0];

   return tnode;
}
/**
 * read sparse matrix from INPUT_FILENAME
 * @return pointer to the head node read
 */
matrix_pointer mread() {
   int num_rows, num_cols, num_terms, num_heads, i;
   int row, col, value, current_row;
   matrix_pointer temp, last, node;
   FILE* fp;
   if(!(fp = fopen(INPUT_FILENAME, "r"))) {
      fprintf(stderr, "Input file is not found. Terminating.\n");
      exit(1);
   }

   fscanf(fp, "%d %d %d", &num_rows, &num_cols, &num_terms);
   num_heads = (num_cols > num_rows) ? num_cols : num_rows;

   node = new_node();
   node->tag = entry;
   node->u.entry.row = num_rows;
   node->u.entry.col = num_cols;
   node->u.entry.value = num_terms;

   if(!num_heads)
```

```
      node->right = node;
   else {
      for(int i=0 ; i<num_heads ; i++) {
         temp = new_node();
         hdnode[i] = temp;
         hdnode[i]->tag = head;
         hdnode[i]->right = temp;
         hdnode[i]->u.next = temp;
      }
      current_row = 0;
      last = hdnode[0];
      for(int i=0 ; i<num_terms ; i++) {
         fscanf(fp, "%d %d %d", &row, &col, &value);
         if(row > current_row) {
            last->right = hdnode[current_row];
            current_row = row;
            last = hdnode[row];
         }
         temp = new_node();
         temp->tag = entry;
         temp->u.entry.value = value;
         temp->u.entry.row = row;
         temp->u.entry.col = col;
         last->right = temp;
         last = temp;

         hdnode[col]->u.next->down = temp;
         hdnode[col]->u.next = temp;
      }
      last->right = hdnode[current_row];

      for(i=0 ; i<num_cols ; i++) {
         hdnode[i]->u.next->down = hdnode[i];
      }

      for(i=0 ; i<num_heads-1 ; i++)
         hdnode[i]->u.next = hdnode[i+1];
      hdnode[num_heads-1]->u.next = node;
      node->right = hdnode[0];
   }
   fclose(fp);
   return node;
}
/**
 * Writes the content of sparse matrix to OUTPUT_FILENAME
 * @param node Pointer to head node of sparse matrix
 */
void mwrite(matrix_pointer node) {
   int i;
   matrix_pointer temp, head = node->right;
   FILE* fp;

   if(!(fp = fopen(OUTPUT_FILENAME, "w"))) {
      fprintf(stderr, "Error while writing output to file.\n");
      exit(1);
   }

   fprintf(fp, "%d %d %d\n",
      node->u.entry.row, node->u.entry.col, node->u.entry.value);

   for(i=0 ; i < node->u.entry.row ; i++) {
      for(temp=head->right ; temp!=head ; temp=temp->right)
         fprintf(fp, "%d %d %d\n",
```

```
            temp->u.entry.row, temp->u.entry.col, temp->u.entry.value);
      head = head->u.next;
   }
   fclose(fp);
}
int main(int argc, const char* argv[]) {
   matrix_pointer node, tnode, temp;
   int num_heads;

   node = mread();
   tnode = mtranspose(node);
   mwrite(tnode);

   // freeing nodes
   num_heads = (node->u.entry.row > node->u.entry.col) ?
      node->u.entry.row : node->u.entry.col;
   for(int i=0 ; i<num_heads ; i++) {
      // freeing initial sparse matrix
      for(matrix_pointer ptr = hdnode[i]->right ; ptr != hdnode[i] ; ){
         temp = ptr;
         ptr = ptr->right;
         free(temp);
      }
      // freeing transposed sparse matrix
      for(matrix_pointer ptr = hdnode_t[i]->right ; ptr != hdnode_t[i] ; ){
         temp = ptr;
         ptr = ptr->right;
         free(temp);
      }
   }
   return 0;
}
```

3.2 문제2 코드

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

#define INPUT_FILENAME "input.txt"
#define OUTPUT_FILENAME "output.txt"
#define MAX_PEOPLE_NUM 100
#define MAX_NAME_LENGTH 30

typedef struct _node{
   int index;
   struct _node* link;
} node;

node* top[MAX_PEOPLE_NUM] = {0};
char names[MAX_PEOPLE_NUM][MAX_NAME_LENGTH];
int nextIdx = 0;

/**
 * Gets index number of given name
 * @param  name Character string
 * @return      index of name string, or -1 if such index is not found
 */
int getNameIdx(char* name) {
   int idx;

   for(idx=0 ; idx<nextIdx ; idx++) {
```

```
      if(!strcmp(name, names[idx]))
        break;
    }
    if(idx == nextIdx) idx = -1;
    return idx;
}
/**
 * Push name index into stack
 * @param stack_idx Index of the stack to push to
 * @param names     Name index to push
 */
void push(int stack_idx, int names) {
    node* new = malloc(sizeof(node));

    new->index = names;
    new->link = top[stack_idx];
    top[stack_idx] = new;
}
/**
 * Pops name index from stack
 * @param  stack_idx Index of the stack to pop
 * @return           Popped name index
 */
int pop(int stack_idx) {
    node* popped;
    int value;

    if(!top[stack_idx]) {
        fprintf(stderr, "Trying to pop an empty stack. Terminating.\n");
        exit(1);
    }

    popped = top[stack_idx];
    value = popped->index;

    top[stack_idx] = top[stack_idx]->link;

    // Avoiding dangling pointers
    free(popped);
    popped = NULL;

    return value;
}
/**
 * Executes command
 * @param in  Input file pointer
 * @param out Output file pointer
 * @param cmd Command in character to execute
 */
void commandHandler(FILE* in, FILE* out, char cmd) {
    char name1[MAX_NAME_LENGTH], name2[MAX_NAME_LENGTH];
    int idx1, idx2, isFriend;
    node *prev;

      if(cmd == 'P') { // Add new person
        fscanf(in, " %s", name1);
        strcpy(names[nextIdx++], name1);
      } else if(cmd == 'F') { // Befriend
        fscanf(in, " %s", name1);
        fscanf(in, " %s", name2);
        idx1 = getNameIdx(name1);
        idx2 = getNameIdx(name2);
```

```
        push(idx1, idx2);
        push(idx2, idx1);
    } else if(cmd == 'U') { // Unfriend
        fscanf(in, " %s", name1);
        fscanf(in, " %s", name2);
        idx1 = getNameIdx(name1);
        idx2 = getNameIdx(name2);

        // remove idx2 from idx1 stack
        prev = NULL;
        for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
            if(prev == NULL) {
                if(ptr->index == idx2) {
                    top[idx1] = ptr->link;
                    free(ptr);
                    break;
                }
            } else if(ptr->index == idx2) {
                prev->link = ptr->link;
                free(ptr);
                break;
            }
            prev = ptr;
        }
        // remove idx1 from idx2 stack
        prev = NULL;
        for(node* ptr = top[idx2] ; ptr != NULL ; ptr = ptr->link) {
            if(prev == NULL) {
                if(ptr->index == idx1) {
                    top[idx2] = ptr->link;
                    free(ptr);
                    break;
                }
            } else if(ptr->index == idx1) {
                prev->link = ptr->link;
                free(ptr);
                break;
            }
            prev = ptr;
        }
    } else if(cmd == 'L') { // List all friends
        fscanf(in, " %s", name1);
        idx1 = getNameIdx(name1);

        if(idx1 == -1) {
            fprintf(out, "None\n");
            return;
        }

        for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
            fprintf(out, "%s", names[ptr->index]);
            if(ptr->link != NULL) fprintf(out, " ");
        }
        fprintf(out, "\n");
    } else if(cmd == 'Q') { // Check if friends
        fscanf(in, " %s", name1);
        fscanf(in, " %s", name2);
        idx1 = getNameIdx(name1);
        idx2 = getNameIdx(name2);

        isFriend = 0;
        for(node* ptr = top[idx1] ; ptr != NULL ; ptr = ptr->link) {
            if(ptr->index == idx2) {
```

```
            isFriend = 1;
            break;
         }
      }
      fprintf(out, "%s\n", isFriend ? "Yes" : "No");
   } else {
      fprintf(stderr, "Incorrect input file format.\n");
      exit(1);
   }
}
int main(int argc, const char* argv[]) {
   FILE* in_fp;
   FILE* out_fp;
   char command;

   if(!(in_fp = fopen(INPUT_FILENAME, "r"))) {
      fprintf(stderr, "Input file not found.\n");
      exit(1);
   }
   if(!(out_fp = fopen(OUTPUT_FILENAME, "w"))) {
      fprintf(stderr, "Cannot open output file.\n");
      exit(1);
   }

   while(fscanf(in_fp, " %c", &command) != EOF) {
      if(command == 'X') break;
      commandHandler(in_fp, out_fp, command);
   }

   fclose(in_fp);
   fclose(out_fp);

   // freeing all stack nodes
   for(int i=0 ; i<nextIdx ; i++) {
      while(top[i] != NULL)
         pop(i);
   }
   return 0;
}
```