컴퓨터학부 20211527 박창현

honey_bee

1. **과제개요**

이 프로그램은 디렉토리 구조를 트리 형태로 출력하고, 디렉토리 내의 파일들을 확장자 기준으로 정리하는 기능을 제공합니다. 프로그램은 tree, arrange, help, exit 명령어를 지원하며, 각 명령어는 특정 옵션을 통해 추가 기능을 제공합니다. 프로그램은 사용자의 홈 디렉토리 내에서만 동작하도록 제한되어 있으며, 잘못된 경로나 옵션 입력 시 적절한 에러 메시지를 출력합니다.

2. **기능(**구현한 기능 요약)

ssu_cleanup

2-1 tree 내장명령어

디렉토리의 구조를 트리 형태로 출력합니다.

Usage : tree <DIR_PATH> [OPTION] ...

- tree <DIR_PATH>: 지정된 디렉토리와 파일을 트리 구조로 출력합니다.
- tree <DIR_PATH> -s: 파일 크기를 함께 출력합니다.
- tree <DIR_PATH> -p: 파일의 권한을 함께 출력합니다.
- tree <DIR_PATH> -sp: 파일 크기 및 권한을 함께 출력합니다.

2-2 arrange 내장명령어

디렉토리 내 파일을 정리합니다.

Usage : arrange <DIR_PATH> [OPTION] ...

- arrange <DIR_PATH>: 기본적으로 파일을 정리합니다.
- arrange <DIR_PATH> -d <OUTPUT_PATH>: 정리된 파일을 특정 디렉토리에 저장합니다.
- arrange <DIR_PATH> -t <SECONDS>: 주어진 초(seconds) 이내 수정된 파일만 정리합니다.
- arrange <DIR_PATH> -x <EXCLUDE_PATHS>: 특정 파일이나 디렉토리를 제외하고 정리합니다.
- arrange <DIR_PATH> -e <EXTENSIONS>: 특정 확장자를 가진 파일만 정리합니다.

옵션 –d, -t, -x, -e를 혼용하여 사용할 수 있습니다.

2-2-1 파일명 중복 처리

중복된 파일을 탐색하여 사용자에게 선택지를 제공합니다.

사용자 선택에 따라 선택한 파일을 유지(select), 비교(diff), 편집(vi), 또는 제외(do not select)할 수 있습니다.

2-3 help 내장명령어

Usage : help

프로그램의 사용법을 출력합니다.

2.4 exit 내장명령어

Usage : exit

프로그램을 종료합니다.

3. **상세설계**(함수 및 모듈 구성, 순서도, 구현한 함수 프로토타입 등

3-1 함수 구성

<ssu_cleanup.c>

3-1-1 main():

프로그램의 메인 루프를 실행합니다.

3-1-2 print_help():

프로그램의 사용법을 출력합니다.

3-1-3 execute_tree(char *command):

tree 명령어를 실행합니다.

3-1-4 print_tree(const char *dir_path, int depth, int show_size, int show_perm, int is_last[]):

디렉토리 구조를 트리 형태로 출력합니다.

3-1-5 execute_arrange(const char *command):

arrange 명령어를 실행합니다.

3-1-6 scan_directory(const char *dir_path, int t_option, time_t time_limit, char exclude_paths[][MAX_NAME], int excl_count, char extensions[][32], int ext_count):

디렉토리를 스캔하여 파일 노드를 생성합니다.

3-1-7 process_duplicates():

중복 파일을 검사하고 처리합니다.

3-1-8 handle_duplicate(FileNode *duplicates[], int count):

중복 파일을 처리합니다.

3-1-9 organize_files(const char *output_path):

파일을 확장자별로 정리합니다.

3-1-10 copy_file(const char *src, const char *dest):

파일을 복사합니다.

3-1-11 is_extension_allowed(const char *ext, char extensions[][32], int ext_count):

주어진 확장자가 허용된 확장자 목록에 있는지 확인합니다.

3-1-12 free_list():

연결 리스트의 메모리를 해제합니다.

3-1-13 get_extension(const char *filename):

파일의 확장자를 추출합니다.

3-1-14 add_file_node(const char *path, const char *extension, time_t mod_time):

파일 노드를 연결 리스트에 추가합니다.

3-1-15 get_permission_string(mode_t mode, char *perm_str):

파일 또는 디렉토리의 권한을 문자열로 변환합니다.

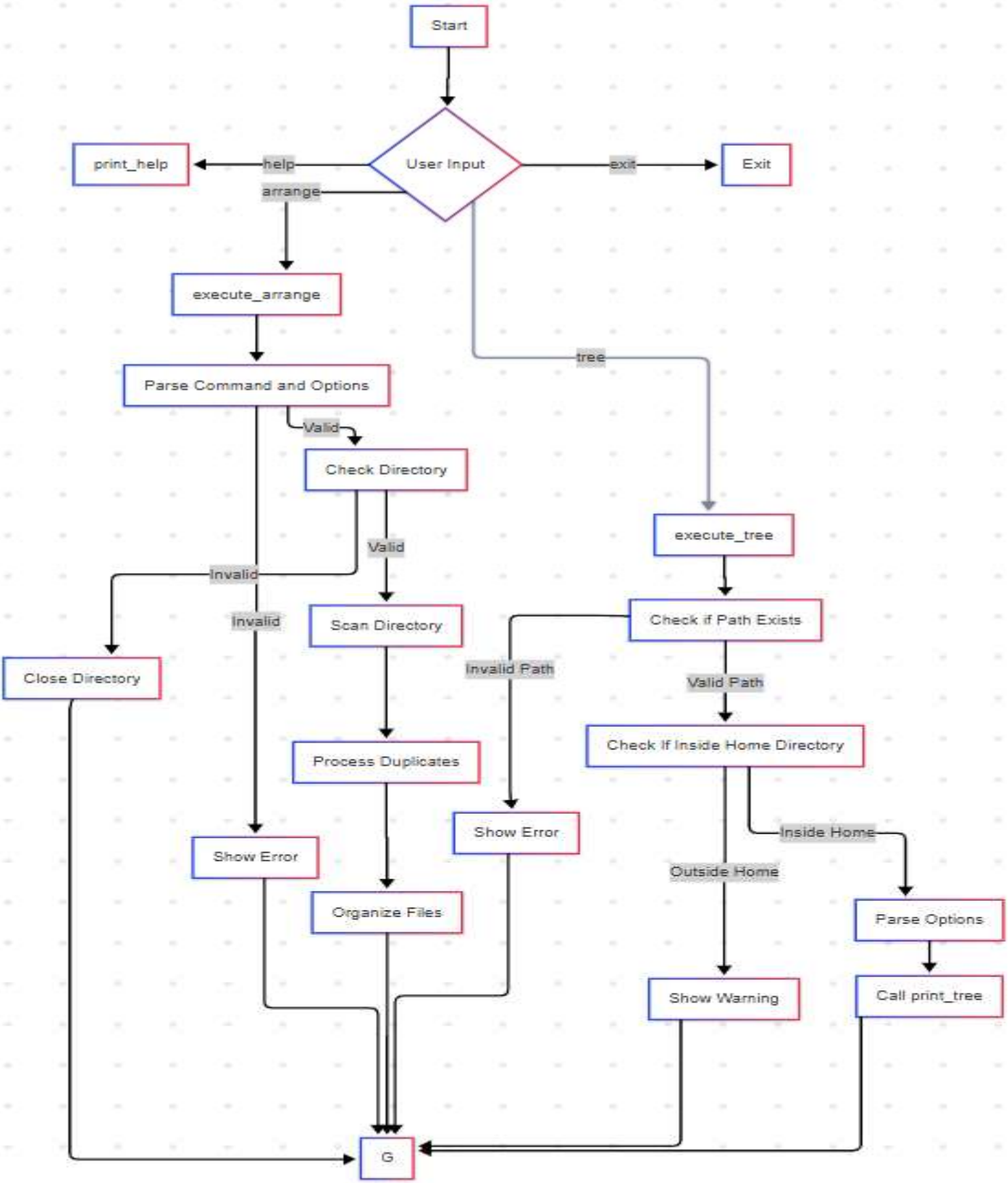
3-1-16 is_inside_home_directory(const char *path):

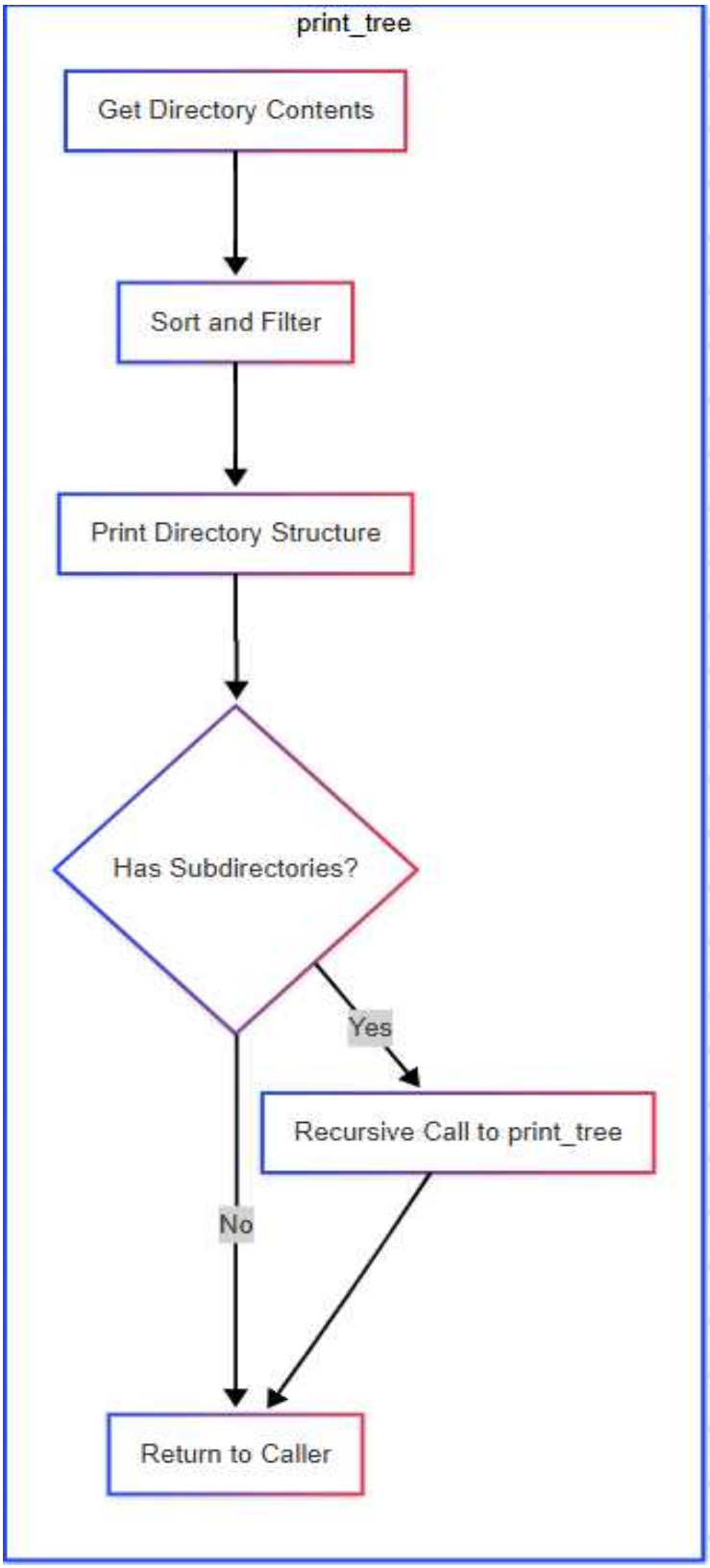
주어진 경로가 사용자의 홈 디렉토리 내에 있는지 확인합니다.

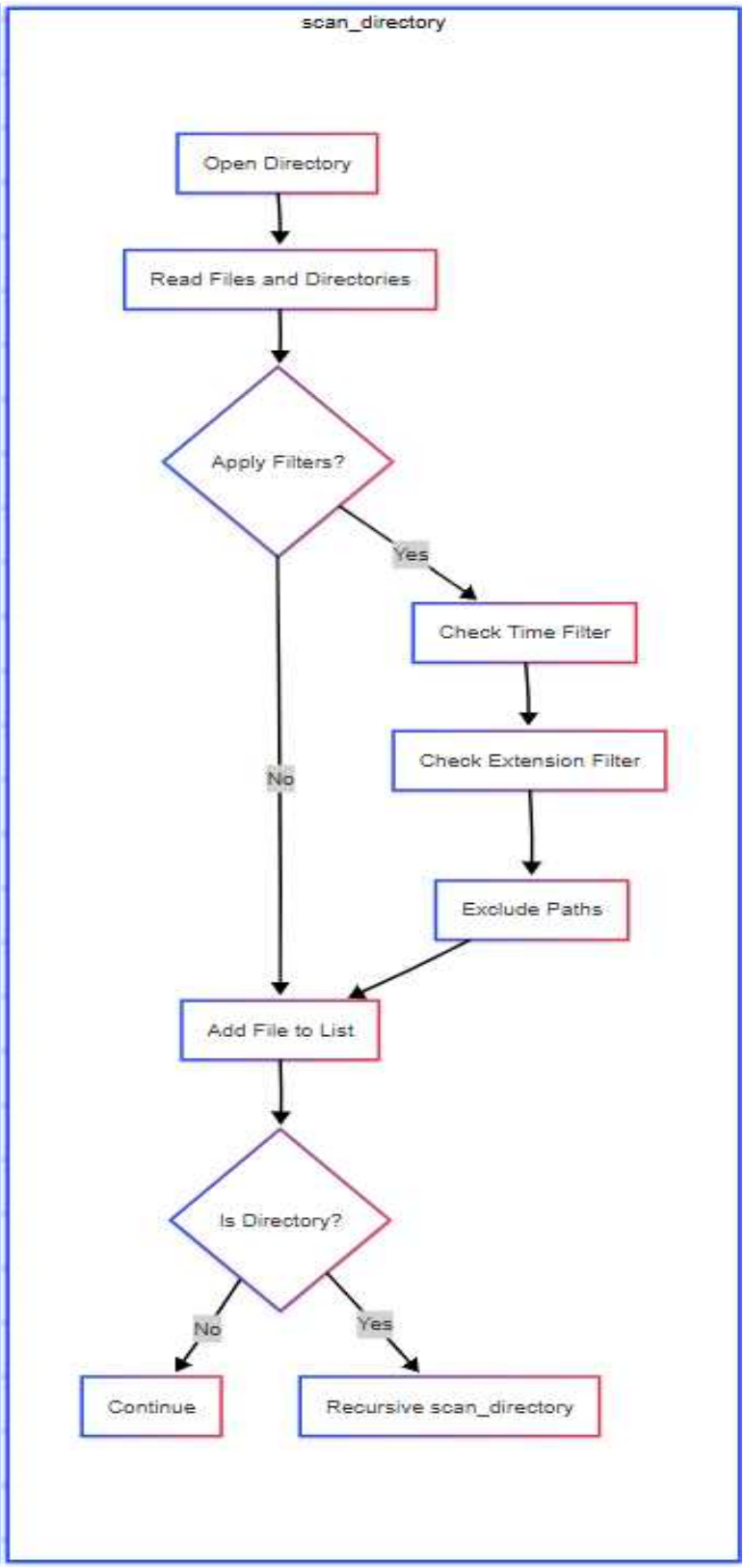
3-2 함수 프로토타입

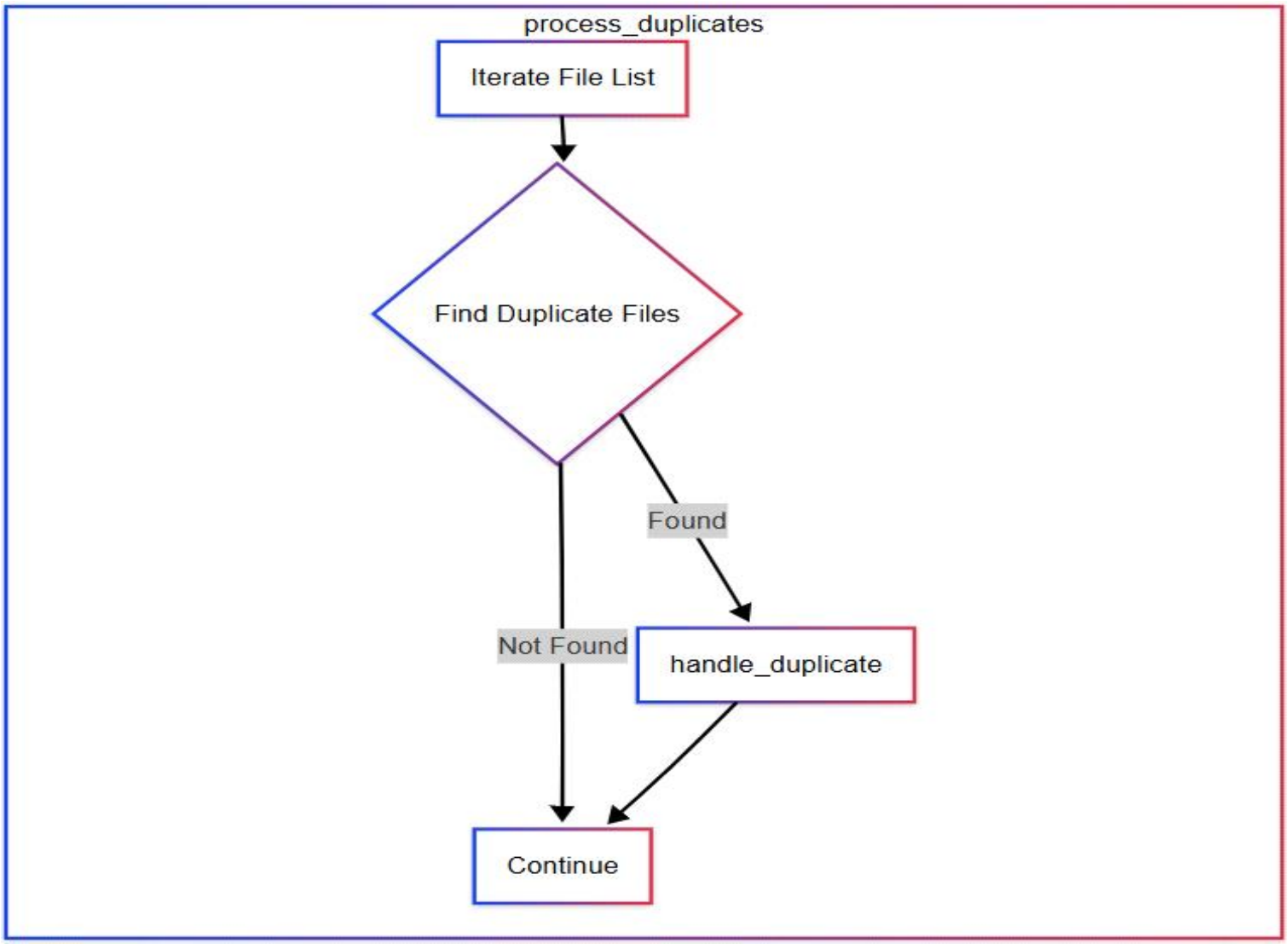
```
void print_help();

void get_permission_string(mode_t mode, char *perm_str);

int is_inside_home_directory(const char *path);

void print_tree(const char *dir_path, int depth, int show_size, int show_perm, int is_last[]);

void execute_tree(char *command);

void add_file_node(const char *path, const char *extension, time_t mod_time);

const char *get_extension(const char *filename);

void free_list();

int is_extension_allowed(const char *ext, char extensions[][32], int ext_count);

void copy_file(const char *src, const char *dest);

void organize_files(const char *output_path);

void handle_duplicate(FileNode *duplicates[], int count);

void process_duplicates();

void scan_directory(const char *dir_path, int t_option, time_t time_limit, char exclude_paths[][MAX_NAME], int excl_count, char extensions[][32], int ext_count);

void execute_arrange(const char *command);

int main();
```

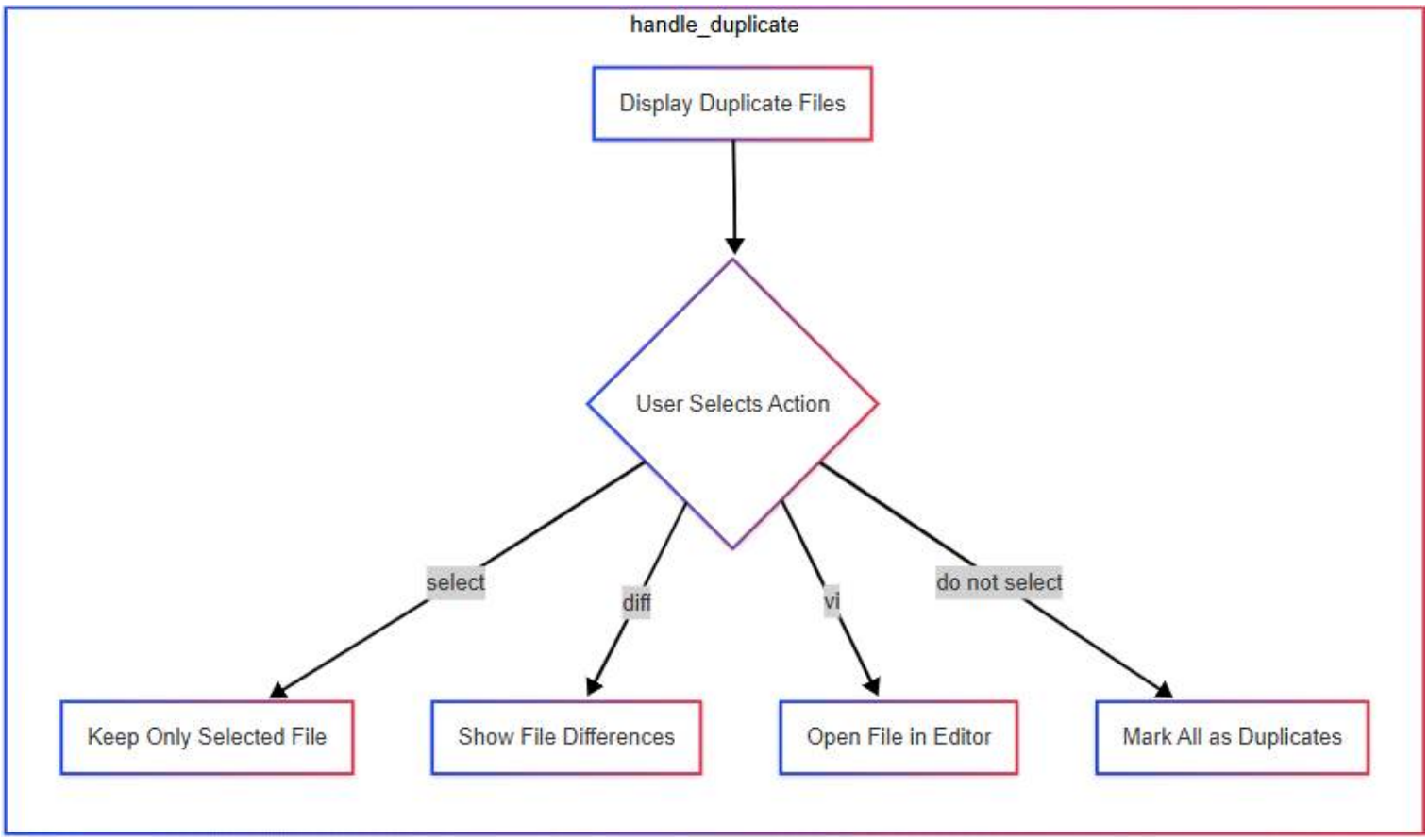
3-3 순서도

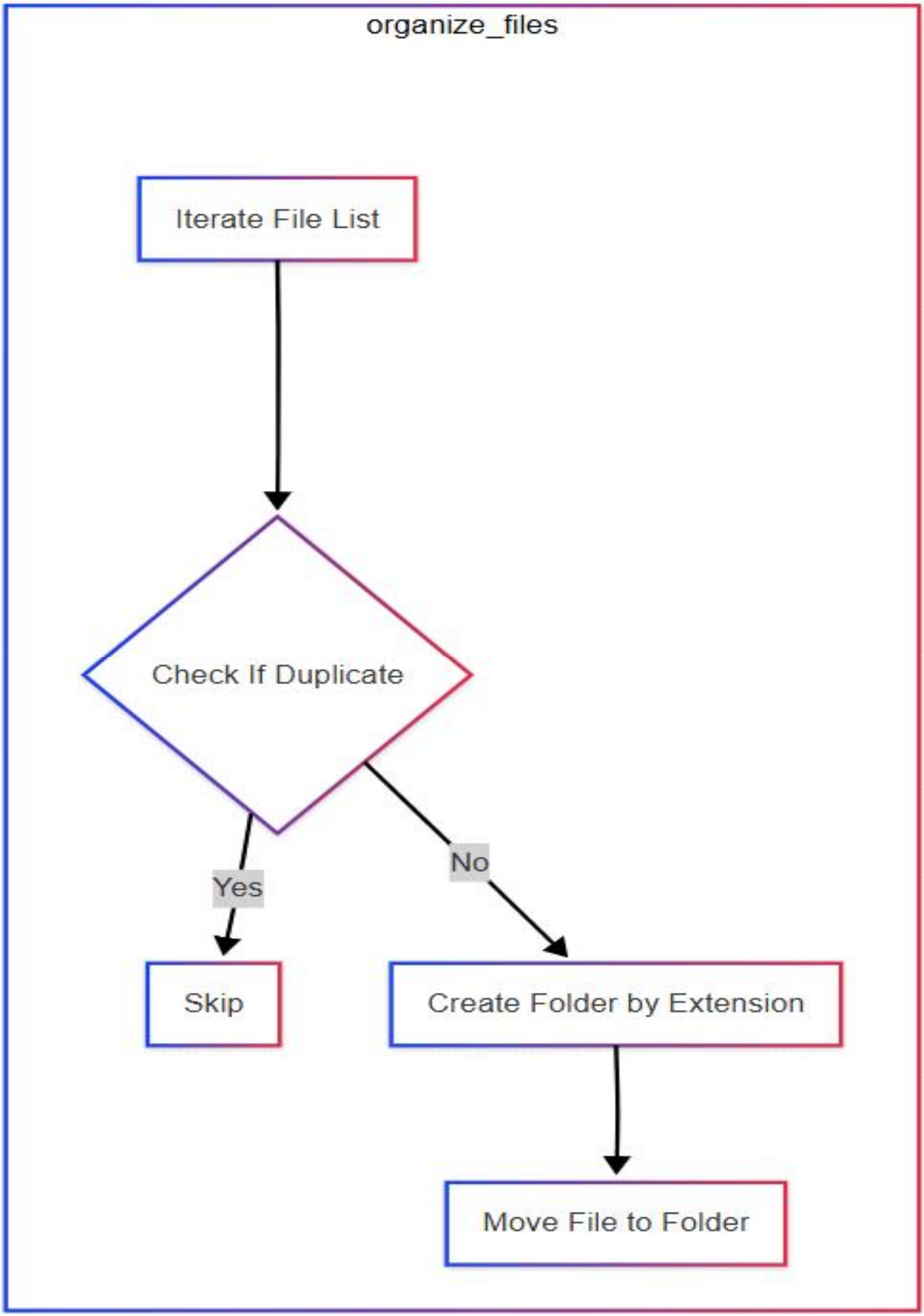

Start
User Input
print_help
help
arrange
exit
Exit
execute_arrange
tree
Parse Command and Options
Valid
Check Directory
execute_tree
Valid
Invalid
Invalid
Scan Directory
Check if Path Exists
Close Directory
Invalid Path
Valid Path
Check If Inside Home Directory
Process Duplicates
Show Error
Inside Home
Show Error
Outside Home
Parse Options
Organize Files
Show Warning
Call print_tree
G


print_tree
Get Directory Contents
Sort and Filter
Print Directory Structure
Has Subdirectories?
Yes
Recursive Call to print_tree
No
Return to Caller


scan_directory
Open Directory
Read Files and Directories
Apply Filters?
Yes
Check Time Filter
Check Extension Filter
Exclude Paths
No
Add File to List
Is Directory?
No
Yes
Continue
Recursive scan_directory


process_duplicates
Iterate File List
Find Duplicate Files
Found
Not Found
handle_duplicate
Continue


handle_duplicate
Display Duplicate Files
User Selects Action
select
diff
vi
do not select
Keep Only Selected File
Show File Differences
Open File in Editor
Mark All as Duplicates


organize_files
Iterate File List
Check If Duplicate
Yes
No
Skip
Create Folder by Extension
Move File to Folder

1. 프로그램 초기 실행 (main())

main() 함수는 사용자의 입력을 받으며, 적절한 명령어 (tree, arrange, help, exit)에 따라 해당 기능을 실행.

2. tree 명령어 흐름

execute_tree():

명령어를 분석하고 경로를 확인.

홈 디렉토리 내인지 확인 후 옵션을 분석.

print_tree()를 호출하여 디렉토리 구조를 출력.

print_tree() (재귀)

디렉토리 내 파일/폴더를 스캔하고 정렬.

트리 구조를 출력한 후, 서브디렉토리가 있다면 재귀적으로 print_tree()를 호출.

3. arrange 명령어 흐름

execute_arrange():

명령어 인자를 분석하고 유효성 검사.

scan_directory()를 호출하여 디렉토리 내 파일을 분석.

process_duplicates()를 호출하여 중복 파일을 처리.

organize_files()를 호출하여 파일을 정리.

scan_directory()

디렉토리를 열고 파일/디렉토리를 읽음.

파일을 옵션(-t, -e, -x)에 따라 필터링.

디렉토리라면 재귀적으로 scan_directory() 호출.

process_duplicates()

파일 목록을 순회하면서 중복 파일을 찾음.

handle_duplicate()를 호출하여 사용자 선택에 따라 파일을 유지/비교/편집/삭제.

organize_files()

확장자별 폴더를 생성하고 파일을 이동.

4. **주석달린 소스코드**(makefile, *.c, *.h 등)

```
#define _GNU_SOURCE     // GNU 확장 기능 사용을 위한 매크로 정의
#include <stdio.h>      // 표준 입출력 함수를 사용하기 위한 헤더 파일
#include <stdlib.h>     // 메모리 할당 및 프로그램 종료 함수를 사용하기 위한 헤더 파일
#include <string.h>     // 문자열 처리 함수를 사용하기 위한 헤더 파일
#include <dirent.h>     // 디렉토리 관련 함수를 사용하기 위한 헤더 파일
#include <sys/stat.h>   // 파일 상태 정보를 다루기 위한 헤더 파일
#include <sys/types.h> // 시스템 데이터 타입 정의를 위한 헤더 파일
#include <sys/wait.h>   // 프로세스 관련 함수를 사용하기 위한 헤더 파일
#include <unistd.h>     // POSIX 운영체제 API를 사용하기 위한 헤더 파일
#include <limits.h>     // 시스템 제한값을 정의하는 상수를 사용하기 위한 헤더 파일
#include <time.h>       // 시간 관련 함수를 사용하기 위한 헤더 파일
#include <errno.h>      // 오류 번호를 다루기 위한 헤더 파일
#define MAX_PATH 4096       // 경로의 최대 길이를 정의
#define MAX_NAME 255        // 파일 또는 디렉토리 이름의 최대 길이를 정의
#define MAX_DUPLICATES 100 // 중복 파일 처리 시 최대 허용 개수
int dir_count = 1, file_count = 0; // 디렉토리와 파일 개수를 저장하는 전역 변수
```

4-1 main

```
int main()
{
    char command[7000];          // 사용자 명령어를 저장할 배열
    int student_id = 20211527; // 학번
    while (1)
    {
        printf("%d> ", student_id);                 // 프롬프트 출력
        fgets(command, sizeof(command), stdin); // 사용자 입력 받기
        command[strcspn(command, "\n")] = 0;     // 개행 문자 제거
        if (strcmp(command, "exit") == 0) // exit 명령어 입력 시 프로그램 종료
            break;
        else if (strcmp(command, "help") == 0) // help 명령어 입력 시 사용법 출력
            print_help();
        else if (strncmp(command, "tree", 4) == 0) // tree 명령어 입력 시 실행
        {
            execute_tree(command);
        }
        else if (strncmp(command, "arrange", 7) == 0) // arrange 명령어 입력 시 실행
        {
            execute_arrange(command);
        }
        else if (strcmp(command, "") == 0) // 엔터만 입력 시 프롬프트 재출력
            continue;
        else // 잘못된 명령어 입력 시 사용법 출력
        {
            print_help();
        }
    }
    return 0;
}
```

4-2 print_help

```
// 프로그램 사용법을 출력하는 함수
void print_help()
{
    printf("Usage:\n");
    printf("> tree <DIR_PATH> [OPTION]...\n");
    printf("  <none> : Display the directory structure recursively\n");
    printf("  -s : Include the size of each file\n");
    printf("  -p : Include the permissions of each directory and file\n");
    printf("> arrange <DIR_PATH> [OPTION]...\n");
    printf("  <none> : Arrange the directory\n");
    printf("  -d <output_path> : Specify the output directory\n");
    printf("  -t <seconds> : Only arrange files modified within given seconds\n");
    printf("  -x <exclude_path1, exclude_path2, ...> : Exclude specific files\n");
}
```

4-3 execute_tree

```
// tree 명령어를 실행하는 함수
void execute_tree(char *command)
{
    char path[MAX_PATH] = "", option[10] = ""; // 경로와 옵션을 저장할 변수
    int show_size = 0, show_perm = 0;          // 옵션 실행 여부를 저장할 변수

    // 명령어에서 경로와 옵션을 추출
    sscanf(command, "tree %s %s", path, option);

    // 경로 길이 초과 검사
    if (strlen(path) >= MAX_PATH)
    {
        fprintf(stderr, "Error: Path length exceeds 4096 bytes.\n");
        return;
    }

    // 경로가 유효한지 검사
    struct stat statbuf1;
    if (stat(path, &statbuf1) != 0)
    {
        printf("Usage : tree <DIR_PATH> [OPTION]\n");
        return;
    }

    // 경로가 디렉토리가 아닌 경우 오류 메시지 출력
    if (!S_ISDIR(statbuf1.st_mode))
    {
        fprintf(stderr, "Error: '%s' is not a directory.\n", path);
        return;
    }
```

```
    // 경로가 홈 디렉토리 내부인지 확인
    if (!is_inside_home_directory(path))
    {
        printf("%s is outside the home directory\n", path);
        return;
    }

    // 옵션 검사 (잘못된 옵션일 경우 Usage 출력)
    if (option[0] != '\0' && strcmp(option, "-s") != 0 && strcmp(option, "-p") != 0 &&
        strcmp(option, "-sp") != 0)
    {
        printf("Usage : tree <DIR_PATH> [OPTION]\n");
        return;
    }
    // 옵션 분석
    if (strcmp(option, "-s") == 0)
    {
        show_size = 1; // 파일 크기 출력 옵션 활성화
    }
    else if (strcmp(option, "-p") == 0)
    {
        show_perm = 1; // 파일 권한 출력 옵션 활성화
    }
    else if (strcmp(option, "-sp") == 0)
    {
        show_size = 1; // 파일 크기 및 권한 출력 옵션 활성화
        show_perm = 1;
    }

    dir_count = 1;
    file_count = 0;                                          // 디렉토리 및 파일 개수 초기화
    int is_last[MAX_PATH] = {0};                              // 부모 디렉토리 마지막 여부 저장 배열
    print_tree(path, 0, show_size, show_perm, is_last);      // 트리 구조 출력
    printf("%d directories, %d files\n", dir_count, file_count); // 디렉토리 및 파일 개수 출력
}
```

4-4 print_tree

```
// 디렉토리 구조를 트리 형태로 출력하는 함수
void print_tree(const char *dir_path, int depth, int show_size, int show_perm, int is_last[])
{
    struct dirent **namelist;
    // scandir 함수를 사용하여 디렉토리 내의 파일 및 디렉토리 목록을 가져옴
    int n = scandir(dir_path, &namelist, NULL, alphasort);
    if (n < 0) // scandir 실패 시 오류 메시지 출력
    {
        perror("scandir");
        return;
    }

    int count = 0; // 현재 디렉토리 내의 파일 및 디렉토리 개수를 저장

    // 디렉토리 내의 모든 파일 및 디렉토리를 순회하며 개수를 셈
    for (int i = 0; i < n; i++)
    {
        // 파일명 길이 검사
        if (strlen(namelist[i]->d_name) >= MAX_NAME)
        {
            fprintf(stderr, "Error: File name length exceeds 256 bytes.\n");
            free(namelist[i]);
            free(namelist);
            return;
        }

        if (strcmp(namelist[i]->d_name, ".") != 0 && strcmp(namelist[i]->d_name, "..") != 0)
        {
            count++; // 현재 디렉토리와 상위 디렉토리를 제외한 개수를 셈
        }
    }
    // 현재 디렉토리의 상태 정보를 가져옴
    struct stat dir_stat;
    if (lstat(dir_path, &dir_stat) < 0)
    {
        perror("lstat");
        return;
    }

    // 현재 디렉토리의 정보 출력 (최상위 디렉토리인 경우)
    if (depth == 0)
    {
        if (show_size || show_perm)
        {
            printf("[");
            if (show_perm)
            {
```

```
            char perm_str[11];
            get_permission_string(dir_stat.st_mode, perm_str);
            printf("%s", perm_str);
        }
        if (show_size)
        {
            printf("%s%ld", show_perm ? " " : "", dir_stat.st_size); // 현재 디렉토리의 크기 출력
        }
        printf("] ");
    }
    printf("%s\n", dir_path); // 현재 디렉토리 경로 출력
}

int idx = 0; // 현재 출력할 파일 또는 디렉토리의 순서를 저장

// 디렉토리 내의 모든 파일 및 디렉토리를 순회하며 트리 구조 출력
for (int i = 0; i < n; i++)
{
    char path[MAX_PATH]; // 파일 또는 디렉토리의 전체 경로를 저장
    struct stat statbuf; // 파일 또는 디렉토리의 상태 정보를 저장
    // 파일 또는 디렉토리의 전체 경로를 생성
    snprintf(path, sizeof(path), "%s/%s", dir_path, namelist[i]->d_name);

    // 현재 디렉토리와 상위 디렉토리는 건너뜀
    if (strcmp(namelist[i]->d_name, ".") == 0 || strcmp(namelist[i]->d_name, "..") == 0)
    {
        free(namelist[i]); // 메모리 해제
        continue;
    }

    // 파일 또는 디렉토리의 상태 정보를 가져옴
    if (lstat(path, &statbuf) < 0)
    {
        perror("lstat");    // 상태 정보 가져오기 실패 시 오류 메시지 출력
        free(namelist[i]); // 메모리 해제
        continue;
    }

    // 현재 파일 또는 디렉토리가 마지막 항목인지 확인
    int is_last_item = (++idx == count);

    // 트리 구조의 들여쓰기를 출력
    for (int j = 0; j < depth; j++)
    {
        if (is_last[j])
            printf("     "); // 부모 디렉토리가 마지막 원소인 경우 빈 칸 출력
        else
            printf("|    "); // 부모 디렉토리가 마지막 원소가 아닌 경우 "|    " 출력
```

```
        }

        // 현재 파일 또는 디렉토리의 출력 형태 결정
        printf("%s── ", is_last_item ? "└" : "├");

        // 옵션에 따라 파일 크기 및 권한 정보 출력
        if (show_size || show_perm)
        {
            printf(" [");
            if (show_perm) // 권한 정보 출력
            {
                char perm_str[11];
                get_permission_string(statbuf.st_mode, perm_str);
                printf("%s", perm_str);
            }
            if (show_size) // 파일 크기 출력
            {
                printf("%s%ld", show_perm ? " " : "", statbuf.st_size);
            }
            printf("] ");
        }

        // 파일 또는 디렉토리 이름 출력
        printf("%s", namelist[i]->d_name);

        // 디렉토리인 경우 '/'를 추가하여 표시
        if (S_ISDIR(statbuf.st_mode))
        {
            printf("/");
            dir_count++; // 디렉토리 개수 증가
        }
        else
        {
            file_count++; // 파일 개수 증가
        }
        printf("\n");

        // 디렉토리인 경우 재귀적으로 탐색
        if (S_ISDIR(statbuf.st_mode))
        {
            is_last[depth] = is_last_item; // 현재 디렉토리가 마지막 원소인지 저장
            print_tree(path, depth + 1, show_size, show_perm, is_last); // 재귀 호출
        }

        free(namelist[i]); // 메모리 해제
    }
    free(namelist); // 메모리 해제
}
```

```
        }

        // 현재 파일 또는 디렉토리의 출력 형태 결정
        printf("%s── ", is_last_item ? "└" : "├");

        // 옵션에 따라 파일 크기 및 권한 정보 출력
        if (show_size || show_perm)
        {
            printf(" [");
            if (show_perm) // 권한 정보 출력
            {
                char perm_str[11];
                get_permission_string(statbuf.st_mode, perm_str);
                printf("%s", perm_str);
            }
            if (show_size) // 파일 크기 출력
            {
                printf("%s%ld", show_perm ? " " : "", statbuf.st_size);
            }
            printf("] ");
        }

        // 파일 또는 디렉토리 이름 출력
        printf("%s", namelist[i]->d_name);

        // 디렉토리인 경우 '/'를 추가하여 표시
        if (S_ISDIR(statbuf.st_mode))
        {
            printf("/");
            dir_count++; // 디렉토리 개수 증가
        }
        else
        {
            file_count++; // 파일 개수 증가
        }
        printf("\n");

        // 디렉토리인 경우 재귀적으로 탐색
        if (S_ISDIR(statbuf.st_mode))
        {
            is_last[depth] = is_last_item; // 현재 디렉토리가 마지막 원소인지 저장
            print_tree(path, depth + 1, show_size, show_perm, is_last); // 재귀 호출
        }

        free(namelist[i]); // 메모리 해제
    }
    free(namelist); // 메모리 해제
}
```

4-5 execute_arrange

```
// arrange 명령어를 실행하는 함수
void execute_arrange(const char *command)
{
    char dir_path[MAX_PATH] = "";                       // 정리할 디렉토리 경로
    char output_path[MAX_PATH] = "";                    // 출력 디렉토리 경로
    char exclude_paths[10][MAX_NAME] = {0};             // 제외할 파일/디렉토리 목록
    char extensions[10][32] = {0};                      // 허용할 확장자 목록
    int excl_count = 0, ext_count = 0, t_option = 0; // 제외 목록 개수, 확장자 개수, 시간 옵션 여부
    time_t time_limit = 0;                              // 시간 제한

    char *args[50]; // 명령어 인자를 저장할 배열
    int arg_count = 0;

    // 명령어를 공백 기준으로 나누어 인자 배열에 저장
    char *token = strtok((char *)command, " ");
    while (token != NULL && arg_count < 50)
    {
        args[arg_count++] = token;
        token = strtok(NULL, " ");
    }

    if (arg_count < 2)
    { // 인자가 충분하지 않으면 사용법 출력
        printf("Usage: arrange <DIR_PATH> [OPTION]\n");
        return;
    }

    if (strlen(args[1]) >= MAX_PATH)
    { // 경로 길이 초과 검사
        fprintf(stderr, "Error: Path length exceeds 4096 bytes.\n");
        return;
    }
    strcpy(dir_path, args[1]); // 첫 번째 인자는 디렉토리 경로

    // 경로가 존재하는지 확인
    struct stat dir_stat;
    if (stat(dir_path, &dir_stat) != 0)
    {
        printf("%s dose not exist\n", dir_path); // 경로가 존재하지 않으면 에러 메시지 출력
        return;                                  // 함수 종료
    }
    // 홈 디렉토리 바깥이면 실행하지 않음
    if (!is_inside_home_directory(dir_path))
    {
        printf("%s is outside the home directory\n", dir_path);
        return;
    }
```

```
// 기본 출력 디렉토리 설정
if (snprintf(output_path, sizeof(output_path), "%s_arranged", dir_path) >= MAX_PATH)
{
    fprintf(stderr, "Error: Output path is too long.\n");
    return;
}
// 옵션 파싱
for (int i = 2; i < arg_count; i++)
{
    if (strcmp(args[i], "-d") == 0)
    { // -d 옵션: 출력 디렉토리 지정
        if (i + 1 >= arg_count)
        {
            printf("Usage: arrange <DIR_PATH> [OPTION]\n");
            return;
        }
        strcpy(output_path, args[++i]); // 다음 인자를 출력 디렉토리로 설정
    }
    else if (strcmp(args[i], "-t") == 0)
    { // -t 옵션: 시간 제한
        if (i + 1 >= arg_count || atoi(args[i + 1]) <= 0)
        {
            printf("Invalid value for -t option\n");
            return;
        }
        t_option = 1;
        time_limit = atoi(args[++i]); // 다음 인자를 시간 제한으로 설정
    }
    else if (strcmp(args[i], "-x") == 0)
    { // -x 옵션: 제외할 파일/디렉토리
        while (i + 1 < arg_count && args[i + 1][0] != '-')
        {
            if (excl_count < 10)
            {
                strcpy(exclude_paths[excl_count++], args[++i]); // 제외 목록에 추가
            }
        }
    }
    else if (strcmp(args[i], "-e") == 0)
    { // -e 옵션: 허용할 확장자
        while (i + 1 < arg_count && args[i + 1][0] != '-')
        {
            if (ext_count < 10)
            {
                strcpy(extensions[ext_count++], args[++i]); // 확장자 목록에 추가
            }
        }
    }
```

```
        else
        {
            printf("Usage: arrange <DIR_PATH> [OPTION]\n"); // 잘못된 옵션 입력 시 사용법 출력
            return;
        }
    }

    if (stat(dir_path, &dir_stat) == -1 || !S_ISDIR(dir_stat.st_mode))
    { // 디렉토리 여부 확인
        printf("%s is not a directory\n", dir_path);
        return;
    }

    scan_directory(dir_path, t_option, time_limit, exclude_paths, excl_count, extensions, ext_count); // 디렉토리 스캔
    process_duplicates();                                                                             // 중복 파일 처리
    mkdir(output_path, 0755);                                                                         // 출력 디렉토리 생성
    organize_files(output_path);                                                                      // 파일 정리
    free_list();                                                                                      // 연결 리스트 메모리 해제

    printf("%s arranged\n", dir_path); // 정리 완료 메시지 출력
}
```

4-6 scan_directory

```
// 디렉토리를 스캔하여 파일 노드를 생성하는 함수
void scan_directory(const char *dir_path, int t_option, time_t time_limit,
                    char exclude_paths[][MAX_NAME], int excl_count,
                    char extensions[][32], int ext_count)
{
    struct dirent *entry;
    struct stat file_stat;
    DIR *dir = opendir(dir_path); // 디렉토리 열기

    if (!dir)
    {
        perror("opendir"); // 디렉토리 열기 실패 시 오류 메시지 출력
        return;
    }
    // 디렉토리 내의 모든 파일 및 디렉토리를 순회
    while ((entry = readdir(dir)))
    {
```

```
    // 현재 디렉토리와 상위 디렉토리는 건너뜀
    if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
        continue;
    // 파일명 길이 검사
    if (strlen(entry->d_name) >= MAX_NAME)
    {
        fprintf(stderr, "Error: File name length exceeds 256 bytes.\n");
        closedir(dir);
        return;
    }

    char full_path[MAX_PATH];
    snprintf(full_path, sizeof(full_path), "%s/%s", dir_path, entry->d_name); // 파일 전체 경로 생성

    // 제외할 디렉토리/파일 체크 (-x 옵션)
    int exclude_flag = 0;
    for (int i = 0; i < excl_count; i++)
    {
        if (strcmp(entry->d_name, exclude_paths[i]) == 0) // 파일/디렉토리명이 제외 목록에 있는지 확인
        {
            exclude_flag = 1;
            break;
        }
    }
    if (exclude_flag) // 제외할 목록에 있으면 건너뜀
        continue;

    // 파일 상태 정보 가져오기
    if (stat(full_path, &file_stat) == -1)
    {
        perror("stat"); // 파일 상태 정보 가져오기 실패 시 오류 메시지 출력
        continue;
    }
    // 디렉토리라면 재귀 호출
    if (S_ISDIR(file_stat.st_mode))
    {
        scan_directory(full_path, t_option, time_limit, exclude_paths, excl_count, extensions, ext_count);
    }
    else
    {
        // 파일 확장자 가져오기
        const char *ext = get_extension(entry->d_name);
        // 확장자 필터링 (-e 옵션)
        int ext_match = (ext_count == 0); // 확장자 목록이 비어있으면 모든 확장자 허용
        for (int i = 0; i < ext_count; i++)
        {
            if (strcmp(ext, extensions[i]) == 0) // 확장자와 비교
            {
```

```
                    ext_match = 1;
                    break;
                }
            }
            if (!ext_match) // 확장자 목록에 없으면 건너뜀
                continue;

            // 시간 필터 (-t 옵션)
            if (t_option && (time(NULL) - file_stat.st_mtime) > time_limit)
                continue;

            add_file_node(full_path, ext, file_stat.st_mtime); // 파일 노드 추가
        }
    }
    closedir(dir); // 디렉토리 닫기
}
```

4-7 process_duplicates

```
// 중복 파일을 처리하는 함수
void process_duplicates()
{
    FileNode *current, *compare;

    // 연결 리스트를 순회하며 중복 파일 검사
    for (current = head; current; current = current->next)
    {
        if (current->is_duplicate) // 이미 처리된 파일은 건너뜀
            continue;

        FileNode *duplicates[MAX_DUPLICATES]; // 중복 파일 저장 배열
        int count = 0;

        // 현재 파일과 동일한 파일명을 가진 모든 파일 찾기
        for (compare = current; compare; compare = compare->next)
        {
            if (compare->is_duplicate) // 이미 중복으로 처리된 파일은 건너뜀
                continue;
            const char *name1 = strrchr(current->path, '/'); // 현재 파일의 이름 추출
            const char *name2 = strrchr(compare->path, '/'); // 비교할 파일의 이름 추출
            if (name1)
                name1++;
            if (name2)
                name2++;
            if (strcmp(name1, name2) == 0) // 파일명이 같다면 중복 리스트에 추가
            {
                duplicates[count++] = compare;
            }
```

```
        }

        // 중복된 파일이 2개 이상이면 처리
        if (count > 1)
        {
            handle_duplicate(duplicates, count); // 중복 파일 처리 함수 호출
        }
    }
}
```

4-8 handle_duplicate

```
// 중복 파일을 처리하는 함수
void handle_duplicate(FileNode *duplicates[], int count)
{
    for (int i = 0; i < count; i++)
    {
        printf("%d. %s\n", i + 1, duplicates[i]->path); // 중복 파일 목록 출력
    }

    printf("Choose an option:\n");
    printf("0. select [num]\n");
    printf("1. diff [num1] [num2]\n");
    printf("2. vi [num]\n");
    printf("3. do not select\n");

    char input[16];
    char command[16];
    int num1, num2;

    while (1)
    {
        printf("20211527> ");
        fgets(input, sizeof(input), stdin);                    // 사용자 입력 받기
        int args = sscanf(input, "%s %d %d", command, &num1, &num2); // 입력 파싱

        if (strcmp(command, "select") == 0 && args == 2 && num1 > 0 && num1 <= count)
        {
            // select: 해당 파일만 유지
            for (int i = 0; i < count; i++)
            {
                if (i + 1 != num1)
                    duplicates[i]->is_duplicate = 1; // 선택된 파일 외의 파일은 중복 처리
            }
            break;
        }
```

```
        else if (strcmp(command, "diff") == 0 && args == 3 && num1 > 0 && num1 <= count && num2 > 0
&& num2 <= count)
        {
            // diff: 두 파일 비교
            pid_t pid = fork(); // 자식 프로세스 생성
            if (pid == 0)
            {
                execlp("diff", "diff", duplicates[num1 - 1]->path, duplicates[num2 - 1]->path, NULL); // diff 명
령어 실행
                perror("execlp");                                                                    // execlp
실패 시 오류 메시지 출력
                exit(1);                                                                            // 자식 프
로세스 종료
            }
            else if (pid > 0)
            {
                wait(NULL); // 자식 프로세스 종료 대기
            }
            else
            {
                perror("fork"); // fork 실패 시 오류 메시지 출력
            }
        }
        else if (strcmp(command, "vi") == 0 && args == 2 && num1 > 0 && num1 <= count)
        {
            // vi: 파일 편집
            pid_t pid = fork(); // 자식 프로세스 생성
            if (pid == 0)
            {
                execlp("vi", "vi", duplicates[num1 - 1]->path, NULL); // vi 명령어 실행
                perror("execlp");                                     // execlp 실패 시 오류 메시지 출력
                exit(1);                                              // 자식 프로세스 종료
            }
            else if (pid > 0)
            {
                wait(NULL); // 자식 프로세스 종료 대기
            }
            else
            {
                perror("fork"); // fork 실패 시 오류 메시지 출력
            }
        }
        else if (strcmp(input, "do not select\n") == 0)
        {
            // do not select: 모든 중복 파일 제거
```

```
            for (int i = 0; i < count; i++)
                duplicates[i]->is_duplicate = 1; // 모든 중복 파일을 중복 처리
            break;
        }
        else
        {
            printf("Invalid command. Try again.\n"); // 잘못된 명령어 입력 시 오류 메시지 출력
        }
    }
}
```

4-9 organize_files

```
// 파일을 확장자별로 정리하는 함수
void organize_files(const char *output_path)
{
    FileNode *current = head;

    // 연결 리스트를 순회하며 파일 정리
    while (current)
    {
        if (!current->is_duplicate)
        { // 중복 파일이 아닌 경우에만 처리
            char dest_dir[MAX_PATH];
            // 확장자별 폴더 생성
            snprintf(dest_dir, sizeof(dest_dir), "%s/%s", output_path, current->extension);
            mkdir(dest_dir, 0755); // 디렉토리 생성

            char dest_path[MAX_PATH];
            // 파일 경로에서 파일 이름만 추출
            const char *file_name = strrchr(current->path, '/');
            if (file_name == NULL)
            {
                file_name = current->path; // '/'가 없으면 경로 전체가 파일 이름
            }
            else
            {
                file_name++; // '/' 다음 부분이 파일 이름
            }
            // 대상 경로 길이 검사
            size_t dest_dir_len = strlen(dest_dir);
            size_t file_name_len = strlen(file_name);
```

```
            // 파일 이동할 경로 설정
            if (snprintf(dest_path, sizeof(dest_path), "%s/%s", dest_dir, file_name) >= MAX_PATH)
            {
                fprintf(stderr, "Error: Destination path is too long.\n");
                return;
            }

            copy_file(current->path, dest_path); // 파일 복사
            current = current->next;
        }
        else
            current = current->next; // 중복 파일은 건너뜀
    }
}
```

4-10 copy_file

```
// 파일을 복사하는 함수
void copy_file(const char *src, const char *dest)
{
    FILE *src_file = fopen(src, "rb"); // 원본 파일 열기
    if (!src_file)
    {
        perror("fopen src"); // 파일 열기 실패 시 오류 메시지 출력
        return;
    }
    FILE *dest_file = fopen(dest, "wb"); // 대상 파일 열기
    if (!dest_file)
    {
        perror("fopen dest"); // 파일 열기 실패 시 오류 메시지 출력
        fclose(src_file);       // 원본 파일 닫기
        return;
    }
    char buffer[4096]; // 파일 복사를 위한 버퍼
    size_t bytes;
    while ((bytes = fread(buffer, 1, sizeof(buffer), src_file)) > 0) // 원본 파일 읽기
    {
        fwrite(buffer, 1, bytes, dest_file); // 대상 파일에 쓰기
    }
    fclose(src_file);  // 원본 파일 닫기
    fclose(dest_file); // 대상 파일 닫기
}
```

4-11 is_extension_allowed

```
// 주어진 확장자가 허용된 확장자 목록에 있는지 확인하는 함수
int is_extension_allowed(const char *ext, char extensions[][32], int ext_count)
{
    if (ext_count == 0) // 확장자 목록이 비어있으면 모든 확장자 허용
        return 1;

    // 확장자 목록에 주어진 확장자가 있는지 확인
    for (int i = 0; i < ext_count; i++)
    {
        if (strcmp(ext, extensions[i]) == 0)
            return 1; // 확장자가 목록에 있으면 1 반환
    }
    return 0; // 확장자가 목록에 없으면 0 반환
}
```

4-12 free_list

```
// 연결 리스트의 메모리를 해제하는 함수
void free_list()
{
    FileNode *current = head;

    // 연결 리스트를 순회하며 메모리 해제
    while (current)
    {
        FileNode *temp = current;
        current = current->next;
        free(temp); // 현재 노드 메모리 해제
    }
    head = NULL; // 헤드를 NULL로 설정하여 메모리 해제 완료
}
```

4-13 get_extension

```
// 파일의 확장자를 추출하는 함수
const char *get_extension(const char *filename)
{
    const char *dot = strrchr(filename, '.');        // 파일명에서 마지막 '.'의 위치를 찾음
    return (dot && dot != filename) ? dot + 1 : ""; // 확장자가 있으면 반환, 없으면 빈 문자열 반환
}
```

4-14 add_file_node

```
// 파일 노드를 연결 리스트에 추가하는 함수
void add_file_node(const char *path, const char *extension, time_t mod_time)
{
    FileNode *new_node = (FileNode *)malloc(sizeof(FileNode)); // 새로운 노드 동적 할당
    if (!new_node)
    {
        perror("malloc"); // 메모리 할당 실패 시 오류 메시지 출력
        return;
    }
    strcpy(new_node->path, path);             // 파일 경로 저장
    strcpy(new_node->extension, extension); // 파일 확장자 저장
    new_node->mod_time = mod_time;            // 파일 수정 시간 저장
    new_node->is_duplicate = 0;               // 중복 여부 초기화
    new_node->next = head;                     // 새로운 노드를 연결 리스트의 맨 앞에 추가
    head = new_node;                           // 헤드를 새로운 노드로 변경
}
```

4-15 get_permission_string

```
// 파일 또는 디렉토리의 권한을 문자열로 변환하는 함수
void get_permission_string(mode_t mode, char *perm_str)
{
    strcpy(perm_str, "----------"); // 기본적으로 모든 권한을 '-'로 초기화
    if (S_ISDIR(mode))                // 디렉토리인 경우 'd'로 표시
        perm_str[0] = 'd';
    if (mode & S_IRUSR) // 사용자 읽기 권한이 있으면 'r'로 표시
        perm_str[1] = 'r';
    if (mode & S_IWUSR) // 사용자 쓰기 권한이 있으면 'w'로 표시
        perm_str[2] = 'w';
    if (mode & S_IXUSR) // 사용자 실행 권한이 있으면 'x'로 표시
        perm_str[3] = 'x';
    if (mode & S_IRGRP) // 그룹 읽기 권한이 있으면 'r'로 표시
        perm_str[4] = 'r';
    if (mode & S_IWGRP) // 그룹 쓰기 권한이 있으면 'w'로 표시
        perm_str[5] = 'w';
    if (mode & S_IXGRP) // 그룹 실행 권한이 있으면 'x'로 표시
        perm_str[6] = 'x';
    if (mode & S_IROTH) // 다른 사용자 읽기 권한이 있으면 'r'로 표시
        perm_str[7] = 'r';
    if (mode & S_IWOTH) // 다른 사용자 쓰기 권한이 있으면 'w'로 표시
        perm_str[8] = 'w';
    if (mode & S_IXOTH) // 다른 사용자 실행 권한이 있으면 'x'로 표시
        perm_str[9] = 'x';
}
```

4-16 is_inside_home_directory

```
// 주어진 경로가 사용자의 홈 디렉토리 내부인지 확인하는 함수
int is_inside_home_directory(const char *path)
{
    char home_path[MAX_PATH];
    const char *home_env = getenv("HOME"); // 환경 변수에서 홈 디렉토리 경로를 가져옴

    if (home_env == NULL) // 홈 디렉토리 환경 변수가 설정되지 않은 경우
    {
        fprintf(stderr, "Error: HOME environment variable is not set.\n");
        return 0;
    }
    realpath(home_env, home_path); // 홈 디렉토리의 절대 경로를 가져옴

    char resolved_path[MAX_PATH];
    if (realpath(path, resolved_path) == NULL) // 주어진 경로를 절대 경로로 변환
    {
        return 0; // 변환 실패 시 0 반환
    }

    // 변환된 경로가 홈 디렉토리 내부인지 확인
    return strncmp(resolved_path, home_path, strlen(home_path)) == 0;
}
```

```
// 파일 노드 구조체 정의
typedef struct FileNode
{
    char path[MAX_PATH];    // 파일 전체 경로
    char extension[32];     // 파일 확장자
    time_t mod_time;        // 파일 수정 시간
    int is_duplicate;       // 중복 여부 체크 변수
    struct FileNode *next; // 다음 노드를 가리키는 포인터
} FileNode;

FileNode *head = NULL; // 연결 리스트의 헤드 노드 (초기값 NULL)
```

4-17 Makefile

```
# 컴파일러 설정
CC = gcc                          # 사용할 컴파일러
CFLAGS = -Wall -Wextra -Wformat-truncation -g  # 컴파일 경고 및 디버깅 옵션
TARGET = ssu_cleanup              # 생성할 실행 파일 이름
SRC = ssu_cleanup.c               # 소스 코드 파일

# 기본 빌드 명령어 (컴파일)
all: $(TARGET)

# 실행 파일 생성 (빌드)
$(TARGET): $(SRC)
	$(CC) $(CFLAGS) -o $(TARGET) $(SRC)

# 디버깅용 컴파일 (-DDEBUG 옵션 추가)
debug: $(SRC)
	$(CC) $(CFLAGS) -DDEBUG -o $(TARGET) $(SRC)

# 실행 명령 (빌드 후 실행)
run: $(TARGET)
	./$(TARGET)

# 불필요한 파일 제거 (클린업)
clean:
	rm -f $(TARGET) *.o

# clean, all, run 등이 실제 파일과 충돌하지 않도록 설정
.PHONY: all clean debug run
```

5. **실행결과**(구현한 모든 기능 및 실행 결과 캡쳐)

5-1-0 ssu_cleanup 실행 시 프롬프트 "학번"> 출력

```
changhyeon@RSP:~$ ./ssu_cleanup
20211527>
```

5-1-1 프롬프트 상에서 지정한 내장명령어 외 기타 명령어 입력 시 help 명령어의 결과를 출력 후 프롬프트 재출력

```
20211527> EXIT
Usage:
> tree <DIR_PATH> [OPTION]...
  <none> : Display the directory structure recursively
  -s : Include the size of each file
  -p : Include the permissions of each directory and file
> arrange <DIR_PATH> [OPTION]...
  <none> : Arrange the directory
  -d <output_path> : Specify the output directory
  -t <seconds> : Only arrange files modified within given seconds
  -x <exclude_path1, exclude_path2, ...> : Exclude specific files
20211527>
```

5-1-2 프롬프트 상에서 엔터만 입력 시 프롬프트 재출력

```
20211527>
20211527>
```

5-2-0 tree 내장 명령어 실행(현재 작업 디렉토리(pwd)가 "/home/changhyeon/test1" 일 때)

```
changhyeon@RSP:~/test1$ ./ssu_cleanup
20211527> tree .
.
├── a/
│   ├── b/
│   ├── c.txt
│   └── d/
├── e.txt
├── f/
│   ├── g/
│   │   └── h.txt
│   └── i.txt
├── ssu_cleanup
└── ssu_cleanup.c
6 directories, 6 files
20211527>
```

```
20211527> tree /home/changhyeon/test1/f
/home/changhyeon/test1/f
├── g/
│   └── h.txt
└── i.txt
2 directories, 2 files
20211527>
```

5-2-1 tree 내장명령어의 –s 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/test1" 일 때)

```
changhyeon@RSP:~/test1$ ./ssu_cleanup
20211527> tree . -s
[4096] .
├── [4096] a/
│   ├── [4096] b/
│   ├── [8] c.txt
│   └── [4096] d/
├── [5] e.txt
├── [4096] f/
│   ├── [4096] g/
│   │   └── [11] h.txt
│   └── [11] i.txt
├── [26544] ssu_cleanup
└── [29515] ssu_cleanup.c
6 directories, 6 files
20211527>
```

```
20211527> tree /home/changhyeon/test1/f -s
[4096] /home/changhyeon/test1/f
├── [4096] g/
│   └── [11] h.txt
└── [11] i.txt
2 directories, 2 files
20211527>
```

5-2-2 tree 내장명령어의 –p 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/test1" 일 때)

```
20211527> tree . -p
[drwxrwxr-x] .
├── [drwxrwxr-x] a/
│   ├── [drwxrwxr-x] b/
│   ├── [-rw-rw-r--] c.txt
│   └── [drwxrwxr-x] d/
├── [-rw-rw-r--] e.txt
├── [drwxrwxr-x] f/
│   ├── [drwxrwxr-x] g/
│   │   └── [-rw-rw-r--] h.txt
│   └── [-rw-rw-r--] i.txt
├── [-rwxrwxr-x] ssu_cleanup
└── [-rw-rw-r--] ssu_cleanup.c
6 directories, 6 files
20211527>
```

```
20211527> tree /home/changhyeon/test1/f -p
[drwxrwxr-x] /home/changhyeon/test1/f
├── [drwxrwxr-x] g/
│   └── [-rw-rw-r--] h.txt
└── [-rw-rw-r--] i.txt
2 directories, 2 files
20211527>
```

5-2-3 tree 내장명령어의 –sp 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/test1" 일 때)

```
20211527> tree . -sp
[drwxrwxr-x 4096] .
├── [drwxrwxr-x 4096] a/
│   ├── [drwxrwxr-x 4096] b/
│   ├── [-rw-rw-r-- 8] c.txt
│   └── [drwxrwxr-x 4096] d/
├── [-rw-rw-r-- 5] e.txt
├── [drwxrwxr-x 4096] f/
│   ├── [drwxrwxr-x 4096] g/
│   │   └── [-rw-rw-r-- 11] h.txt
│   └── [-rw-rw-r-- 11] i.txt
├── [-rwxrwxr-x 26544] ssu_cleanup
└── [-rw-rw-r-- 29515] ssu_cleanup.c
6 directories, 6 files
20211527>
```

```
20211527> tree /home/changhyeon/test1/f -sp
[drwxrwxr-x 4096] /home/changhyeon/test1/f
├── [drwxrwxr-x 4096] g/
│   └── [-rw-rw-r-- 11] h.txt
└── [-rw-rw-r-- 11] i.txt
2 directories, 2 files
20211527>
```

5-2-4 첫 번째 인자로 올바르지 않은 경로(존재하지 않는 디렉토리) 입력 시 Usage 출력 후 프롬프트 재출력

```
20211527> tree /home/changhyeon/test100
Usage : tree <DIR_PATH> [OPTION]
20211527>
```

5-2-5 첫 번째 인자로 입력받은 경로(절대경로 또는 상대경로)가 길이 제한(4096 Byte)을 넘어선 경우, 그리고 절대경로 내 한 파일 및 디렉토리 이름의 길이 제한 을 (256 Byte)넘어선 경우 에러 처리 또한 해당 경로가 디렉토리가 아니거나 입력받은 경로(절대 경로또는 상대경로)가 사용자의 홈 디렉토리($HOME 또는 ~) 를 벗어나는 경우 에러 처리 후 프롬프트 재출력

```
/ test1 / 1234567890123...345678901234
```

```
20211527> tree /home/changhyeon/test1
Error: File name length exceeds 256 bytes.
```

```
20211527> tree /home/changhyeon/test1/f/i.txt
Error: '/home/changhyeon/test1/f/i.txt' is not a directory.
20211527>
```

5-2-6 첫 번째 인자로 입력받은 경로(절대 경로)가 사용자의 홈 디렉토리 ($HOME, ~)를 벗어나는 경우 "<입력 받은 경로> is outsidethe home directory" 표준 출력 후 프롬프트 재출력

```
20211527> tree /etc
/etc is outside the home directory
20211527>
```

5-2-7 두 번째 인자로 올바르지 않은 옵션이 들어왔을 경우 출력 Usage후 프롬프트 재출력

```
20211527> tree /home/changhyeon/test1 -x
Usage : tree <DIR_PATH> [OPTION]
20211527>
```

//

```
20211527> tree /home/changhyeon/test2
/home/changhyeon/test2
├── a/
│   ├── b/
│   ├── c.c
│   └── d/
├── e.c
└── f/
    ├── g/
    │   └── h.c
    └── i.c
6 directories, 4 files
```

```
20211527> tree /home/changhyeon/test3
/home/changhyeon/test3
├── a/
│   ├── b/
│   ├── c.txt
│   └── d/
├── e.txt
└── f/
    ├── g/
    │   └── h.c
    └── i.c
6 directories, 4 files
```

```
20211527> tree /home/changhyeon/test4
/home/changhyeon/test4
├── a/
│   ├── abc.txt
│   └── hello.txt
└── b/
    └── hello.txt
3 directories, 3 files
```

5-3-0 arrange 내장명령어 실행 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

```
changhyeon@RSP:~$ ./ssu_cleanup
20211527> arrange
Usage: arrange <DIR_PATH> [OPTION]
20211527>
```

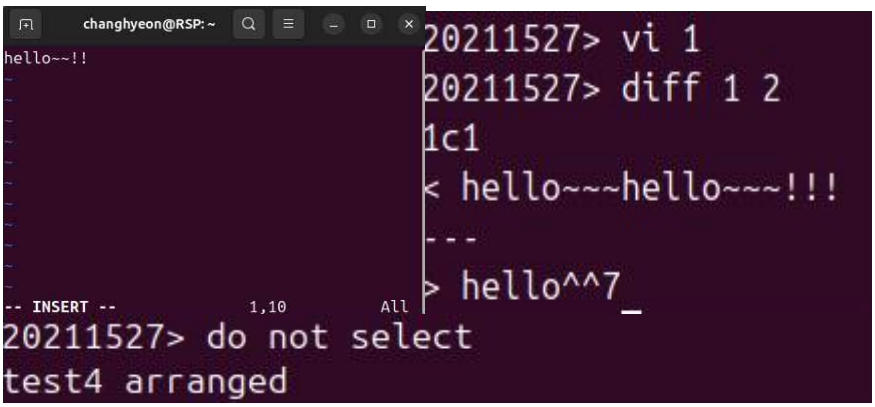
```
20211527> arrange hello
hello dose not exist
20211527>
```

```
20211527> arrange /home/changhyeon/test2/e.c
/home/changhyeon/test2/e.c is not a directory
20211527>
```

```
20211527> arrange /home/changhyeon/test2
/home/changhyeon/test2 arranged
20211527> tree /home/changhyeon/test2_arranged
/home/changhyeon/test2_arranged
└── c/
    ├── c.c
    ├── e.c
    ├── h.c
    └── i.c
2 directories, 4 files
20211527>
```

```
20211527> arrange test3
test3 arranged
20211527> tree /home/changhyeon/test3_arranged
/home/changhyeon/test3_arranged
├── c/
│   ├── h.c
│   └── i.c
└── txt/
    ├── c.txt
    └── e.txt
3 directories, 4 files
20211527>
```

```
20211527> arrange test4
1. test4/a/hello.txt
2. test4/b/hello.txt
Choose an option:
0. select [num]
1. diff [num1] [num2]
2. vi [num]
3. do not select
20211527> diff 1 2
1c1
< hello~~!!
---
> hello^^7
20211527>
```




```
20211527> tree test4_arranged
test4_arranged
└── txt/
    └── abc.txt
2 directories, 1 files
```

5-3-1 arrange 내장명령어의 –d 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

```
20211527> arrange test2 -d hello
test2 arranged
20211527> tree hello
hello
└── c/
    ├── c.c
    ├── e.c
    ├── h.c
    └── i.c
2 directories, 4 files
20211527>
```

5-3-2 arrange 내장명령어의 –t 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

```
20211527> arrange test2 -t 30
test2 arranged
20211527> tree test2_arranged
test2_arranged
└── c/
    ├── c.c
    └── e.c
2 directories, 2 files
20211527>
```

5-3-3 arrange 내장명령어의 –x 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

```
20211527> arrange test2 -x f a
test2 arranged
20211527> tree test2_arranged
test2_arranged
└── c/
    └── e.c
2 directories, 1 files
20211527>
```

5-3-4 arrange 내장명령어의 –e 옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

```
20211527> arrange test3 -e c
test3 arranged
20211527> tree test3_arranged
test3_arranged
└── c/
    ├── h.c
    └── i.c
2 directories, 2 files
20211527>
```

5-3-5 arrange 내장명령어 다중옵션 (현재 작업 디렉토리(pwd)가 "/home/changhyeon/" 일 때)

// test5/a/abc.txt, test5/a/hello.txt, test5/c/hello.txt 는 30초 이내에 수정되었음

```
20211527> tree test5
test5
├── a/
│   ├── abc.txt
│   └── hello.txt
├── b/
│   └── hello.txt
└── c/
    ├── h.h
    └── hello.txt
4 directories, 5 files
```

```
20211527> arrange test5 -d hello2 -t 30 -x b -e txt
1. test5/a/hello.txt
2. test5/c/hello.txt
Choose an option:
0. select [num]
1. diff [num1] [num2]
2. vi [num]
3. do not select
20211527> diff 1 2
1c1
< hello~~~hello~~~!!!12
---
> 1234567890hello321
20211527> select 1
test5 arranged
```

```
20211527> tree hello2
hello2
└── txt/
    ├── abc.txt
    └── hello.txt
2 directories, 2 files
20211527>
```

5-3-6 첫 번째 인자 입력이 없는 경우 Usage 출력 후 프롬프트 재출력

```
20211527> arrange
Usage: arrange <DIR_PATH> [OPTION]
20211527>
```

5-3-7 첫 번째 인자로 입력받은 경로(절대경로 또는 상대경로)가 길이 제한(4096 Byte)을 넘어선 경우, 그리고 절대경로 내 한 파일 및 디렉토리 이름의 길이 제한 을 (256 Byte)넘어선 경우 에러 처리 또한 해당 경로가 디렉토리가 아니거나 입력받은 경로(절대 경로 또는 상대경로)가 사용자의 홈 디렉토리($HOME 또는 ~) 를 벗어나는 경우 에러 처리 후 프롬프트 재출력

```
20211527> arrange test1
Error: File name length exceeds 256 bytes.
```

```
20211527> arrange /etc
/etc is outside the home directory
20211527>
```

5-3-8 두 번째 인자로 올바르지 않은 옵션이 들어왔을 경우 출력 Usage 후 프롬프트 재출력

```
20211527> arrange test5 -w
Usage: arrange <DIR_PATH> [OPTION]
20211527>
```

5-4-0 help 내장명령어

```
20211527> help
Usage:
> tree <DIR_PATH> [OPTION]...
  <none> : Display the directory structure recursively
  -s : Include the size of each file
  -p : Include the permissions of each directory and file
> arrange <DIR_PATH> [OPTION]...
  <none> : Arrange the directory
  -d <output_path> : Specify the output directory
  -t <seconds> : Only arrange files modified within given seconds
  -x <exclude_path1, exclude_path2, ...> : Exclude specific files
20211527>
```

5-5-0 exit 내장명령어

```
20211527> exit
changhyeon@RSP:~$
```

전체 소스코드 <ssu_cleanup.c>

```
#define _GNU_SOURCE     // GNU 확장 기능 사용을 위한 매크로 정의
#include <stdio.h>      // 표준 입출력 함수를 사용하기 위한 헤더 파일
#include <stdlib.h>     // 메모리 할당 및 프로그램 종료 함수를 사용하기 위한 헤더 파일
#include <string.h>     // 문자열 처리 함수를 사용하기 위한 헤더 파일
#include <dirent.h>     // 디렉토리 관련 함수를 사용하기 위한 헤더 파일
#include <sys/stat.h>   // 파일 상태 정보를 다루기 위한 헤더 파일
#include <sys/types.h> // 시스템 데이터 타입 정의를 위한 헤더 파일
#include <sys/wait.h>   // 프로세스 관련 함수를 사용하기 위한 헤더 파일
#include <unistd.h>     // POSIX 운영체제 API를 사용하기 위한 헤더 파일
#include <limits.h>     // 시스템 제한값을 정의하는 상수를 사용하기 위한 헤더 파일
#include <time.h>       // 시간 관련 함수를 사용하기 위한 헤더 파일
#include <errno.h>      // 오류 번호를 다루기 위한 헤더 파일

#define MAX_PATH 4096       // 경로의 최대 길이를 정의
#define MAX_NAME 255        // 파일 또는 디렉토리 이름의 최대 길이를 정의
#define MAX_DUPLICATES 100 // 중복 파일 처리 시 최대 허용 개수

int dir_count = 1, file_count = 0; // 디렉토리와 파일 개수를 저장하는 전역 변수

// 프로그램 사용법을 출력하는 함수
void print_help()
{
    printf("Usage:₩n");
    printf("> tree <DIR_PATH> [OPTION]...₩n");
    printf("  <none> : Display the directory structure recursively₩n");
    printf("  -s : Include the size of each file₩n");
    printf("  -p : Include the permissions of each directory and file₩n");
    printf("> arrange <DIR_PATH> [OPTION]...₩n");
    printf("  <none> : Arrange the directory₩n");
    printf("  -d <output_path> : Specify the output directory₩n");
    printf("  -t <seconds> : Only arrange files modified within given seconds₩n");
    printf("  -x <exclude_path1, exclude_path2, ...> : Exclude specific files₩n");
}

// 파일 또는 디렉토리의 권한을 문자열로 변환하는 함수
void get_permission_string(mode_t mode, char *perm_str)
{
    strcpy(perm_str, "----------"); // 기본적으로 모든 권한을 '-'로 초기화
    if (S_ISDIR(mode))                  // 디렉토리인 경우 'd'로 표시
        perm_str[0] = 'd';
    if (mode & S_IRUSR) // 사용자 읽기 권한이 있으면 'r'로 표시
        perm_str[1] = 'r';
```

```
    if (mode & S_IWUSR) // 사용자 쓰기 권한이 있으면 'w'로 표시
        perm_str[2] = 'w';
    if (mode & S_IXUSR) // 사용자 실행 권한이 있으면 'x'로 표시
        perm_str[3] = 'x';
    if (mode & S_IRGRP) // 그룹 읽기 권한이 있으면 'r'로 표시
        perm_str[4] = 'r';
    if (mode & S_IWGRP) // 그룹 쓰기 권한이 있으면 'w'로 표시
        perm_str[5] = 'w';
    if (mode & S_IXGRP) // 그룹 실행 권한이 있으면 'x'로 표시
        perm_str[6] = 'x';
    if (mode & S_IROTH) // 다른 사용자 읽기 권한이 있으면 'r'로 표시
        perm_str[7] = 'r';
    if (mode & S_IWOTH) // 다른 사용자 쓰기 권한이 있으면 'w'로 표시
        perm_str[8] = 'w';
    if (mode & S_IXOTH) // 다른 사용자 실행 권한이 있으면 'x'로 표시
        perm_str[9] = 'x';
}

// 주어진 경로가 사용자의 홈 디렉토리 내부인지 확인하는 함수
int is_inside_home_directory(const char *path)
{
    char home_path[MAX_PATH];
    const char *home_env = getenv("HOME"); // 환경 변수에서 홈 디렉토리 경로를 가져옴

    if (home_env == NULL) // 홈 디렉토리 환경 변수가 설정되지 않은 경우
    {
        fprintf(stderr, "Error: HOME environment variable is not set.₩n");
        return 0;
    }
    realpath(home_env, home_path); // 홈 디렉토리의 절대 경로를 가져옴

    char resolved_path[MAX_PATH];
    if (realpath(path, resolved_path) == NULL) // 주어진 경로를 절대 경로로 변환
    {
        return 0; // 변환 실패 시 0 반환
    }

    // 변환된 경로가 홈 디렉토리 내부인지 확인
    return strncmp(resolved_path, home_path, strlen(home_path)) == 0;
}

// 디렉토리 구조를 트리 형태로 출력하는 함수
void print_tree(const char *dir_path, int depth, int show_size, int show_perm, int is_last[])
```

```
{
    struct dirent **namelist;
    // scandir 함수를 사용하여 디렉토리 내의 파일 및 디렉토리 목록을 가져옴
    int n = scandir(dir_path, &namelist, NULL, alphasort);
    if (n < 0) // scandir 실패 시 오류 메시지 출력
    {
        perror("scandir");
        return;
    }

    int count = 0; // 현재 디렉토리 내의 파일 및 디렉토리 개수를 저장

    // 디렉토리 내의 모든 파일 및 디렉토리를 순회하며 개수를 셈
    for (int i = 0; i < n; i++)
    {
        // 파일명 길이 검사
        if (strlen(namelist[i]->d_name) >= MAX_NAME)
        {
            fprintf(stderr, "Error: File name length exceeds 256 bytes.₩n");
            free(namelist[i]);
            free(namelist);
            return;
        }

        if (strcmp(namelist[i]->d_name, ".") != 0 && strcmp(namelist[i]->d_name, "..") != 0)
        {
            count++; // 현재 디렉토리와 상위 디렉토리를 제외한 개수를 셈
        }
    }

    // 현재 디렉토리의 상태 정보를 가져옴
    struct stat dir_stat;
    if (lstat(dir_path, &dir_stat) < 0)
    {
        perror("lstat");
        return;
    }

    // 현재 디렉토리의 정보 출력 (최상위 디렉토리인 경우)
    if (depth == 0)
    {
        if (show_size || show_perm)
        {
```

```
            printf("[");
            if (show_perm)
            {
                char perm_str[11];
                get_permission_string(dir_stat.st_mode, perm_str);
                printf("%s", perm_str);
            }
            if (show_size)
            {
                printf("%s%ld", show_perm ? " " : "", dir_stat.st_size); // 현재 디렉토리의 크기 출력
            }
            printf("] ");
        }
        printf("%s₩n", dir_path); // 현재 디렉토리 경로 출력
}

int idx = 0; // 현재 출력할 파일 또는 디렉토리의 순서를 저장

// 디렉토리 내의 모든 파일 및 디렉토리를 순회하며 트리 구조 출력
for (int i = 0; i < n; i++)
{
    char path[MAX_PATH]; // 파일 또는 디렉토리의 전체 경로를 저장
    struct stat statbuf; // 파일 또는 디렉토리의 상태 정보를 저장
    // 파일 또는 디렉토리의 전체 경로를 생성
    snprintf(path, sizeof(path), "%s/%s", dir_path, namelist[i]->d_name);

    // 현재 디렉토리와 상위 디렉토리는 건너뜀
    if (strcmp(namelist[i]->d_name, ".") == 0 || strcmp(namelist[i]->d_name, "..") == 0)
    {
        free(namelist[i]); // 메모리 해제
        continue;
    }

    // 파일 또는 디렉토리의 상태 정보를 가져옴
    if (lstat(path, &statbuf) < 0)
    {
        perror("lstat");    // 상태 정보 가져오기 실패 시 오류 메시지 출력
        free(namelist[i]); // 메모리 해제
        continue;
    }

    // 현재 파일 또는 디렉토리가 마지막 항목인지 확인
    int is_last_item = (++idx == count);
```

```
// 트리 구조의 들여쓰기를 출력
for (int j = 0; j < depth; j++)
{
    if (is_last[j])
        printf("    "); // 부모 디렉토리가 마지막 원소인 경우 빈 칸 출력
    else
        printf("│   "); // 부모 디렉토리가 마지막 원소가 아닌 경우 "│   " 출력
}

// 현재 파일 또는 디렉토리의 출력 형태 결정
printf("%s── ", is_last_item ? "└" : "├");

// 옵션에 따라 파일 크기 및 권한 정보 출력
if (show_size || show_perm)
{
    printf(" [");
    if (show_perm) // 권한 정보 출력
    {
        char perm_str[11];
        get_permission_string(statbuf.st_mode, perm_str);
        printf("%s", perm_str);
    }
    if (show_size) // 파일 크기 출력
    {
        printf("%s%ld", show_perm ? " " : "", statbuf.st_size);
    }
    printf("] ");
}

// 파일 또는 디렉토리 이름 출력
printf("%s", namelist[i]->d_name);

// 디렉토리인 경우 '/'를 추가하여 표시
if (S_ISDIR(statbuf.st_mode))
{
    printf("/");
    dir_count++; // 디렉토리 개수 증가
}
else
{
    file_count++; // 파일 개수 증가
}
```

```
            printf("₩n");

            // 디렉토리인 경우 재귀적으로 탐색
            if (S_ISDIR(statbuf.st_mode))
            {
                is_last[depth] = is_last_item; // 현재 디렉토리가 마지막 원소인지 저장
                print_tree(path, depth + 1, show_size, show_perm, is_last); // 재귀 호출
            }

            free(namelist[i]); // 메모리 해제
        }
        free(namelist); // 메모리 해제
}

// tree 명령어를 실행하는 함수
void execute_tree(char *command)
{
        char path[MAX_PATH] = "", option[10] = ""; // 경로와 옵션을 저장할 변수
        int show_size = 0, show_perm = 0;              // 옵션 실행 여부를 저장할 변수

        // 명령어에서 경로와 옵션을 추출
        sscanf(command, "tree %s %s", path, option);

        // 경로 길이 초과 검사
        if (strlen(path) >= MAX_PATH)
        {
            fprintf(stderr, "Error: Path length exceeds 4096 bytes.₩n");
            return;
        }

        // 경로가 유효한지 검사
        struct stat statbuf1;
        if (stat(path, &statbuf1) != 0)
        {
            printf("Usage : tree <DIR_PATH> [OPTION]₩n");
            return;
        }

        // 경로가 디렉토리가 아닌 경우 오류 메시지 출력
        if (!S_ISDIR(statbuf1.st_mode))
        {
            fprintf(stderr, "Error: '%s' is not a directory.₩n", path);
            return;
```

```
    }

    // 경로가 홈 디렉토리 내부인지 확인
    if (!is_inside_home_directory(path))
    {
        printf("%s is outside the home directory₩n", path);
        return;
    }

    // 옵션 검사 (잘못된 옵션일 경우 Usage 출력)
    if (option[0] != '₩0' && strcmp(option, "-s") != 0 && strcmp(option, "-p") != 0 &&
        strcmp(option, "-sp") != 0)
    {
        printf("Usage : tree <DIR_PATH> [OPTION]₩n");
        return;
    }

    // 옵션 분석
    if (strcmp(option, "-s") == 0)
    {
        show_size = 1; // 파일 크기 출력 옵션 활성화
    }
    else if (strcmp(option, "-p") == 0)
    {
        show_perm = 1; // 파일 권한 출력 옵션 활성화
    }
    else if (strcmp(option, "-sp") == 0)
    {
        show_size = 1; // 파일 크기 및 권한 출력 옵션 활성화
        show_perm = 1;
    }

    dir_count = 1;
    file_count = 0;                                          // 디렉토리 및 파일 개수 초기화
    int is_last[MAX_PATH] = {0};                             // 부모 디렉토리 마지막 여부 저장 배열
    print_tree(path, 0, show_size, show_perm, is_last);      // 트리 구조 출력
    printf("%d directories, %d files₩n", dir_count, file_count); // 디렉토리 및 파일 개수 출력
}

// 파일 노드 구조체 정의
typedef struct FileNode
{
    char path[MAX_PATH];    // 파일 전체 경로
```

```
    char extension[32];      // 파일 확장자
    time_t mod_time;         // 파일 수정 시간
    int is_duplicate;        // 중복 여부 체크 변수
    struct FileNode *next; // 다음 노드를 가리키는 포인터
} FileNode;

FileNode *head = NULL; // 연결 리스트의 헤드 노드 (초기값 NULL)

// 파일 노드를 연결 리스트에 추가하는 함수
void add_file_node(const char *path, const char *extension, time_t mod_time)
{
    FileNode *new_node = (FileNode *)malloc(sizeof(FileNode)); // 새로운 노드 동적 할당
    if (!new_node)
    {
        perror("malloc"); // 메모리 할당 실패 시 오류 메시지 출력
        return;
    }
    strcpy(new_node->path, path);               // 파일 경로 저장
    strcpy(new_node->extension, extension); // 파일 확장자 저장
    new_node->mod_time = mod_time;              // 파일 수정 시간 저장
    new_node->is_duplicate = 0;                 // 중복 여부 초기화
    new_node->next = head;                      // 새로운 노드를 연결 리스트의 맨 앞에 추가
    head = new_node;                            // 헤드를 새로운 노드로 변경
}

// 파일의 확장자를 추출하는 함수
const char *get_extension(const char *filename)
{
    const char *dot = strrchr(filename, '.');          // 파일명에서 마지막 '.'의 위치를 찾음
    return (dot && dot != filename) ? dot + 1 : ""; // 확장자가 있으면 반환, 없으면 빈 문자열 반환
}

// 연결 리스트의 메모리를 해제하는 함수
void free_list()
{
    FileNode *current = head;

    // 연결 리스트를 순회하며 메모리 해제
    while (current)
    {
        FileNode *temp = current;
        current = current->next;
        free(temp); // 현재 노드 메모리 해제
```

```
    }
    head = NULL; // 헤드를 NULL로 설정하여 메모리 해제 완료
}

// 주어진 확장자가 허용된 확장자 목록에 있는지 확인하는 함수
int is_extension_allowed(const char *ext, char extensions[][32], int ext_count)
{
    if (ext_count == 0) // 확장자 목록이 비어있으면 모든 확장자 허용
        return 1;

    // 확장자 목록에 주어진 확장자가 있는지 확인
    for (int i = 0; i < ext_count; i++)
    {
        if (strcmp(ext, extensions[i]) == 0)
            return 1; // 확장자가 목록에 있으면 1 반환
    }
    return 0; // 확장자가 목록에 없으면 0 반환
}

// 파일을 복사하는 함수
void copy_file(const char *src, const char *dest)
{
    FILE *src_file = fopen(src, "rb"); // 원본 파일 열기
    if (!src_file)
    {
        perror("fopen src"); // 파일 열기 실패 시 오류 메시지 출력
        return;
    }
    FILE *dest_file = fopen(dest, "wb"); // 대상 파일 열기
    if (!dest_file)
    {
        perror("fopen dest"); // 파일 열기 실패 시 오류 메시지 출력
        fclose(src_file);       // 원본 파일 닫기
        return;
    }
    char buffer[4096]; // 파일 복사를 위한 버퍼
    size_t bytes;
    while ((bytes = fread(buffer, 1, sizeof(buffer), src_file)) > 0) // 원본 파일 읽기
    {
        fwrite(buffer, 1, bytes, dest_file); // 대상 파일에 쓰기
    }
    fclose(src_file);  // 원본 파일 닫기
    fclose(dest_file); // 대상 파일 닫기
```

```
}

// 파일을 확장자별로 정리하는 함수
void organize_files(const char *output_path)
{
    FileNode *current = head;

    // 연결 리스트를 순회하며 파일 정리
    while (current)
    {
        if (!current->is_duplicate)
        { // 중복 파일이 아닌 경우에만 처리
            char dest_dir[MAX_PATH];
            // 확장자별 폴더 생성
            snprintf(dest_dir, sizeof(dest_dir), "%s/%s", output_path, current->extension);
            mkdir(dest_dir, 0755); // 디렉토리 생성

            char dest_path[MAX_PATH];
            // 파일 경로에서 파일 이름만 추출
            const char *file_name = strrchr(current->path, '/');
            if (file_name == NULL)
            {
                file_name = current->path; // '/'가 없으면 경로 전체가 파일 이름
            }
            else
            {
                file_name++; // '/' 다음 부분이 파일 이름
            }

            // 대상 경로 길이 검사
            size_t dest_dir_len = strlen(dest_dir);
            size_t file_name_len = strlen(file_name);

            // 파일 이동할 경로 설정
            if (snprintf(dest_path, sizeof(dest_path), "%s/%s", dest_dir, file_name) >= MAX_PATH)
            {
                fprintf(stderr, "Error: Destination path is too long.₩n");
                return;
            }

            copy_file(current->path, dest_path); // 파일 복사
            current = current->next;
        }
```

```
        else
            current = current->next; // 중복 파일은 건너뜀
    }
}

// 중복 파일을 처리하는 함수
void handle_duplicate(FileNode *duplicates[], int count)
{
    for (int i = 0; i < count; i++)
    {
        printf("%d. %s₩n", i + 1, duplicates[i]->path); // 중복 파일 목록 출력
    }

    printf("Choose an option:₩n");
    printf("0. select [num]₩n");
    printf("1. diff [num1] [num2]₩n");
    printf("2. vi [num]₩n");
    printf("3. do not select₩n");

    char input[16];
    char command[16];
    int num1, num2;

    while (1)
    {
        printf("20211527> ");
        fgets(input, sizeof(input), stdin);                    // 사용자 입력 받기
        int args = sscanf(input, "%s %d %d", command, &num1, &num2); // 입력 파싱

        if (strcmp(command, "select") == 0 && args == 2 && num1 > 0 && num1 <= count)
        {
            // select: 해당 파일만 유지
            for (int i = 0; i < count; i++)
            {
                if (i + 1 != num1)
                    duplicates[i]->is_duplicate = 1; // 선택된 파일 외의 파일은 중복 처리
            }
            break;
        }
        else if (strcmp(command, "diff") == 0 && args == 3 && num1 > 0 && num1 <= count && num2 > 0
&& num2 <= count)
        {
            // diff: 두 파일 비교
            pid_t pid = fork(); // 자식 프로세스 생성
```

```
            if (pid == 0)
            {
                execlp("diff", "diff", duplicates[num1 - 1]->path, duplicates[num2 - 1]->path, NULL); // diff 명령어 실행
                perror("execlp");                                                                    // execlp 실패 시 오류 메시지 출력
                exit(1);                                                                             // 자식 프로세스 종료
            }
            else if (pid > 0)
            {
                wait(NULL); // 자식 프로세스 종료 대기
            }
            else
            {
                perror("fork"); // fork 실패 시 오류 메시지 출력
            }
        }
        else if (strcmp(command, "vi") == 0 && args == 2 && num1 > 0 && num1 <= count)
        {
            // vi: 파일 편집
            pid_t pid = fork(); // 자식 프로세스 생성
            if (pid == 0)
            {
                execlp("vi", "vi", duplicates[num1 - 1]->path, NULL); // vi 명령어 실행
                perror("execlp");                                     // execlp 실패 시 오류 메시지 출력
                exit(1);                                              // 자식 프로세스 종료
            }
            else if (pid > 0)
            {
                wait(NULL); // 자식 프로세스 종료 대기
            }
            else
            {
                perror("fork"); // fork 실패 시 오류 메시지 출력
            }
        }
        else if (strcmp(input, "do not select₩n") == 0)
        {
            // do not select: 모든 중복 파일 제거
            for (int i = 0; i < count; i++)
                duplicates[i]->is_duplicate = 1; // 모든 중복 파일을 중복 처리
            break;
        }
        else
```

```
        {
            printf("Invalid command. Try again.₩n"); // 잘못된 명령어 입력 시 오류 메시지 출력
        }
    }
}

// 중복 파일을 처리하는 함수
void process_duplicates()
{
    FileNode *current, *compare;

    // 연결 리스트를 순회하며 중복 파일 검사
    for (current = head; current; current = current->next)
    {
        if (current->is_duplicate) // 이미 처리된 파일은 건너뜀
            continue;

        FileNode *duplicates[MAX_DUPLICATES]; // 중복 파일 저장 배열
        int count = 0;

        // 현재 파일과 동일한 파일명을 가진 모든 파일 찾기
        for (compare = current; compare; compare = compare->next)
        {
            if (compare->is_duplicate) // 이미 중복으로 처리된 파일은 건너뜀
                continue;

            const char *name1 = strrchr(current->path, '/'); // 현재 파일의 이름 추출
            const char *name2 = strrchr(compare->path, '/'); // 비교할 파일의 이름 추출
            if (name1)
                name1++;
            if (name2)
                name2++;

            if (strcmp(name1, name2) == 0) // 파일명이 같다면 중복 리스트에 추가
            {
                duplicates[count++] = compare;
            }
        }

        // 중복된 파일이 2개 이상이면 처리
        if (count > 1)
        {
            handle_duplicate(duplicates, count); // 중복 파일 처리 함수 호출
```

```
        }
    }
}

// 디렉토리를 스캔하여 파일 노드를 생성하는 함수
void scan_directory(const char *dir_path, int t_option, time_t time_limit,
                    char exclude_paths[][MAX_NAME], int excl_count,
                    char extensions[][32], int ext_count)
{
    struct dirent *entry;
    struct stat file_stat;
    DIR *dir = opendir(dir_path); // 디렉토리 열기

    if (!dir)
    {
        perror("opendir"); // 디렉토리 열기 실패 시 오류 메시지 출력
        return;
    }

    // 디렉토리 내의 모든 파일 및 디렉토리를 순회
    while ((entry = readdir(dir)))
    {
        // 현재 디렉토리와 상위 디렉토리는 건너뜀
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
            continue;
        // 파일명 길이 검사
        if (strlen(entry->d_name) >= MAX_NAME)
        {
            fprintf(stderr, "Error: File name length exceeds 256 bytes.₩n");
            closedir(dir);
            return;
        }

        char full_path[MAX_PATH];
        snprintf(full_path, sizeof(full_path), "%s/%s", dir_path, entry->d_name); // 파일 전체 경로 생성

        // 제외할 디렉토리/파일 체크 (-x 옵션)
        int exclude_flag = 0;
        for (int i = 0; i < excl_count; i++)
        {
            if (strcmp(entry->d_name, exclude_paths[i]) == 0) // 파일/디렉토리명이 제외 목록에 있는지 확인
            {
                exclude_flag = 1;
```

```
            break;
        }
    }
    if (exclude_flag) // 제외할 목록에 있으면 건너뜀
        continue;

    // 파일 상태 정보 가져오기
    if (stat(full_path, &file_stat) == -1)
    {
        perror("stat"); // 파일 상태 정보 가져오기 실패 시 오류 메시지 출력
        continue;
    }

    // 디렉토리라면 재귀 호출
    if (S_ISDIR(file_stat.st_mode))
    {
        scan_directory(full_path, t_option, time_limit, exclude_paths, excl_count, extensions, ext_count);
    }
    else
    {
        // 파일 확장자 가져오기
        const char *ext = get_extension(entry->d_name);

        // 확장자 필터링 (-e 옵션)
        int ext_match = (ext_count == 0); // 확장자 목록이 비어있으면 모든 확장자 허용
        for (int i = 0; i < ext_count; i++)
        {
            if (strcmp(ext, extensions[i]) == 0) // 확장자와 비교
            {
                ext_match = 1;
                break;
            }
        }
        if (!ext_match) // 확장자 목록에 없으면 건너뜀
            continue;

        // 시간 필터 (-t 옵션)
        if (t_option && (time(NULL) - file_stat.st_mtime) > time_limit)
            continue;

        add_file_node(full_path, ext, file_stat.st_mtime); // 파일 노드 추가
    }
}
```

```
    closedir(dir); // 디렉토리 닫기
}

// arrange 명령어를 실행하는 함수
void execute_arrange(const char *command)
{
    char dir_path[MAX_PATH] = "";                       // 정리할 디렉토리 경로
    char output_path[MAX_PATH] = "";                    // 출력 디렉토리 경로
    char exclude_paths[10][MAX_NAME] = {0};             // 제외할 파일/디렉토리 목록
    char extensions[10][32] = {0};                      // 허용할 확장자 목록
    int excl_count = 0, ext_count = 0, t_option = 0; // 제외 목록 개수, 확장자 개수, 시간 옵션 여부
    time_t time_limit = 0;                              // 시간 제한

    char *args[50]; // 명령어 인자를 저장할 배열
    int arg_count = 0;

    // 명령어를 공백 기준으로 나누어 인자 배열에 저장
    char *token = strtok((char *)command, " ");
    while (token != NULL && arg_count < 50)
    {
        args[arg_count++] = token;
        token = strtok(NULL, " ");
    }

    if (arg_count < 2)
    { // 인자가 충분하지 않으면 사용법 출력
        printf("Usage: arrange <DIR_PATH> [OPTION]₩n");
        return;
    }

    if (strlen(args[1]) >= MAX_PATH)
    { // 경로 길이 초과 검사
        fprintf(stderr, "Error: Path length exceeds 4096 bytes.₩n");
        return;
    }

    strcpy(dir_path, args[1]); // 첫 번째 인자는 디렉토리 경로

    // 경로가 존재하는지 확인
    struct stat dir_stat;
    if (stat(dir_path, &dir_stat) != 0)
    {
        printf("%s dose not exist₩n", dir_path); // 경로가 존재하지 않으면 에러 메시지 출력
```

```
    return;                                    // 함수 종료
}

// 홈 디렉토리 바깥이면 실행하지 않음
if (!is_inside_home_directory(dir_path))
{
    printf("%s is outside the home directory₩n", dir_path);
    return;
}

// 기본 출력 디렉토리 설정
if (snprintf(output_path, sizeof(output_path), "%s_arranged", dir_path) >= MAX_PATH)
{
    fprintf(stderr, "Error: Output path is too long.₩n");
    return;
}

// 옵션 파싱
for (int i = 2; i < arg_count; i++)
{
    if (strcmp(args[i], "-d") == 0)
    { // -d 옵션: 출력 디렉토리 지정
        if (i + 1 >= arg_count)
        {
            printf("Usage: arrange <DIR_PATH> [OPTION]₩n");
            return;
        }
        strcpy(output_path, args[++i]); // 다음 인자를 출력 디렉토리로 설정
    }
    else if (strcmp(args[i], "-t") == 0)
    { // -t 옵션: 시간 제한
        if (i + 1 >= arg_count || atoi(args[i + 1]) <= 0)
        {
            printf("Invalid value for -t option₩n");
            return;
        }
        t_option = 1;
        time_limit = atoi(args[++i]); // 다음 인자를 시간 제한으로 설정
    }
    else if (strcmp(args[i], "-x") == 0)
    { // -x 옵션: 제외할 파일/디렉토리
        while (i + 1 < arg_count && args[i + 1][0] != '-')
        {
```

```
                if (excl_count < 10)
                {
                    strcpy(exclude_paths[excl_count++], args[++i]); // 제외 목록에 추가
                }
            }
        }
        else if (strcmp(args[i], "-e") == 0)
        { // -e 옵션: 허용할 확장자
            while (i + 1 < arg_count && args[i + 1][0] != '-')
            {
                if (ext_count < 10)
                {
                    strcpy(extensions[ext_count++], args[++i]); // 확장자 목록에 추가
                }
            }
        }
        else
        {
            printf("Usage: arrange <DIR_PATH> [OPTION]₩n"); // 잘못된 옵션 입력 시 사용법 출력
            return;
        }
    }

    if (stat(dir_path, &dir_stat) == -1 || !S_ISDIR(dir_stat.st_mode))
    { // 디렉토리 여부 확인
        printf("%s is not a directory₩n", dir_path);
        return;
    }

    scan_directory(dir_path, t_option, time_limit, exclude_paths, excl_count, extensions, ext_count); // 디렉토리 스캔
    process_duplicates();                                                                             // 중복 파일 처리
    mkdir(output_path, 0755);                                                                         // 출력 디렉토리 생성
    organize_files(output_path);                                                                      // 파일 정리
    free_list();                                                                                      // 연결 리스트 메모리 해제

    printf("%s arranged₩n", dir_path); // 정리 완료 메시지 출력
}

// 메인 함수
int main()
{
    char command[7000];          // 사용자 명령어를 저장할 배열
```

```
    int student_id = 20211527; // 학번

    while (1)
    {
        printf("%d> ", student_id);                    // 프롬프트 출력
        fgets(command, sizeof(command), stdin); // 사용자 입력 받기
        command[strcspn(command, "₩n")] = 0;      // 개행 문자 제거

        if (strcmp(command, "exit") == 0) // exit 명령어 입력 시 프로그램 종료
            break;
        else if (strcmp(command, "help") == 0) // help 명령어 입력 시 사용법 출력
            print_help();
        else if (strncmp(command, "tree", 4) == 0) // tree 명령어 입력 시 실행
        {
            execute_tree(command);
        }
        else if (strncmp(command, "arrange", 7) == 0) // arrange 명령어 입력 시 실행
        {
            execute_arrange(command);
        }
        else if (strcmp(command, "") == 0) // 엔터만 입력 시 프롬프트 재출력
            continue;
        else // 잘못된 명령어 입력 시 사용법 출력
        {
            print_help();
        }
    }
    return 0;
}
```