컴퓨터학부 20201662 이정현

목차

1. 과제 개요

리눅스 시스템 상에서 사용자가 상태 관리를 원하는 파일 또는 디렉토리에 대해 경로를 추적하고, 추적된 파일들에 대해 백업 및 복원을 하는 ssu_repo 프로그램을 작성하였다. 추적 경로는 작업 경로 내에 생성된 레포 디렉토리(.repo) 내의 .staging.log 파일을 통해 관리되며, 백업 내역을 담고 있는 커밋 정보는 .commit.log를 통해 관리된다. 해당 프로그램에서 사용 가능한 내장 명령어는 add, remove, status, commit, revert, log, help, exit으로 총 8가지이며, 각각의 내장 명령어는 해당하는 프로세스를 fork() 및 exec()를 통해 실행시키는 방식으로 구현하였다.

2. 기능

ssu_repo 프로그램은 프롬프트를 출력하여 내장 명령어를 입력 받고, 입력 받은 내장 명령어에 해당하는 프로세스를 실행시킴으로써 내장 명령어를 실행한다.

```
// main.c
// 사용자 입력을 기다리는 프롬프트 출력
// exit 입력 시, 프로그램 종료
while(1){
    printf("20201662 > ");
    fgets(input, sizeof(input), stdin);
    input[strlen(input) - 1] = '\0';

    // argList: 공백을 기준으로 자른 토큰들을 가리키는 포인터들를 저장한 리스트
    // argCnt: 토큰의 개수
    // 엔터만 입력한 경우: argList 는 NULL, argCnt 는 0
    if((argList = getArglist(input, &argCnt)) != NULL){
        if(!strcmp(argList[0], "exit")) break;

        // 해당하는 내장 명령어 실행
        for(i = 0; i < NUM_CMD; i++){
            if(!strcmp(argList[0], commandList[i])){
                builtin_command(argList, argCnt);
                break;
            }
```

```
        }

        // 프롬프트에서 지정한 내장명령어 외 기타 명령어를 입력한 경우: help 명령어 실행
        if(i == NUM_CMD)
            builtin_help(NULL);

        // 내장명령어, 경로 등을 저장했던 메모리 해제
        for(i = 0; i < argCnt; i++)
            free(argList[i]);
        free(argList);
    }

    memset(input, 0, sizeof(input));

    printf("\n");
}
```

1) add <PATH>

스테이징 구역에 PATH를 추가하여 해당 경로 내의 파일 및 디렉토리의 변경 내역을 추적하는 명령어이다. 입력 받은 경로(PATH)는 .staing.log에 기록되며, 만약 이미 add된 경로였다면 프롬프트에 "PATH" already added in staging area를 출력하고 해당 경로는 스테이징 구역에 기록하지 않는다.

① initPath(): 현재 홈 디렉토리, 작업 디렉토리, 각 로그파일 등의 경로를 초기화하여 전역 변수에 저장

② initStagingList(): Staging.log 파일을 바탕으로 스테이징 구역 내에서 변경 내역을 추적할 경로들을 연결리스트(trackedPaths)에 저장

③ isNotExistAnyNode(): PATH로 입력 받은 경로가 이미 추적 대상인지 확인하기 위해 연결리스트(trackedPaths)를 탐색

④ 연결리스트(trackedPaths)에 존재하지 않는 경로가 하나라도 있다면 해당 PATH를 추적 경로로 지정하기 위해 .staging.log에 add "PATH" 추가

```
// 홈 디렉토리, 현재 실행 디렉토리 등을 초기화
    initPath();

    // 입력 받은 경로를 절대 경로(fullPath)로 변환하고, 경로에 해당하는 파일/디렉토리명(name)을 저장하고
    // 해당 파일/디렉토리가 올바른 경로인지 확인
    if(processPath(argv[1], fullPath, name) < 0 || checkPath(fullPath, name) < 0){
        fprintf(stderr, "ERROR: %s is wrong path\n", argv[1]);
        exit(1);
    }
```

```
    // 변경사항을 추적할 경로 리스트(trackedPaths)와 추적하지 않을 경로 리스트(untrackedPaths)
초기화
    // 해당 연결리스트는 struct.h 에 전역 변수로 선언되어 있음
    initStagingList();

    // 추적할 경로들의 리스트(trackedPaths) 탐색
    if((result = isNotExistAnyNode(trackedPaths, trackedPaths, fullPath, name)) < 0){
        fprintf(stderr, "ERROR: %s is wrong path\n", argv[1]);
        exit(1);
    }
    // 해당 경로의 정보를 담고있는 노드가 하나라도 존재하지 않는 경우
    else if(result){
        FILE *fp;

        if((fp = fopen(stagingPATH, "a+")) == NULL){
            fprintf(stderr, "fopen error for %s\n", stagingPATH);
            exit(1);
        }
        fprintf(fp, "add \"%s\"\n", fullPath);
        fprintf(stdout, "add \"%s\"\n", argv[1]);

        fclose(fp);
    }
    // 해당 경로의 정보를 담고있는 노드가 모두 존재하는 경우
    else{
        fprintf(stdout, "\"%s\" already exist in staging area\n", argv[1]);
    }
```

2) remove <PATH>

스테이징 구역에 PATH를 추가하여 해당 경로 내의 파일 및 디렉토리의 변경 내역을 추적하지 않는 명령어이다. 입력 받은 경로(PATH)는 .staing.log에 기록되며, 만약 이미 remove된 경로였다면 프롬프트에 “PATH” already removed from staging area를 출력하고 해당 경로는 스테이징 구역에 기록하지 않는다.

① initPath(): 현재 홈 디렉토리, 작업 디렉토리, 각 로그파일 등의 경로를 초기화하여 전역 변수에 저장

② initStagingList(): Staging.log 파일을 바탕으로 스테이징 구역 내에서 변경 내역을 추적하지 않을 경로들을 연결리스트(untrackedPaths)에 저장

③ isNotExistAnyNode(): PATH로 입력 받은 경로가 이미 추적 하지 않는 대상인지 확인하기 위해 연결리스트(untrackedPaths)를 탐색

④ 연결리스트(untrackedPaths)에 존재하지 않는 경로가 하나라도 있다면 해당 PATH를 추적하지 않는 경로로 지정하기 위해 .staging.log에 remove “PATH” 추가

```
// 홈 디렉토리, 현재 실행 디렉토리 등을 초기화
    initPath();

    // 입력 받은 경로를 절대 경로(fullPath)로 변환하고, 경로에 해당하는 파일/디렉토리명(name)을 저장하고
    // 해당 파일/디렉토리가 올바른 경로인지 확인
    if(processPath(argv[1], fullPath, name) < 0 || checkPath(fullPath, name) < 0){
        fprintf(stderr, "ERROR: %s is wrong path\n", argv[1]);
        exit(1);
    }

    // 변경사항을 추적할 경로 리스트(trackedPaths)와 추적하지 않을 경로 리스트(untrackedPaths) 초기화
    // 해당 연결리스트는 struct.h 에 전역 변수로 선언되어 있음
    initStagingList();

    // 추적하지 않을 경로들의 리스트(ubtrackedPaths) 탐색
    if((result = isNotExistAnyNode(untrackedPaths, untrackedPaths, fullPath, name)) < 0){
        fprintf(stderr, "ERROR: %s is wrong path\n", argv[1]);
        exit(1);
    }
    // 해당 경로의 정보를 담고있는 노드가 하나라도 존재하지 않는 경우
    else if(result){
        FILE *fp;

        if((fp = fopen(stagingPATH, "a+")) == NULL){
            fprintf(stderr, "fopen error for %s\n", stagingPATH);
            exit(1);
        }
        fprintf(fp, "remove \"%s\"\n", fullPath);
        fprintf(stdout, "remove \"%s\"\n", argv[1]);

        fclose(fp);
    }
    // 해당 경로의 정보를 담고있는 노드가 모두 존재하는 경우
    else{
        fprintf(stdout, "\"%s\" already removed from staging area\n", argv[1]);
```

```
    }
```

3) status

스테이징 구역에 추가된 경로들에 대하여 변경 내역을 추적하여 정보를 출력하는 명령어이다. add 명령어를 통해 추적 대상으로 지정된 경로는 변경 내역이 추적되어 정보가 출력되고, remove 명령어를 통해 추적하지 않는 대상으로 지정된 경로는 변경 내역을 추적하지 않고 이에 대한 어떠한 출력도 하지 않는다. 또한 스테이징 구역에 존재하지 않는 경로들은, 즉 add 및 remove 모두 된 적이 없는 경로들은 untracked files로 분류되어 출력된다.

① initPath(): 현재 홈 디렉토리, 작업 디렉토리, 각 로그파일 등의 경로를 초기화하여 전역 변수에 저장

② initStagingList(): Staging.log 파일을 바탕으로 스테이징 구역 내에서 변경 내역을 추척할 경로들에 대한 연결리스트(trackedPaths)와 변경 내역을 추적하지 않을 경로들에 대한 연결리스트(untrackedPaths)에 저장하고, 스테이징 구역에 포함되지 않은 경로들에 대한 연결리스트(unknownPaths)도 초기화

③ initCommitList(): commit.log 파일을 바탕으로 백업된 파일들을 연결리스트(commitList)에 저장

④ checkChanges(): 변경 내역을 추적할 경로(trackedPaths)들과 백업 내역(commitList)들을 비교하며, 변경 사항을 연결리스트(changesList)에 저장

⑤ 변경 사항(changeList) 및 스테이징 구역 외의 경로(unknownPaths)들에 대해 출력

```
initPath(); // 홈 디렉토리, 현재 실행 디렉토리 등을 초기화

    initStagingList(); // 스테이징 구역 내의 경로와 구역 외의 경로를 연결리스트에 저장(trackedPaths, untrackedPaths, unknownPaths)
    initCommitList(NULL); // 커밋 구역의 경로를 연결리스트에 저장 (commitList)

    checkChanges(trackedPaths, commitList, &changesList); // 변경 사항을 changesList 에 저장하고, 세부 변경 내용을 각 인자에 저장

    // 스테이징 구역에 포함되어 있는 경로 내 파일들에 대하여 출력
    if(changesList != NULL){
        printf("Changes to be commited:\n");
        printChangesList();
    }

    // 스테이징 구역에 없는 경로 내 파일들에 대하여 출력
    if(unknownPaths != NULL){
        printf("Untracked files: \n");
        printUnknownPaths(unknownPaths);
    }
```

```
// 마지막 백업 파일에서 변경되거나 삭제, 새롭게 추가된 파일이 없는 경우
if(changesList == NULL && unknownPaths == NULL){
    printf("Nothing to commit\n");
}
```

4) Commit <NAME>

커밋 이름인 NAME을 입력 받아 해당 이름으로 버전 디렉토리를 생성하여 추적 대상인 경로(trackedPaths) 내 파일들에 대하여 각자의 마지막 백업 파일에서 변경이 있는 경우에 대해서만 백업을 진행하는 명령어이다.

① initPath(): 현재 홈 디렉토리, 작업 디렉토리, 각 로그파일 등의 경로를 초기화하여 전역 변수에 저장

② processName(): 입력 받을 커밋 이름은 띄어쓰기도 허용하므로, 이름으로 들어온 명령행 인자들을 하나의 이름으로 가공

③ initStagingList(): Staging.log 파일을 바탕으로 스테이징 구역 내에서 변경 내역을 추척할 경로들에 대한 연결리스트(trackedPaths)와 변경 내역을 추적하지 않을 경로들에 대한 연결리스트(untrackedPaths)에 저장하고, 스테이징 구역에 포함되지 않은 경로들에 대한 연결리스트(unknownPaths)도 초기화

④ initCommitList(): commit.log 파일을 바탕으로 백업된 파일들을 연결리스트(commitList)에 저장

⑤ isAvailableCommitName(): commit.log 파일을 확인함으로써 전달 인자인 name으로 커밋된 적이 없는 경우에만 커밋 진행

⑥ checkChanges(): 변경 내역을 추적할 경로(trackedPaths)들과 백업 내역(commitList)들을 비교하며, 변경 사항을 연결리스트(changesList)에 저장

⑦ bakckupChanges(): 변경 사항(changesList)이 있는 파일들을 레포 디렉토리(.repo) 내에 생성한 버전 디렉토리에 저장 및 commit.log 파일에 백업 내역 기록

⑧ printChangesInfo(): 변경 사항이 있는 파일의 개수와 추가 및 삭제된 줄 수 출력

⑨ printChangesList(): 변경 사항이 있는 파일 내역을 프롬프트에 출력

```
// 홈 디렉토리, 현재 실행 디렉토리 등을 초기화
    initPath();

    // 입력 받은 이름은 띄어쓰기도 허용하므로, argv[1]~argv[n]까지 변수를 하나의 이름으로 가공
    if(processName(argc - 1, argv + 1, name) < 0){
        fprintf(stderr, "ERROR: name cannot exceed 255 bytes\n");
        exit(1);
    }

    initStagingList();
    initCommitList(NULL);

    sprintf(path, "%s/%s", repoPATH, name);
```

```
// 입력받은 이름으로 커밋된 적이 없는 경우: 버전 디렉토리 생성
if(isAvailableCommitName(name)){
	if(mkdir(path, 0775) < 0){
		fprintf(stderr, "mkdir error for %s\n", path);
		exit(1);
	}
}
// 입력받은 이름과 동일한 버전 디렉토리가 존재하는 경우: 에러 처리 후, 프롬프트 재출력
else{
	fprintf(stderr, "\"%s\" is already exist in repo\n", name);
	exit(1);
}

// 변경 사항을 changesList 에 저장하고, 세부 변경 내용을 각 인자에 저장
// 변경 사항을 버전 디렉토리에 백업
checkChanges(trackedPaths, commitList, &changesList);
backupChanges(name);

// 변경된 파일이 없는 경우: 해당 버전 디렉토리 삭제
if(changesList == NULL){
	rmdir(path);
	printf("Nothing to commit\n");
}
else{
	struct dirent **namelist;

	printf("commit to \"%s\"\n", name);

	printChangesInfo();
	printChangesList();

	// 변경 내용 중 removed 만 존재하는 경우, 백업디렉토리에는 파일들이 백업되지 않음
	// 즉, 로그 파일에만 커밋 내역 기록되고, 해당 커밋 이름의 디렉토리는 생성되면 안됨
	if(scandir(path, &namelist, NULL, alphasort) == 2)
		rmdir(path);
}
```

5) revert <NAME>

커밋 이름인 NAME을 입력 받아 해당 이름의 버전으로 현재 작업 디렉토리에 파일을 복구

하는 명령어이다. 이 때, 작업 디렉토리에 다른 파일들이 있다면 이는 그대로 유지한다. 또한 동일한 파일이 있는 경우에는 백업 디렉토리 내 파일과 현재 작업 경로 내 파일 간의 변경 사항이 있는 경우에만 복원을 진행한다.

① initPath(): 현재 홈 디렉토리, 작업 디렉토리, 각 로그파일 등의 경로를 초기화하여 전역 변수에 저장

② processName(): 입력 받을 커밋 이름은 띄어쓰기도 허용하므로, 이름으로 들어온 명령행 인자들을 하나의 이름으로 가공

③ initStagingList(): Staging.log 파일을 바탕으로 스테이징 구역 내에서 변경 내역을 추척할 경로들에 대한 연결리스트(trackedPaths)와 변경 내역을 추적하지 않을 경로들에 대한 연결리스트(untrackedPaths)에 저장하고, 스테이징 구역에 포함되지 않은 경로들에 대한 연결리스트(unknownPaths)도 초기화

④ initCommitList(): commit.log 파일을 바탕으로 전달 인자인 name을 이름으로 하는 버전 디렉토리까지 백업 상태를 연결리스트(commitList)에 저장

⑤ checkRecover(): 변경 내역을 추적할 경로(trackedPaths)들 중 백업 내역(commitList)에 있는 파일들에 대하여 현재 작업 경로 내 파일들을 비교하며 복원 대상에 해당하는 파일을 현재 작업 경로에 복원하고, 복원 내역을 연결리스트(recoverList)에 저장

⑥ printRecoverList(): 복원한 파일 내역을 프롬프트에 출력

```
// 홈 디렉토리, 현재 실행 디렉토리 등을 초기화
    initPath();

    // 입력 받은 이름은 띄어쓰기도 허용하므로, argv[1]~argv[n]까지 변수를 하나의 이름으로 가공
    if(processName(argc - 1, argv + 1, name) < 0){
        fprintf(stderr, "ERROR: name cannot exceed 255 bytes\n");
        exit(1);
    }

    initStagingList();
    initCommitList(name); // name 버전 디렉토리를 포함한 백업 상태 연결 리스트 생성

    // name 버전 디렉토리로 복원
    // 복원한 파일은 recoverList 연결리스트에 저장
    checkRecover(commitList, &recoverList);
    printf("0\n");

    if(recoverList == NULL){
        printf("nothing changed with \"%s\"\n", name);
    }
    else{
        printf("revert to \"%s\"\n", name);
```

```
        printRecoverList();
    }
```

6) log [NAME]

커밋 기록에 대해 로그를 출력하는 명령어이다. 명령행 인자가 없는 경우에는 commit.log 파일에 기록된 모든 커밋 로그를 출력하고, 인자 NAME이 입력된 경우에는 NAME을 이름으로 하는 버전 디렉토리의 커밋 로그를 출력한다. 만약 입력 받은 인자를 이름으로 커밋한 이력이 없다면 에러 처리를 한다.

① Commit.log 파일을 한 줄씩 읽어와 커밋 로그 출력

```
if(sscanf(buf, "commit: \"%[^\"]\" - new file: \"%[^\"]\"", commitName, path)
== 2)
                    strcpy(action, "new file");
                else if(sscanf(buf, "commit: \"%[^\"]\" - modified: \"%[^\"]\"", commitName, path) == 2)
                    strcpy(action, "modified");
                else if(sscanf(buf, "commit: \"%[^\"]\" - removed: \"%[^\"]\"", commitName, path) == 2)
                    strcpy(action, "removed");

                else{
                    fprintf(stderr, "ERROR: \"commit.log\" file contains invalid values\n");
                    exit(1);
                }
```

7) help [COMMAND]

ssu_repo 프로그램의 내장 명령어에 대한 설명을 출력하는 명령어이다. 명령행 인자가 없는 경우에는 모든 내장 명령어에 대한 설명을 출력하고, 인자 COMMAND가 입력된 경우에는 해당하는 내장 명령어의 설명을 출력한다. 만약 입력 받은 인자에 해당하는 내장 명령어가 존재하지 않는 경우에는 에러 처리를 한다.

```
// 모든 내장 명령어에 대한 설명을 출력하는 경우
    if(argc == 1){
        printf("Usage: \n");
        printf("      > %s\n", USAGE_ADD);
        printf("      > %s\n", USAGE_REMOVE);
        printf("      > %s\n", USAGE_STATUS);
        printf("      > %s\n", USAGE_COMMIT);
        printf("      > %s\n", USAGE_REVERT);
        printf("      > %s\n", USAGE_LOG);
        printf("      > %s\n", USAGE_HELP);
        printf("      > %s\n", USAGE_EXIT);
    }
    // 특정 명령어에 대한 설명을 출력하는 경우
```

```
else{
        if(!strcmp(argv[1], "add")) printf("Usage: %s\n", USAGE_ADD);
        else if(!strcmp(argv[1], "remove")) printf("Usage: %s\n", USAGE_REMOVE);
        else if(!strcmp(argv[1], "status")) printf("Usage: %s\n", USAGE_STATUS);
        else if(!strcmp(argv[1], "commit")) printf("Usage: %s\n", USAGE_COMMIT);
        else if(!strcmp(argv[1], "revert")) printf("Usage: %s\n", USAGE_REVERT);
        else if(!strcmp(argv[1], "log")) printf("Usage: %s\n", USAGE_LOG);
        else if(!strcmp(argv[1], "help")) printf("Usage: %s\n", USAGE_HELP);
        else if(!strcmp(argv[1], "exit")) printf("Usage: %s\n", USAGE_EXIT);
        else printf("ERROR: help [add | status | commit | revert | log | help | exit]\n");
}
```

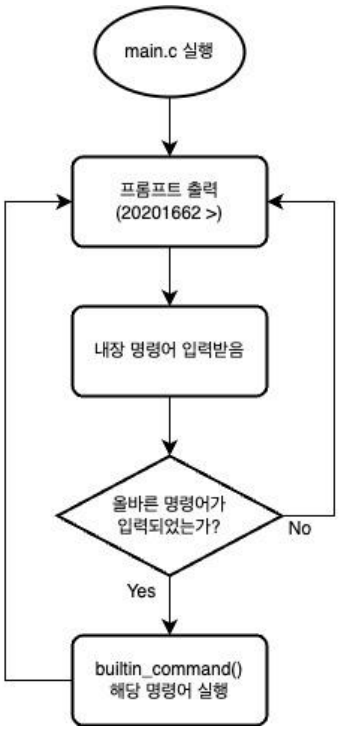
8) exit

현재 실행 중인 ssu_repo 프로그램을 종료하는 명령어이다.

3. 상세 설계

명령어	add	remove	status	commit	revert	log	help	exit
구현	○	○	○	○	○	○	○	○

main.c에서 프롬프트를 출력하고 내장 명령어와 명령어에 따른 인자를 입력 받는다. 입력 받은 내장 명령어가 존재하는 내장 명령어인지 확인하고, builtin_commnad()를 통해 각 내장 명령어들을 실행시킬 프로세스를 생성하고 이를 실행한다.



builtin_commnad()의 코드는 다음과 같으며, command.c 파일에서 확인할 수 있다. 즉, 모든 내장 명령어는 아래의 코드를 통해 실행된다.

```
// 내장 명령어를 실행하는 함수
void builtin_command(char **argList, int argc)
{
	char **argv = (char **)malloc(sizeof(char *) * (argc + 1));
	int i;

	// execv()를 실행할 때, 실행할 프로그램에 전달할 인자들을 나타내는 배열의 마지막 원소는 NULL 포인터여야함
	for(i = 0; i < argc; i++)
		argv[i] = argList[i];
	argv[i] = NULL;

	// help 명령어인 경우: help 다음에 온 인자 또는 NULL 전달
	if(!strcmp(argv[0], "help"))
		builtin_help(argv[1]);
	// 그 외의 다른 명령어인 경우: 해당 명령어 프로세스 실행
	else{
		pid_t pid;

		// 내장 명령어 실행 -> 뒤의 인자만 전달 (명령어 자체는 전달하지 않음)
		if((pid = fork()) < 0){
			fprintf(stderr, "fork error for %s\n", argv[0]);
			exit(1);
		}
		else if(pid == 0){
			execv(argv[0], argv);
			exit(0);
		}
		else{
			wait(NULL);
		}

	}

	free(argv);
}
```

1) 구조체와 연결리스트

ssu_repo 프로그램을 구현하기 위해 연결리스트를 통해 각 파일들의 상태를 관리하였다. 연결리스트의 노드를 위한 구조체와 연결리스트의 선언 및 연결리스트 관련 함수들은 모두 struct.h와 struct.c에서 확인할 수 있다. 해당 프로그램에서 사용된 연결리스트는 다음과 같다.

- DirNode *trackedPaths
  - ➓ 스테이징 구역 내의 경로 중 추적 대상이 되는 경로들의 정보를 저장한 연결리스트
  - ➓ add 명령어를 통해 staging.log에 기록된 경로들이 대상이 됨
  - ➓ 해당 연결리스트는 initStagingList()에서 초기화 진행
- DirNode *untrackedPaths
  - ➓ 스테이징 구역 내의 경로 중 추적 하지 않을 경로들의 정보를 저장한 연결리스트
  - ➓ remove 명령어를 통해 staging.log에 기록된 경로들이 대상이 됨
  - ➓ 해당 연결리스트는 initStagingList()에서 초기화 진행
- DirNode *unknownPaths
  - ➓ 스테이징 구역에 포함되지 않은 경로들의 정보를 저장한 연결리스트
  - ➓ add 및 remove 명령어 모두 실행되지 않은 경로들이 대상이 됨
  - ➓ 해당 연결리스트는 initStagingList()에서 초기화 진행
- DirNode *commitList
  - ➓ 레포 디렉토리에 백업된 파일들의 경로들의 정보를 저장한 연결리스트
  - ➓ commit 명령어를 통해 commit.log에 기록된 경로들이 대상이 됨
  - ➓ 해당 연결리스트는 initCommitList()에서 초기화 진행
- FileNode *recoverList
  - ➓ 레포 디렉토리에서 원본 경로로 복원을 진행할 파일들의 정보를 저장한 연결리스트
  - ➓ 해당 연결리스트는 checkRecover()에서 초기화 진행
- ChangeNode *changeList
  - ➓ 추적하고 있는 경로들 중 백업 파일과 비교했을 때 변경 사항이 있는 파일들의 정보를 저장한 연결리스트
  - ➓ 해당 연결 리스트는 checkChange()에서 초기화 진행

각 연결리스트들의 노드를 구성하고 있는 구조체는 다음과 같다.

➓ Struct DirNode: 디렉토리의 이름, 경로, 상위/하위/형제 디렉토리 노드, 하위 파일 등

```
// 디렉토리 노드
typedef struct DirNode{
    char name[MAX_NAME + 1]; // 해당 디렉토리의 이름
    char path[MAX_PATH + 1]; // 해당 디렉토리의 원본 경로

    int isTracked; // 해당 디렉토리 자체가 추적 대상인지 확인하는 플래그 -> staging.log 관련한 변수

    struct DirNode *parentDir; // 상위 디렉토리
    struct DirNode *subDir; // 하위 디렉토리
    struct FileNode *fileHead; // 하위 파일들의 리스트
```

```
    // 형제 디렉토리
    struct DirNode *next;
    struct DirNode *prev;
}DirNode;
```

❿ Struct FileNode: 파일의 이름, 경로, 상위 디렉토리 노드, 형제 파일 노드 등

```
// 파일 노드
typedef struct FileNode{
    char name[MAX_NAME + 1]; // 해당 파일의 이름
    char path[MAX_PATH + 1]; // 해당 파일의 경로
    char commit[MAX_NAME + 1]; // 버전 디렉토리 이름 -> commit.log 관련한 변수

    // 해당 파일의 상위 디렉토리 노드
    struct DirNode *parentDir;

    // 형제 파일
    struct FileNode *prev;
    struct FileNode *next;
}FileNode;
```

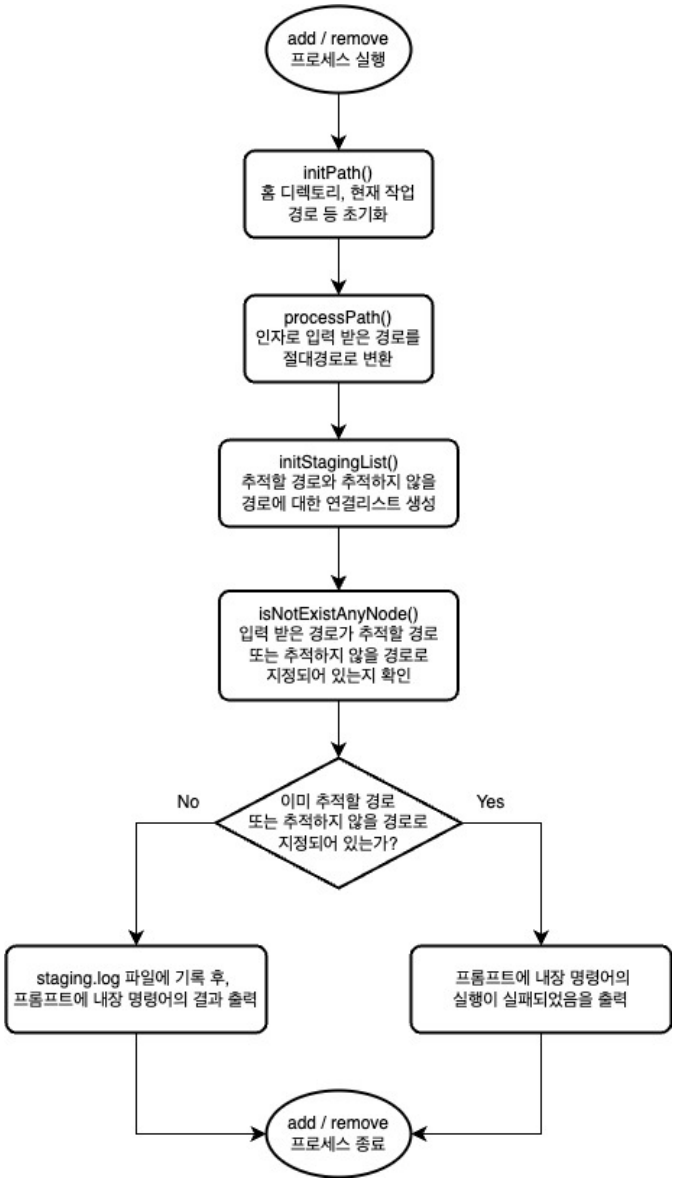
❿ Struct ChangeNode: 파일의 경로, 변경 사항, 추가/삭제된 줄 수 등

```
// 파일의 변경 내역을 저장하는 노드
typedef struct ChangeNode{
    char path[MAX_PATH + 1];
    char action[MAX_NAME + 1];

    int insertions;
    int deletions;

    struct ChangeNode *prev;
    struct ChangeNode *next;
}ChangeNode;
```

2) 내장 명령어: add, remove

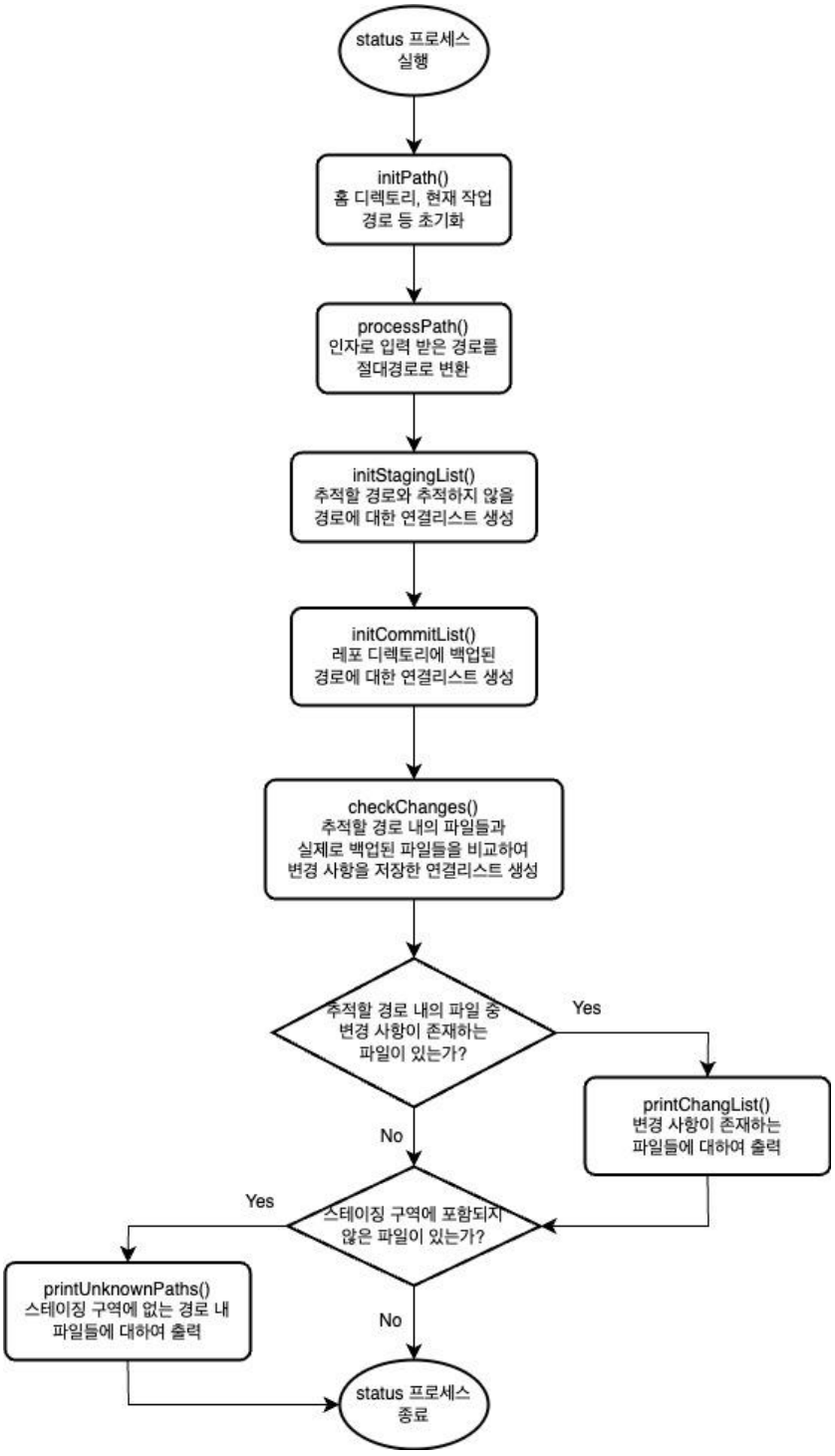
스테이징 구역의 경로는 add 또는 remove 명령어를 통해 추적 대상이거나 추적하지 않을 대상을 의미한다. 즉, 스테이징 구역 내의 경로는 추적 대상과 추적하지 않을 대상으로 나뉜다. add <PATH>명령어가 실행되었다면 PATH는 추적할 대상이 되며, 만약 이전에 remove를 통해 추적하지 않을 대상으로 지정한 적이 있다면, 해당 경로는 추적하지 않을 대상에서 제외된다. 동일하게 remove <PATH> 명령어가 실행되었다면 PATH는 추적하지 않을 대상이 되며, 만약 이전에 add를 통해 추적하지 않을 대상으로 지정한 적이 있다면, 해당 경로는 추적할 대상에서 제외된

다.



3) 내장 명령어: status

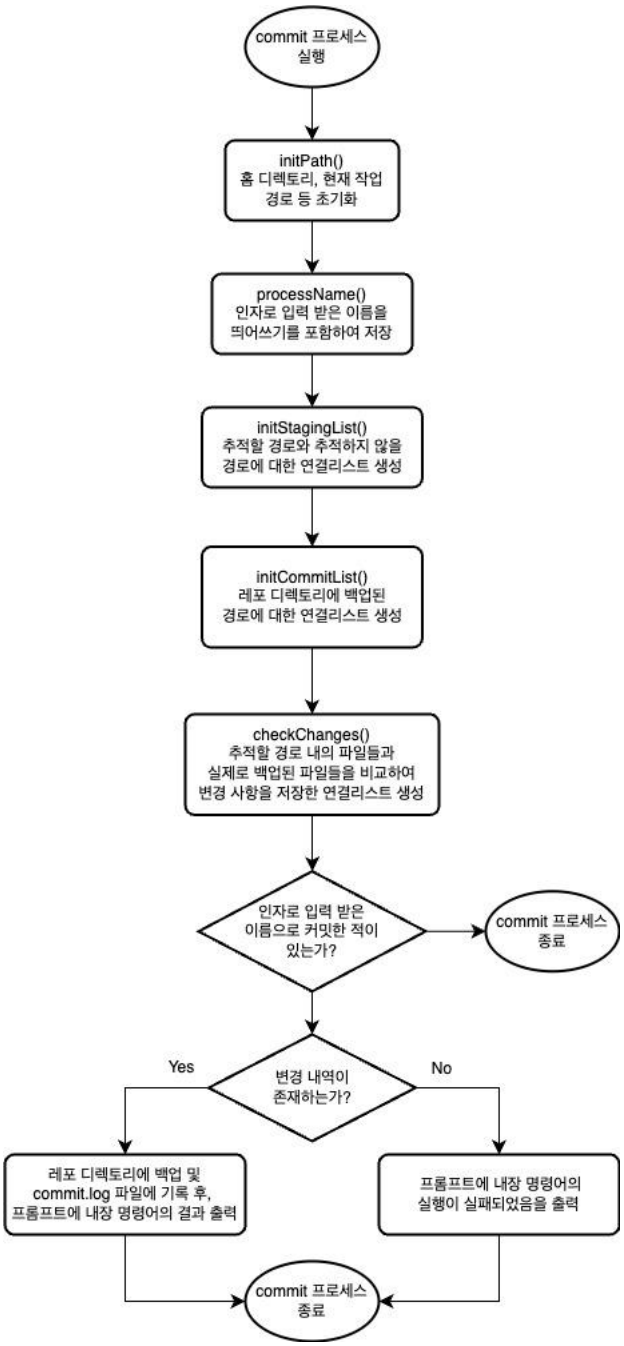
현재 작업 경로 내에서 추적할 대상에 포함되는 경로에 대하여 변경 내역을 추적하여 프롬프트에 출력한다. 이 때 스테이징 구역에 포함되지 않는 경로에 대해서는 추적하지 않는 경로라고 출력한다. 즉, remove 명령어를 통해 추적하지 않을 대상으로 지정된 경로들을 제외한 나머지 경로들에 대해 변경 내역을 추적하여 프롬프트에 출력한다.



4) 내장 명령어: commit

Commit.log 파일에 기록된 커밋 내역을 통해 레포 디렉토리에 백업된 파일들에 대하여 각각의 파일들의 가장 최근 버전을 연결리스트로 관리한다. 또한 백업 파일들의 최신 버전과 추적 대상 경로 내 파일들을 비교하여 변경 사항을 관리한다. commit <NAME> 명령어 입력 시에 레포 디렉토리 내부에 NAME에 해당하는 버전 디렉토리를 생성하여 변경 사항이 존재하는 파일들을 백업하고 commit.log에 파일의 경로와 변경 내역을 기록한다. 이 때, 삭제된 파일은 commit.log 파일

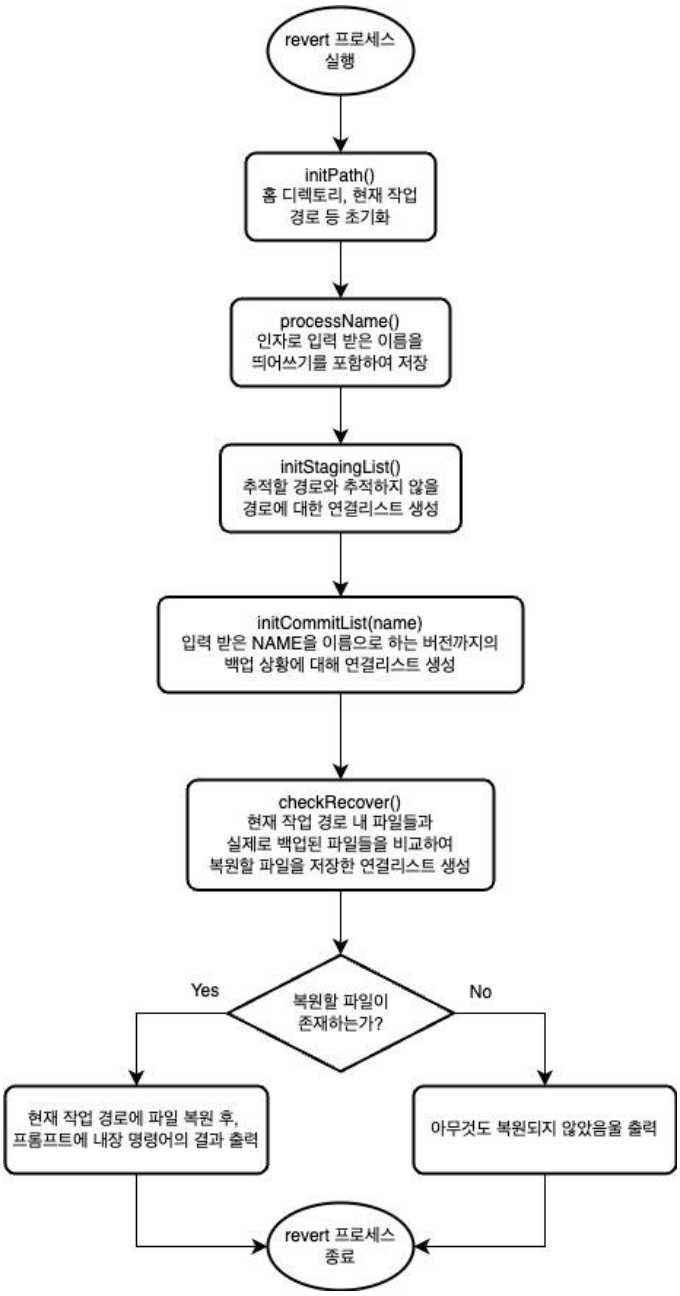
에는 기록되지만 레포 디렉토리에는 백업되지 않는다. 만약 NAME에 해당하는 커밋 내역이 이미 존재한다면 commit 명령어는 실행되지 않는다. 추적 대상이 되는 경로 내 파일 모두 변경 내역이 존재하지 않는다면, 버전 디렉토리는 생성되지 않으며 commit.log 파일에도 아무것도 기록되지 않는다.



5) 내장 명령어: revert

이전에 작업 경로 내의 파일들을 레포 디렉토리에 백업(커밋)한 버전 디렉토리들에 대하여 특정 버전의 파일들을 작업 경로에 복원한다. 즉, revert <NAME> 실행 시 레포 디렉토리 내에서 NAME에 해당하는 버전 디렉토리를 찾아 해당 디렉토리에 저장된 파일들을 작업 경로에 복원한다. 만약 복원할 파일과 동일한 이름의 파일이 작업 경로에 존재한다면, 두 파일의 내용이 다를 때에만 복원을 진행한다. 또한 복원할 파일이 아닌 파일이 작업 경로에 존재했다면, 이는 그대

로 둔다. 또한 복원된 백업 파일 또한 삭제되지 않는다.



4. 실행 결과

1) add

❿ add 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우)

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > add
ERROR: <PATH> is not include
Usage: add <PATH> : add files/directories to staging area

20201662 > add asd.txt
ERROR: asd.txt is wrong path

20201662 > add a.txt
add "./a.txt"

20201662 > add a.txt
"./a.txt" already exist in staging area

20201662 > add ./a
add "./a"

20201662 > add /home/ubuntu/lsp/a/b
"./a/b" already exist in staging area

20201662 > add ./
add "."

20201662 > remove a
remove "./a"

20201662 > add /home/ubuntu/lsp/a/a.txt
add "./a/a.txt"

20201662 > add ./a
add "./a"
```

❿ 현재 작업 디렉토리 트리 구조

```
ubuntu@ip-192-168-0-101:~/lsp$ tree
.
├── a
│   ├── a.txt
│   ├── b
│   │   └── c.txt
│   └── c.txt
├── a.txt
├── add
├── b
├── commit
├── help
├── log
├── remove
├── revert
├── ssu_repo
└── status
```

❿ 스테이징 구역(.staging.log) 파일 내용

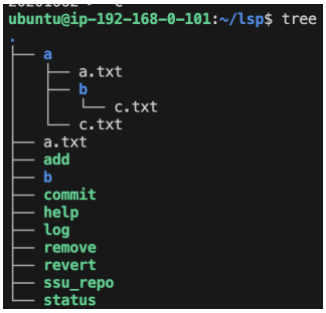
```
ubuntu > lsp > .repo > ≡ .staging.log
1   add "/home/ubuntu/lsp/a.txt"
2   add "/home/ubuntu/lsp/a"
3   add "/home/ubuntu/lsp"
4   remove "/home/ubuntu/lsp/a"
5   add "/home/ubuntu/lsp/a/a.txt"
6   add "/home/ubuntu/lsp/a"
7
```

2) remove

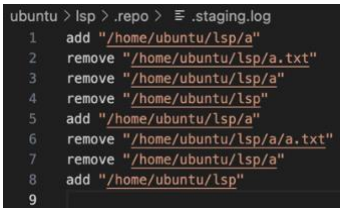
❿ remove 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우)

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > add a
add "./a"

20201662 > remove
ERROR: <PATH> is not include
Usage: remove <PATH> : remove files/directories from staging area

20201662 > remove asd.txt
ERROR: asd.txt is wrong path

20201662 > remove a.txt
remove "./a.txt"

20201662 > remove a.txt
"./a.txt" already removed from staging area

20201662 > remove ./a
remove "./a"

20201662 > remove /home/ubuntu/lsp/a/b
"./a/b" already removed from staging area

20201662 > remove ./
remove "."

20201662 > add a
add "./a"

20201662 > remove /home/ubuntu/lsp/a/a.txt
remove "./a/a.txt"

20201662 > remove ./a
remove "./a"
```

❿ 현재 작업 디렉토리 트리 구조

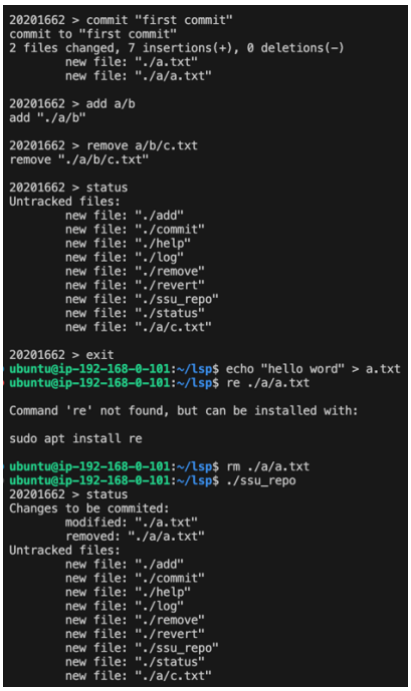


❿ 스테이징 구역(.staging.log) 파일 내용

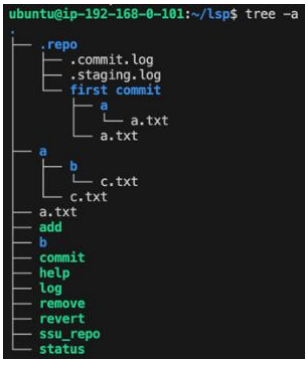


3) status

❿ status 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우, a.txt와 a/a.txt는 직전에 “first commit”이라는 이름으로 백업되어 있음)



❿ 현재 작업 디렉토리 트리 구조



❿ 스테이징 구역(.staging.log) 파일 내용

```
ubuntu > lsp > .repo >  .staging.log
1   add "/home/ubuntu/lsp/a.txt"
2   add "/home/ubuntu/lsp/a/a.txt"
3   add "/home/ubuntu/lsp/a/b"
4   remove "/home/ubuntu/lsp/a/b/c.txt"
5
```

❿ 커밋 로그(.commit.log) 파일 내용

```
ubuntu > lsp > .repo >  .commit.log
1   commit: "first commit" - new file: "/home/ubuntu/lsp/a.txt"
2   commit: "first commit" - new file: "/home/ubuntu/lsp/a/a.txt"
3
```

4) commit

❿ commit 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우)

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > status
Changes to be commited:
        modified: "./a.txt"
        removed: "./a/a.txt"
Untracked files:
        new file: "./add"
        new file: "./commit"
        new file: "./help"
        new file: "./log"
        new file: "./remove"
        new file: "./revert"
        new file: "./ssu_repo"
        new file: "./status"
        new file: "./a/c.txt"

20201662 > add ./a/b/c.txt
add "./a/b/c.txt"

20201662 > commit
ERROR: <PATH> is not include
Usage: commit <name> : backup staging area with commit name

20201662 > commit "second commit"
commit to "second commit"
3 files changed, 1 insertions(+), 6 deletions(-)
        modified: "./a.txt"
        removed: "./a/a.txt"
        new file: "./a/b/c.txt"

20201662 > commit "first commit"
"first commit" is already exist in repo

20201662 > commit "third "commit"
Nothing to commit
```

❿ 스테이징 구역(.staging.log) 파일 내용

```
ubuntu > lsp > .repo >  .staging.log
1   add "/home/ubuntu/lsp/a.txt"
2   add "/home/ubuntu/lsp/a/a.txt"
3   add "/home/ubuntu/lsp/a/b"
4   remove "/home/ubuntu/lsp/a/b/c.txt"
5   add "/home/ubuntu/lsp/a/b/c.txt"
6
```

❿ commit 이후 현재 작업 디렉토리 트리 구조

```
ubuntu@ip-192-168-0-101:~/lsp$ tree -a
.
├── .repo
│   ├── .commit.log
│   ├── .staging.log
│   ├── first commit
│   │   ├── a
│   │   │   └── a.txt
│   │   └── a.txt
│   └── second commit
│       ├── a
│       │   └── b
│       │       └── c.txt
│       └── a.txt
├── a
│   ├── b
│   │   └── c.txt
│   └── c.txt
├── a.txt
├── add
├── b
├── commit
├── help
├── log
├── remove
├── revert
├── ssu_repo
└── status
```

❿ 커밋 로그(.commit.log) 파일 내용

```
ubuntu > lsp > .repo >  ≡ .commit.log
1   commit: "first commit" - new file: "/home/ubuntu/lsp/a.txt"
2   commit: "first commit" - new file: "/home/ubuntu/lsp/a/a.txt"
3   commit: "second commit" - modified: "/home/ubuntu/lsp/a.txt"
4   commit: "second commit" - removed: "/home/ubuntu/lsp/a/a.txt"
5   commit: "second commit" - new file: "/home/ubuntu/lsp/a/b/c.txt"
6
```

5) revert

❿ revert 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우)

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > revert
ERROR: <PATH> is not include
Usage: commit <name> : backup staging area with commit name

20201662 > revert "forth commit"
revert to "forth commit"
recover "/home/ubuntu/lsp/a/b/c.txt" from "third commit"

20201662 > revert "second commit"
revert to "second commit"
recover "/home/ubuntu/lsp/a.txt" from "second commit"
recover "/home/ubuntu/lsp/a/b/c.txt" from "second commit"

20201662 > revert "second commit"
nothing changed with "second commit"

20201662 > revert "tmp commit"
"tmp commit" is not exist in repo
```

❿ 현재 작업 디렉토리 트리 구조

```
ubuntu@ip-192-168-0-101:~/lsp$ tree -a
.
├── .repo
│   ├── .commit.log
│   ├── .staging.log
│   ├── first commit
│   │   ├── a
│   │   │   └── a.txt
│   │   └── a.txt
│   ├── second commit
│   │   ├── a
│   │   │   └── b
│   │   │       └── c.txt
│   │   └── a.txt
│   └── third commit
│       └── a
│           └── b
│               └── c.txt
├── add
├── b
├── commit
├── help
├── log
├── remove
├── revert
├── ssu_repo
└── status
```

❿ 커밋 로그(.commit.log) 파일 내용

```
ubuntu > lsp > .repo >  ≡ .commit.log
1   commit: "first commit" - new file: "/home/ubuntu/lsp/a.txt"
2   commit: "first commit" - new file: "/home/ubuntu/lsp/a/a.txt"
3   commit: "second commit" - modified: "/home/ubuntu/lsp/a.txt"
4   commit: "second commit" - removed: "/home/ubuntu/lsp/a/a.txt"
5   commit: "second commit" - new file: "/home/ubuntu/lsp/a/b/c.txt"
6   commit: "third commit" - modified: "/home/ubuntu/lsp/a/b/c.txt"
7   commit: "forth commit" - removed: "/home/ubuntu/lsp/a.txt"
8
```

❿ revert 이후 현재 작업 디렉토리 트리 구조

```
ubuntu@ip-192-168-0-101:~/lsp$ tree -a
.
├── .repo
│   ├── .commit.log
│   ├── .staging.log
│   ├── first commit
│   │   ├── a
│   │   │   └── a.txt
│   │   └── a.txt
│   ├── second commit
│   │   ├── a
│   │   │   └── b
│   │   │       └── c.txt
│   │   └── a.txt
│   └── third commit
│       └── a
│           └── b
│               └── c.txt
├── a
│   └── b
│       └── c.txt
├── a.txt
├── add
├── b
├── commit
├── help
├── log
├── remove
├── revert
├── ssu_repo
└── status
```

6) log

❿ revert 내장 명령어 실행 (현재 작업 디렉토리(pwd)가 “home/ubuntu/lsp”일 경우)

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > log
commit: "first commit"
 - new file: "/home/ubuntu/lsp/a.txt"
 - new file: "/home/ubuntu/lsp/a/a.txt"
commit: "second commit"
 - modified: "/home/ubuntu/lsp/a.txt"
 - removed: "/home/ubuntu/lsp/a/a.txt"
 - new file: "/home/ubuntu/lsp/a/b/c.txt"
commit: "third commit"
 - modified: "/home/ubuntu/lsp/a/b/c.txt"
commit: "forth commit"
 - removed: "/home/ubuntu/lsp/a.txt"

20201662 > log "second commit"
commit: "second commit"
 - modified: "/home/ubuntu/lsp/a.txt"
 - removed: "/home/ubuntu/lsp/a/a.txt"
 - new file: "/home/ubuntu/lsp/a/b/c.txt"
```

❿ 현재 작업 디렉토리 트리 구조

```
ubuntu@ip-192-168-0-101:~/lsp$ tree -a
.
├── .repo
│   ├── .commit.log
│   ├── .staging.log
│   ├── first commit
│   │   ├── a
│   │   │   └── a.txt
│   │   └── a.txt
│   ├── second commit
│   │   ├── a
│   │   │   └── b
│   │   │       └── c.txt
│   │   └── a.txt
│   └── third commit
│       └── a
│           └── b
│               └── c.txt
├── add
├── b
├── commit
├── help
├── log
├── remove
├── revert
├── ssu_repo
└── status
```

❿ 커밋 로그(.commit.log) 파일 내용

```
ubuntu > lsp > .repo > ≡ .commit.log
1   commit: "first commit" - new file: "/home/ubuntu/lsp/a.txt"
2   commit: "first commit" - new file: "/home/ubuntu/lsp/a/a.txt"
3   commit: "second commit" - modified: "/home/ubuntu/lsp/a.txt"
4   commit: "second commit" - removed: "/home/ubuntu/lsp/a/a.txt"
5   commit: "second commit" - new file: "/home/ubuntu/lsp/a/b/c.txt"
6   commit: "third commit" - modified: "/home/ubuntu/lsp/a/b/c.txt"
7   commit: "forth commit" - removed: "/home/ubuntu/lsp/a.txt"
8
```

7) help

❿ help 내장 명령어 실행

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > help
Usage:
    > add <PATH> : add files/directories to staging area
    > remove <PATH> : remove files/directories from staging area
    > status : show staging area status
    > commit <name> : backup staging area with commit name
    > revert <name> : revert commit directory with commit name
    > log : show commands for log
    > help : show commands for program
    > exit : exit program

20201662 > help add
Usage: add <PATH> : add files/directories to staging area

20201662 > help log
Usage: log : show commands for log

20201662 > help status
Usage: status : show staging area status
```

8) exit

❿ exit 내장 명령어 실행

```
ubuntu@ip-192-168-0-101:~/lsp$ ./ssu_repo
20201662 > exit
ubuntu@ip-192-168-0-101:~/lsp$
```