컴퓨터프로그래밍3 01분반
Term Project

컴퓨터융합학부
202002502 배민수

제출일 2023-06-19(월)

1. 프로그램 설계

< 프로그램의 목적 >

- 사용자로부터 명령어(-a, -i, -r)와 인자(index, string)를 입력받아, 텍스트 파일에 문자열을 추가, 삽입, 삭제를 수행할 수 있는 Line Editor 프로그램

터미널에서 해당 파일을 실행하면서 파일명을 같이 받아오고, 명령어와 인자들은 프로그램 내부 코드에서 while문을 통해 반복적으로 받아낸다. exit이 입력으로 들어오면 프로그램을 종료한다.

< Page 구조체 >

- Page 구조체를 정의하여 활용한다. 해당 구조체는 갖고 있는 원소의 개수 변수, 문자열을 10개까지 담을 수 있는 배열(최대 80자로 문자열이 주어진다는 조건이 주어짐), 다음 페이지와 이전 페이지의 포인터를 갖고 있다.

< appendLine 함수 >

- 모든 원소 중 가장 마지막 원소의 그 바로 다음 자리에 원소를 삽입한다.

재귀 구조를 활용하여 구현하였다.
만약 현재 구조체부터 마지막 구조체까지의 모든 원소의 개수가 현재 구조체의 원소 개수보다 더 많다면, 뒤쪽 구조체에 원소가 더 존재한다는 뜻이다.

즉, 다음 구조체로 넘어가서 appendLine을 수행해야하므로 다음 구조체로부터 마지막 구조체까지의 모든 원소의 개수와, 다음 페이지 포인터를

인자로 넘겨 재귀 호출하여 기능을 수행한다.

현재 구조체로부터 마지막 구조체까지의 모든 원소의 개수와 현재 구조체의 원소 개수와 같아지더라도, 현재 구조체에 원소가 꽉 차 있으면, 이 경우에도 다음 페이지로 넘어가서 원소 삽입을 해줘야한다. 이때도 재귀를 호출한다.

< insertLine 메소드 >
사용자로부터 받은 index와 문자열을 활용하여 특정 위치에 적절히 삽입한다.

만약 index가 현재 구조체 원소 개수보다 큰 경우, 삽입 위치가 뒷쪽 페이지에 존재한다는 뜻이다.

이 경우, 재귀를 활용하여 다음 페이지로 넘어가 기능을 수행한다.

삽입 위치가 현재 구조체에 있는 경우더라도, 현재 구조체가 꽉 차 있다면, 바로 다음 위치에 페이지를 새로 생성한 후(다음 페이지가 이미 존재했다면, 그 페이지와 현재 페이지의 사이에 하나를 새로 생성), 현재 구조체의 마지막 원소를 그 페이지의 첫 번째 자리에 삽입한다.

이 후에는 삽입 위치로부터 끝까지 원소를 한 칸씩 뒤로 밀고, 삽입 위치에 원소를 삽입해주는 방식으로 설계하였다.

< removeLine 메소드 >
사용자로부터 제거 위치를 입력받아 삭제를 수행한다.

만약 제거 위치가 현재 구조체 개수보다 큰 경우, 제거 위치가 뒷쪽 페이지에 존재한다는 뜻이므로, 다음 페이지로 넘어가는 재귀를 호출한다.

제거 위치가 현재 구조체에 있고, 그 위치가 페이지의 마지막 라인인 경우에는 삭제 후 당겨줄 뒷 원소가 아예 없으므로, 삭제 위치의 원소를 삭제 후(빈 문

자열 덧씌우기), 바로 리턴하여 종료한다.

만약 그렇지 않다면, 삭제 위치 이후의 원소들을 한 칸씩 앞으로 당겨주고, 마지막 원소가 당기기 전에 원래 있던 위치를 빈 문자열로 초기화해주는 식으로 설계했다.

< 기존 텍스트 파일 로드 >
기존에 있던 텍스트 파일의 내용을 불러와서 구조체에 적절히 삽입한 후, 명령어들을 수행하도록 설계하였다.

삽입할 때, 최적화를 위해 각 구조체에 최대 5개까지만 삽입되도록 설계하였다. (최초 로드할 때 삽입 시에만)

< 예외 처리 >
1. 명령어 종류가 잘못 입력된 경우
2. 인자로 빈 문자열이 입력된 경우
3. index 값이 유효하지 않은 경우

해당 예외들에 대해, appendLine, insertLine, removeLine을 수행하기 이전에 조건문을 통해 미리 예외 경우를 걸러내어 경고 메시지를 출력해주는 식으로 예외 처리를 설계하였다.

< 디버깅, 구조 확인, 파일 쓰기 >
디버깅 및 구조 확인을 쉽게 하기 위해, 매번 명령어를 실행할 때마다 모든 구조체의 원소 정보를 출력하도록 설계하였다.

exit이 들어오고나서는, 구조체의 모든 원소를 텍스트 파일에 write하고, 구조체 원소 개수 정보를 간략히 출력해준 뒤, 생성해줬던 page 동적 메모리들을 free해주고 프로그램을 종료하는 식으로 설계했다.

2. 구체적인 동작 원리 (코드 설명)

상세한 동작 원리를 코드 이미지와 함께 설명하겠다.

2-1. 구조체 및 함수 설명

< Page 구조체 >

```
typedef struct Page{
    int count; // 저장된 문자열 개수
    char lines[10][100]; // 문자열 10개 저장 가능 (문자열은
    최대 80자 입력됨)
    struct Page * nextPage; // 다음 페이지 구조체 포인터
    struct Page * prevPage; // 이전 페이지 구조체 포인터
} Page;
```

문자열을 담는 페이지에 해당하는 구조체이다.

멤버는 다음과 같다.

count : 현재 저장되어 있는 문자열의 개수
lines : 최대 10개의 문자열을 저장할 수 있는 배열 (각 문자열은 최대 80자로 들어온다고 문제에서 조건이 주어짐)
nextPage : 다음 페이지의 포인터
prevPage : 이전 페이지의 포인터

양 옆의 페이지 포인터를 가짐으로써 이중 연결 리스트의 기능을 수행한다.

< newPage 함수 >

```
// page 구조체 동적 메모리 생성
Page * newPage(Page * prev) {
    Page * newPage = (Page *)malloc(sizeof(Page));
    newPage->count = 0;
    newPage->nextPage = NULL;
    newPage->prevPage = prev;
    return newPage;
}
```

새로운 Page 구조체를 동적 메모리 할당하는 함수이다.

appendLine, insertLine에서 newPage를 실행하는 구문이 있는데, 함수 내에서 생성한 Page 구조체를, 함수가 리턴된 후에도 유지하기 위해 동적 메모리 할당으로 newPage를 구현했다.

함수 내에서 구조체를 새로 할당하고, 이중 연결 리스트에 맞게 nextPage와 prevPage를 적절히 설정해주었다.

< appendLine 함수 >

사용자로부터 -a 명령어와 문자열을 인자로 입력받았을 때, 해당 문자열을 가장 마지막 원소의 한 칸 뒤의 위치에 삽입하는 함수이다.

main 함수에서 최초 호출 시 Page 포인터 인자로 mainPage를 받아 시작한다.

함수의 수행 내용은 다음과 같다.

만약 "인자로 받은 페이지로부터 마지막 페이지까지의 원소 개수의 총합"이, "인자로 받은 페이지 내 원소 개수"보다 크다면, append 해야할 위치가 뒷쪽 페이지에 존재한다는 의미이다.

이 때는 다음 페이지를 인자로 넘겨 appendLine을 재귀 호출해준다. 물론 다음 페이지가 NULL이라면 newPage로 새로 생성해주고 호출한다.

그런데 위 경우에 해당하지 않는 경우 중에서, 재귀를 활용해야 하는 상황이 더 있다.

바로 현재 페이지가 가득 찬 경우인데, 이 때도 마찬가지로 다음 페이지로 넘어가서 원소를 추가해야하므로, 마찬가지로 재귀 호출을 실행한다.

만약 두 경우 모두 해당하지 않는다면, 현재 페이지에 원소를 추가하면 된다.

현재 페이지의 마지막 원소 바로 다음 칸에 원소를 삽입해주고, 로컬 카운트와 글로벌 카운트를 적절히 갱신해준다.

< insertLine 함수 >

```
void insertLine(Page* page, int idx, char* str, int
cntGlobal, int *ptrCntGlobal) {
    int cntCurPage = page->count;

    // 삽입 위치가 뒷 페이지에 있는 경우
    if (idx > cntCurPage) {
        insertLine(page->nextPage, idx-cntCurPage, str,
        cntGlobal, ptrCntGlobal);
        return;
    }

    // 페이지가 꽉 찬 경우, 뒷 페이지 하나 생성 후, 마지막
    원소를 거기에 삽입
    if (cntCurPage == 10) {
        if (page->nextPage == NULL) {
            page->nextPage = newPage(page);
        } else {
            Page * nextTmpPtr = page->nextPage;
            page->nextPage = newPage(page);
            Page * nextCurPtr = page->nextPage;
            nextCurPtr->nextPage = nextTmpPtr;
            nextTmpPtr->prevPage = nextCurPtr;
        }

        strcpy(page->nextPage->lines[0], page->lines[9]);
        page->count--;
        cntCurPage--;
        page->nextPage->count++;
    }
```

```
    // 한 칸씩 뒤로 밀기
    for (int i=cntCurPage; i>=idx; i--) {
        strcpy(page->lines[i], page->lines[i-1]);
    }

    // 확보한 자리에 원소 삽입
    strcpy(page->lines[idx-1], str);
    page->count++;
    cntCurPage++;
    (*ptrCntGlobal)++;

    return;
}
```

사용자로부터 -i 명령어와 삽입 위치, 문자열을 인자로 입력받았을 때 실행하는 함수이다.

우선 두 가지 경우를 살펴보자.

1) 삽입 위치가 뒷 페이지에 있는 경우
삽입 위치를 나타내는 idx가 현재 페이지의 원소 개수보다 큰 경우, 이는 삽입 위치가 뒷쪽 페이지에 존재한다는 것을 의미한다.

즉, 다음 페이지 포인터를 인자로 넘겨 재귀 호출한다. 이 때 idx는 현재 페이지의 원소 개수를 뺀 값을 인자로 넘겨준다.

2) 삽입 위치가 현재 페이지이지만 페이지가 꽉 차 있는 경우
이 때는 현재 페이지의 마지막 원소를 다음 페이지로 넘기고, 삽입 위치로부터 마지막 원소까지 한 칸씩 뒤로 민 다음 삽입 위치에 원소를 삽입해주어야 한다.

즉 삽입에 앞서 먼저 마지막 원소를 다음 페이지로 넘겨주는 작업을 한다.

1번과 2번 케이스 외에는, 현재 페이지에 적절히 삽입하면 된다.
삽입 위치부터 마지막 원소까지 모두 한 칸씩 뒤로 밀어주고, 삽입 위치에 원소를 덧씌워주면 된다.

이 때 페이지 카운트와 글로벌 카운트를 적절히 갱신해준다.

< removeLine 함수 >

```
void removeLine(Page* page, int idx, int cntGlobal, int *
ptrCntGlobal) {
    int cntCurPage = page->count;

    // 지울 위치가 뒷 페이지에 있는 경우
    if (idx > cntCurPage) {
        removeLine(page->nextPage, idx - cntCurPage,
        cntGlobal, ptrCntGlobal);
        return;
    }

    // 지울 위치가 페이지의 마지막 라인인 경우
    if (idx == 10) {
        page->lines[idx-1][0] = '\0';
        page->count--;
        (*ptrCntGlobal)--;
        return;
    }


    // 한 칸씩 앞으로 당기기
    for (int i=idx; i<cntCurPage; i++) {
        strcpy(page->lines[i-1], page->lines[i]);
    }
```

```
    // 마지막으로 당긴 원소가 원래 있던 칸을 빈 문자열로
    갱신해주기
    page->lines[cntCurPage-1][0] = '\0';
    page->count--;
    (*ptrCntGlobal)--;
    return;
}
```

사용자로부터 입력 받은 삭제 위치에 있는 원소를 삭제하는 함수이다.

이 때도 두 가지 경우를 생각해봐야한다.

1) 삭제 위치가 뒷쪽 페이지에 있는 경우

idx가 현재 페이지 원소 개수보다 크다면, 이는 삭제해야 할 위치가 뒷쪽 페이지에 존재한다는 의미이다.

이 때는 다음 페이지 포인터를 인자로 넘겨주면서 재귀 호출한다. 이 때, idx 자리에는 idx에 현재 페이지 원소 개수를 뺀 값을 넘겨준다.

2) 삭제 위치가 현재 페이지 마지막 원소인 경우

이 때는 원소 삭제 후 뒤쪽 원소가 없기 때문에 당겨줄 필요가 없으므로, 그냥 마지막 원소에 빈 문자열을 할당해주기만 하면 된다.

위 두 경우 모두 해당하지 않는다면, 현재 페이지에서 적절히 삭제하면 된다.

우선 삭제 위치 바로 뒤에서부터 마지막 원소까지 한 칸씩 앞으로 당기고, 마지막 원소가 원래 위치해있었던 자리에 빈 문자열을 할당해주면 끝이다.

이 때 현재 페이지의 원소 개수와 글로벌 카운트를 적절히 갱신해준다.

2-2. main 함수

이제 main 함수에서 수행하는 내용들을 살펴보자.

```
// 명령어 최초 입력
char inputTmp[100];
char args[3][100] = {"\0", "\0", "\0"};
int idxArgs;

idxArgs = 0;
gets(inputTmp);
char *ptrSplit = strtok(inputTmp, " ");
while (ptrSplit != NULL) {
    strcpy(args[idxArgs++], ptrSplit);
    ptrSplit = strtok(NULL, " ");
}
```

가장 먼저 최초로 명령어를 하나 입력받는다.
인자를 최대로 많이 받는 경우는 명령어가 -i일 때이다. 인덱스와 문자열을 포

함하여 총 3개를 받는데, 따라서 args 배열 행 개수를 3개로 설정하였고, 문자열 길이를 고려하여 넉넉하게 100개의 열로 설정하였다.

gets로 사용자로부터 입력 한 줄을 받고, 이를 공백을 기준으로 tokenize하여 args에 적절히 삽입해준다.

```
// 전체 원소 카운트, 메인 페이지 초기화
int cntGlobal = 0;
Page mainPage = { 0, NULL, NULL, NULL };

// 명령어 종류 정의
char a[] = "-a";
char i[] = "-i";
char r[] = "-r";
```

모든 구조체의 총 원소 개수를 의미하는 cntGlobal과, 첫 페이지인 mainPage 구조체를 할당했다.

그리고 a, i, r는 각각 명령어 종류를 나타내는 문자열이다.

```c
// 기존 텍스트 파일 로드해서 구조체에 집어 넣기
FILE * fileOrigin = fopen(argv[2], "r");
Page * curPagePtr = &mainPage;
int iter_cnt = 0;
int move_cnt = 0;

if (fileOrigin != NULL) {
    char strOrigin[100];

    while (fgets(strOrigin, 100, fileOrigin) != NULL) {
        if (strOrigin[strlen(strOrigin)-1] == '\n') {
            strOrigin[strlen(strOrigin)-1] = '\0';
        }
        if (iter_cnt > 0 && iter_cnt % 5 == 0) {
            curPagePtr->nextPage = newPage(curPagePtr);
            curPagePtr = curPagePtr->nextPage;
            move_cnt++;
        }
        appendLine(curPagePtr, strOrigin, cntGlobal -
        move_cnt * 5, &cntGlobal);
        iter_cnt++;
    }

    fclose(fileOrigin);
}
```

기존에 있던 텍스트 파일의 내용을 불러와 구조체에 적절히 삽입하는 구문이다.

fopen으로 텍스트 파일을 불러오고, fgets로 한 줄씩 읽어오면서 NULL일 때까지 계속 구조체에 삽입해준다.

이 때 텍스트 파일에서 한줄을 가져올 때 개행 문자로 같이 가져오므로, 이 부분을 null string으로 바꾸어 개행 문자를 제거해준다.

그리고 최적화를 위해, 각 구조체마다 5개가 삽입되면 다음 페이지로 넘겨서 이어서 삽입하도록 구현했다.

삽입하는 기능은 appendLine으로 수행한다. 이 때 다음 페이지로 넘어가서

첫 라인부터 바로 삽입할 수 있도록, cntGlobal에서, 페이지 이동 횟수를 나타내는 move_cnt에 5를 곱한 값을 빼주어 인자로 넘겨준다.

```
char * cmd;
char str[100];
int idx;
```

```
// 명령어 실행 (exit이 들어올 때까지 반복)
while (strcmp(args[0], "exit") != 0) {
    cmd = args[0];

    if (strcmp(cmd, a) == 0) { // 명령어가 -a 일 때
        strcpy(str, args[1]);
        if (str[strlen(str)-1] == '\n') { // 개행 문자
        제거
            str[strlen(str)-1] = '\0';
        }
        if (strlen(str) == 0) { // 빈 문자열이 입력된
        경우
            printf("input text data invalid!! (null
            error)\n");
        } else {
            appendLine(&mainPage, str, cntGlobal, &
            cntGlobal);
        }
```

명령어로 exit이 들어올 때까지 반복하는 while문이다.

만약 명령어가 -a 이라면, appendLine을 수행해준다.

이 때, 문자열의 마지막 문자가 개행 문자라면, 여기에 null string을 덧씌워 제거해준다.

만약 문자열이 빈 문자열이라면 경고 메시지를 출력한다. 그렇지 않다면 appendLine 함수를 수행하여 원소를 삽입해준다.

```
} else if (strcmp(cmd, i) == 0) { // 명령어가 -i 일
때
    idx = atoi(args[1]);
    strcpy(str, args[2]);
    if (str[strlen(str)-1] == '\n') { // 개행 문자
    제거
        str[strlen(str)-1] = '\0';
    }
    if (idx < 1 || idx > cntGlobal) { // 삽입 위치
    인덱스가 유효한 값이 아닌 경우
        printf("insert invalid!!\n");
    } else if (strlen(str) == 0) { // 빈 문자열이
    입력된 경우
        printf("input text data invalid!! (null
        error)\n");
    } else {
        insertLine(&mainPage, idx, str, cntGlobal, &
        cntGlobal);
    }
```

명령어가 -i 일 때의 실행 내용이다.

우선 문자열의 마지막 문자가 개행인 경우 null string으로 덧씌워준다.

만약 입력받은 idx 값이 유효하지 않은 경우, (1 미만이거나 cntGlobal을 초과하는 경우) 경고 메시지를 출력한다.

입력받은 문자열이 빈 문자열인 경우에도 경고 메시지를 출력한다.

그 외의 경우에는 정상적으로 insertLine을 수행해준다.

```
} else if (strcmp(cmd, r) == 0) { // 명령어가 -r 일
때
    idx = atoi(args[1]);
    if (idx < 1 || idx > cntGlobal) { // 삭제 위치
    인덱스가 유효한 값이 아닌 경우
        printf("remove invalid!!\n");
    } else {
        removeLine(&mainPage, idx, cntGlobal, &
        cntGlobal);
    }
} else { // 입력된 명령어 종류가 유효한 값이 아닌 경우
    printf("command invalid!!\n");
}
```

명령어가 -r 인 경우 실행하는 내용이다.

입력 받은 삭제 위치 idx가 유효하지 않은 경우 경고 메시지를 출력하고, 그렇지 않은 경우 정상적으로 removeLine을 실행해준다.

그 외의 경우에는 명령어가 잘못 입력된 경우이므로 경고 메시지를 출력한다.

```
// 디버깅 및 구조 확인을 위한, 구조체 원소 출력 코드
int idx_debug = 0; // 구조체 lines에 접근할 index
int cntPut_debug = 0; // 전체적으로 원소 출력해낸
개수
int itemCntPage_debug = 0; // 현재 구조체에서
출력해낸 원소 개수
int pageNum_debug = 1; // 구조체 번호
Page * ptrPage_debug = &mainPage;
char * strItem_debug = ptrPage_debug->lines[0];
```

```
printf
("=========================================\n");
while (cntPut_debug < cntGlobal) {
    if (idx_debug == 0) {
        printf("<page %d>\n", pageNum_debug++);
    }

    if (strlen(strItem_debug) != 0) { // 특정
    자리에 원소가 존재한다면 출력해줌
        printf("%s\n", strItem_debug);
        cntPut_debug++;
        itemCntPage_debug++;
    }

    idx_debug++;

    // 현재 구조체의 원소를 모두 출력했다면 다음
    구조체로 포인터 옮기기
    if (itemCntPage_debug == ptrPage_debug->count) {
        ptrPage_debug = ptrPage_debug->nextPage;
        idx_debug = 0;
        itemCntPage_debug = 0;
    }

    strItem_debug = ptrPage_debug->lines[idx_debug];
}
printf
("=========================================\n");
```

명령어를 하나 실행할 때마다 모든 구조체의 원소들을 페이지별로 출력하는 부분이다. 디버깅 및 구조 확인에 용이하다.

idx_debug를 1씩 증가시키면서 구조체의 원소들을 순차적으로 조회한다. 만약 해당 위치의 원소가 빈 문자열이 아니라면, 이를 출력하고 cntPut_debug에 출력 원소 개수를 기록한다.

만약에 itemCntPage_debug와 현재 페이지의 count가 같다면, 다음 페이지로 포인터를 옮기고 idx_debug와 itemCntPage_debug를 적절히 초기화한다.

이를 cntPut_debug가 cntGlobal보다 작은 동안 반복 수행한다.

```
    // 명령어와 인자들을 담은 args 배열 초기화해주기
    for (int k=0; k<3; k++) {
        args[k][0] = '\0';
    }

    // 명령어와 인자들 다시 입력 받기
    idxArgs = 0;
    gets(inputTmp);
    char *ptrSplit = strtok(inputTmp, " ");
    while (ptrSplit != NULL) {
        strcpy(args[idxArgs++], ptrSplit);
        ptrSplit = strtok(NULL, " ");
    }
}
```

다음 명령어 입력 및 수행을 위해 args를 적절히 초기화해주고, 사용자로부터 명령어와 인자를 다시 입력받아 args에 할당해준다.

```
// 동작이 끝난 파일을 write 하는 코드
FILE * file = fopen(argv[2], "w");
idx = 0; // 구조체 문자열 조회 인덱스
int cntPut = 0; // 파일에 넣은 개수
int itemCntPage = 0; // 현재 구조체에서 파일에 넣은 원소
개수
int pageNum = 1; // 구조체 번호
Page * ptrPage = &mainPage;
char * strItem = ptrPage->lines[0];
```

```
while (cntPut < cntGlobal) {
    if (strlen(strItem) != 0) {
        fputs(strItem, file);
        fputs("\n", file);
        cntPut++;
        itemCntPage++;
    }

    idx++;

    // 현재 구조체에서 원소 다 파일에 넣었으면 다음
    구조체로 포인터 이동
    if (itemCntPage == ptrPage->count) {
        ptrPage = ptrPage->nextPage;
        idx = 0;
        itemCntPage = 0;
    }

    strItem = ptrPage->lines[idx];
}

fclose(file);
```

명령어로 exit을 받으면 수행을 종료하고, 구조체의 모든 원소를 텍스트 파일에 write한다.

로직은 명령어 실행 while문 내에서의 구조체 원소 출력 코드와 유사하다.

idx를 1씩 증가시키면서 구조체의 원소를 순차적으로 순회한다.

만약 idx 위치의 원소가 빈 문자열이 아닌 경우 fputs로 파일에 작성해주고, 파일에 작성한 원소 개수를 의미하는, cntPut과 현재 페이지에서 파일에 작성해낸 원소 개수를 의미하는 itemCntPage를 1씩 증가시켜준다.

만약 itemCntPage가 현재 페이지의 count와 같다면, 다음 페이지로 옮기면서 idx와 itemCntpage를 적절히 초기화해준다.

```
Page * tmpPtr = &mainPage;
int ii = 1;

while (tmpPtr != NULL) {
    printf("%d번째 구조체 원소 개수 : %d\n", ii++,
    tmpPtr->count);
    tmpPtr = tmpPtr->nextPage;
}
```

파일에 작성을 끝마친 후, 마지막으로 모든 구조체의 각각의 원소 개수를 편의상 출력해준다.

```
// 동적 메모리 free 하는 코드 작성
Page * iterPagePtr = mainPage.nextPage;
Page * tmpPagePtr;
while (iterPagePtr != NULL) {
    tmpPagePtr = iterPagePtr->nextPage;
    free(iterPagePtr);
    iterPagePtr = tmpPagePtr;
}

return 0;
```

수행 과정에서 Page 구조체를 동적 메모리 할당했던 부분들을 모두 free 해주고 프로그램을 종료한다.

3. 가산점 항목 구현

< 연속적인 명령어 입력 >

```
// 명령어 실행 (exit이 들어올 때까지 반복)
while (strcmp(args[0], "exit") != 0) {
    cmd = args[0];
```

```
// 명령어와 인자들을 담은 args 배열 초기화해주기
for (int k=0; k<3; k++) {
    args[k][0] = '\0';
}

// 명령어와 인자들 다시 입력 받기
idxArgs = 0;
gets(inputTmp);
char *ptrSplit = strtok(inputTmp, " ");
while (ptrSplit != NULL) {
    strcpy(args[idxArgs++], ptrSplit);
    ptrSplit = strtok(NULL, " ");
}
```

while문을 수행한다. 탈출 조건은 명령어로 exit이 들어올 때이다.

각 단계에서 명령어에 맞는 수행 내용을 실행한 다음에는, args를 초기화하고 다시 사용자로부터 gets로 한 줄을 입력받아, 그 것을 tokenize하고 args에 적절히 할당한 후 다시 while의 조건식을 판단한다.

< Page 최적화 >

```c
// 기존 텍스트 파일 로드해서 구조체에 집어 넣기
FILE * fileOrigin = fopen(argv[2], "r");
Page * curPagePtr = &mainPage;
int iter_cnt = 0;
int move_cnt = 0;

if (fileOrigin != NULL) {
    char strOrigin[100];

    while (fgets(strOrigin, 100, fileOrigin) != NULL) {
        if (strOrigin[strlen(strOrigin)-1] == '\n') {
            strOrigin[strlen(strOrigin)-1] = '\0';
        }
        if (iter_cnt > 0 && iter_cnt % 5 == 0) {
            curPagePtr->nextPage = newPage(curPagePtr);
            curPagePtr = curPagePtr->nextPage;
            move_cnt++;
        }
        appendLine(curPagePtr, strOrigin, cntGlobal -
        move_cnt * 5, &cntGlobal);
        iter_cnt++;
    }

    fclose(fileOrigin);
}
```

기존 텍스트 파일을 로드할 때, Page 최적화를 수행하면서 삽입하는 코드이다.

원래는 appendLine을 반복적으로 수행하면, 현재 페이지에 계속 넣다가 가득 차서 페이지 count가 10이 되었을 때 다음 페이지로 넘어가서 삽입을 계속 수행할텐데,

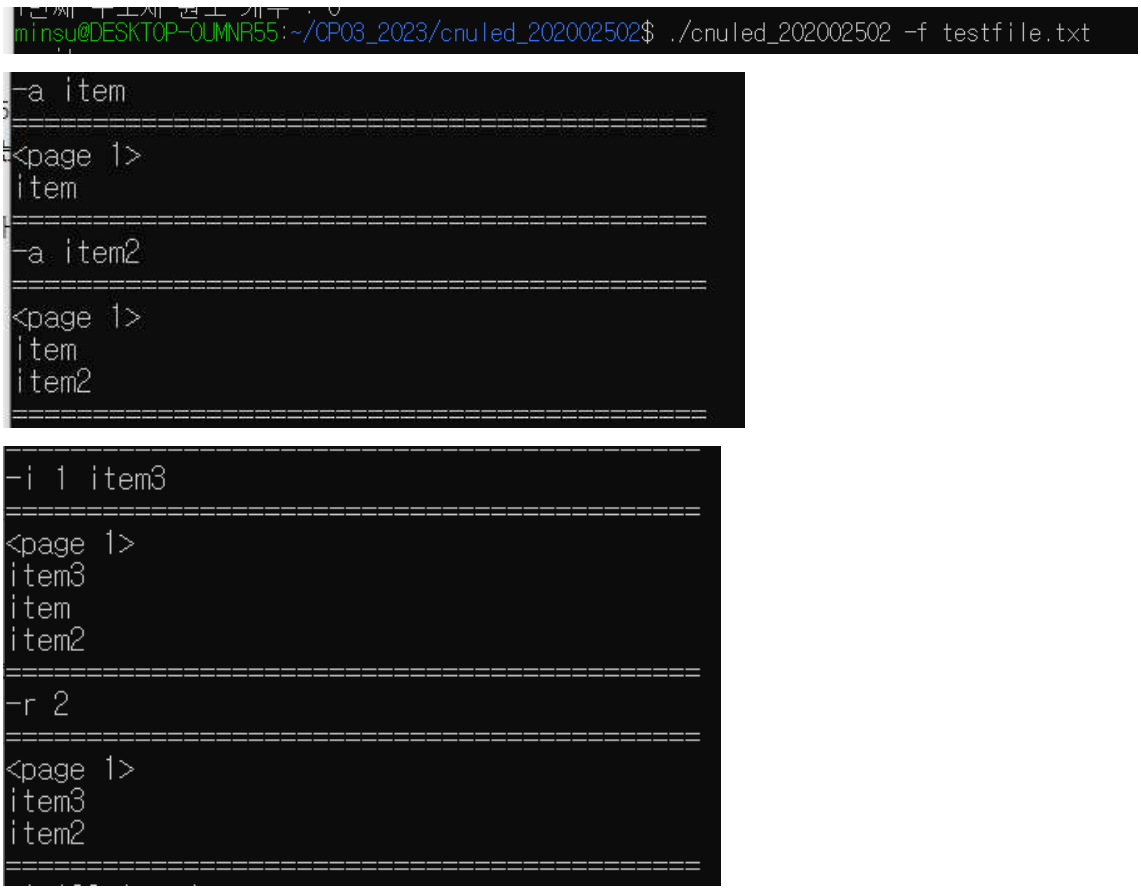
최적화를 위해 페이지에 5개의 원소가 삽입되면 다음 페이지로 넘어가서 삽입하도록 코드를 작성했다.

iter_cnt는 현재 시점까지 넣은 원소의 개수를 의미한다. 만약 이 값이 0보다 크고 5의 배수라면, 포인터를 다음 페이지로 이동시키고 move_cnt를 1 증가시킨다. (move_cnt는 페이지 이동 횟수를 의미함)

그리고 appendLine을 호출할 때는, cntGlobal에 move_cnt*5를 빼준 값을 인자로 넘겨준다.

이렇게 되면 예를 들어 cntGlobal이 5, 6, 7, 8일 때 실제로 인자로 넘기는 값은 0, 1, 2, 3이 된다. 그리고 이 때는 포인터가 다음 페이지로 넘어간 상태이므로, appendLine을 실제로 수행할 때는 다음 페이지의 첫 번째, 두 번째, 세 번째, 네 번째 자리에 원소를 삽입하게 되는 꼴이 된다.

4. 수행 결과 캡처



5. 느낀 점

평가 조건이 꽤 널널한 편이고, 그에 따라 개발의 자유도도 높아서 딱히 막히거나 어려운 부분은 없었다.

개인적으론 시험 기간임을 고려해봐도 딱 적당한 난이도인 것 같다.