자료구조2 실습
13주차 과제

제출일	21.11.	전 공	컴퓨터소프트웨어공학과
과 목	자료구조 2 실습	학 번	20184612
담당교수	홍 민 교수님	이 름	김동민

| 목 차 |

1. 정렬: 쉘 정렬 프로그램

1.1 문제 분석
1.2 소스 코드
1.3 소스 코드 분석
1.4 실행 창
1.5 느낀 점

2. 정렬: 합병 정렬 프로그램

2.1 문제 분석
2.2 소스 코드
2.3 소스 코드 분석
2.4 실행 창
2.5 느낀 점

3. 정렬: 퀵 정렬 프로그램

3.1 문제 분석
3.2 소스 코드
3.3 소스 코드 분석
3.4 실행 창
3.5 느낀 점

4. 정렬을 하며 느낀 점

1. 정렬

쉘 정렬 프로그램

- data.txt에 학생의 정보가 이름, 학번, 전화번호로 저장되어 있다. 이를 읽어와 쉘 정렬을 이용하여 학번 순으로 내림차순 정렬하여 출력하시오

1.1 문제 분석

조건 1 동적할당을 통하여 파일 데이터의 개수에 맞게 입력받음.
조건 2 쉘 정렬을 이용하여 학번 순으로 내림차순 정렬하여 출력
조건 3 data.txt파일에는 이름, 학번, 전화번호가 있고 직접 생성하여 입력받음

- 이 문제는 쉘 정렬을 이용하여 파일로부터 읽어온 데이터를 학번 순으로 내림차순 정렬하여 출력하는 문제이다. 쉘 정렬은 요소들이 멀리 떨어진 위치로도 이동할 수 있는 정렬이다. 리스트를 일정한 기준에 따라 분류하여 연속적이지 않은 여러 개의 부분 리스트를 만들고, 각 부분리스트를 삽입 정렬을 이용하여 정렬을 수행한다. 처음에는 간격을 전체의 절반으로 시작하여 간격을 절반씩 줄여가며 크기를 줄여간다. 만약 간격이 짝수가 된다면 간격에 1을 더하여 정렬을 수행한다.
쉘 정렬의 복잡도는 최악의 경우는 O(n^2)지만 평균적으로 O(n^1.5)이기 때문에 앞에서 했던 정렬 방법보다는 빠른 정렬이라고 할 수 있다.
이 문제의 경우 데이터파일에는 이름, 학번, 전화번호가 저장되어 있기 때문에 이를 모두 저장할 수 있는 구조체를 선언하고, 이 구조체를 포인터로 선언하여 파일 데이터의 개수에 맞게 동적할당하여 사용할 것이다. 또한 학번 순으로 내림차순 정렬을 해야 하기 때문에 데이터를 비교할 때는 구조체 안의 학번 값을 이용하여 비교를 수행한다.

1.2 소스 코드

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {        //학생 정보 구조체
    char* name;         //이름 포인터
    int num;            //학번 정수
    char* phone;        //전화번호 포인터
}Student;

void inc_insertion_sort(Student list[], int first, int last, int gap);
void shell_sort(Student list[], int n);
void print(Student list[], int count);

int main() {
    FILE *fp;               //파일포인터
    int count = 0;          //배열의 개수
    Student *data;          //학생 데이터 포인터
    char name[20], phone[20];
    int num;
    int i;

    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {             //파일 끝까지 반복하여 임시변수에 입력받으면서 count를 증가
        fscanf(fp, "%s%d%s", name, &num, phone);
        count++;
    }
    rewind(fp);                     //파일포인터를 맨앞으로 돌림
    data = (Student*)malloc(sizeof(Student)*count);     //data포인터 동적할당
    for(i=0; i<count; i++) {
        fscanf(fp, "%s%d%s", name, &data[i].num, phone);    //파일로부터 데이터 입력 학번은 직접 입력받음
        data[i].name = (char*)malloc(sizeof(char)*(strlen(name) + 1));  //name포인터 동적할당
        strcpy(data[i].name, name);                                     //문자열 복사

        data[i].phone = (char*)malloc(sizeof(char)*(strlen(phone) + 1));    //phone포인터 동적할당
        strcpy(data[i].phone, phone);                                       //문자열 복사
    }
    printf("<정렬 전> \n");
    print(data, count);     //정렬 전 배열 출력

    shell_sort(data, count);    //쉘정렬 함수 호출

    printf("\n<정렬 후>\n");
    print(data, count);     //정렬 후 배열 출력


    for (i = 0; i < count; i++) {   //이름포인터와 phone포인터 동적할당 해제
        free(data[i].name);
        free(data[i].phone);
    }
    free(data);         //data배열 동적할당 해제
    fclose(fp);         //파일포인터 close
    return 0;
}
void inc_insertion_sort(Student list[], int first, int last, int gap) { //입력 간격으로 떨어져 있는 요소들을 삽입
    int i, j;
    Student key;                                    //정렬을 하기 위한 사물
    for (i = first + gap; i <= last; i = i + gap) {     //gap만큼 이동하여 앞끼리 비교
        key = list[i];                              //
        for (j = i - gap; j >= first && key.num < list[j].num; j = j - gap) { //만약 key의 num값이 리스트[j]의 num값보다 크다면 수행
            list[j + gap] = list[j];            //list[j]의 값을 뒤로 이동
        }
        list[j + gap] = key;                //
    }
}
void shell_sort(Student list[], int n) {    //쉘 정렬 함수
    int i, gap;
    for (gap = n / 2; gap > 0; gap = gap / 2) {//gap을 절반의 절반씩의 간격이 1이 될때까지 간격을 줄임
        if ((gap % 2) == 0)gap++;           //만약 gap이 짝수라면 gap에서 1 증가
        for (i = 0; i < gap; i++) {         //부분리스트의 개수 gap만큼 삽입정렬 함수를 반복
            inc_insertion_sort(list, i, n - 1, gap);
        }
    }
}
void print(Student list[], int count) {     //배열 출력 함수
    int i;
    for (i = 0; i < count; i++) {       //배열의 개수 count만큼 반복하여 배열 출력
        printf("%s %d %s\n", list[i].name, list[i].num, list[i].phone);
    }
}
```

1.3 소스 코드 분석

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

typedef struct {          //학생 정보 구조체
    char* name;           //이름 포인터
    int num;              //학번 정수
    char* phone;          //전화번호 포인터
}Student;
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. 학생 데이터를 저장할 구조체를 선언한다.
구조체 안에는 이름을 저장할 문자 포인터 name, 학번을 저장할 정수형 num, 전화번호를 저장할 문자 포인터 phone을 가지고 있다.

```
void inc_insertion_sort(Student list[], int first, int last, int gap);
void shell_sort(Student list[], int n);
void print(Student list[], int count);
```

3. 필요한 함수들을 선언한다.

- 일정한 간격으로 떨어져 있는 요소들을 삽입 정렬할 inc_insertion_sort 함수를 선언한다.
- 간격을 나타내는 gap을 줄이는 shell_sort함수를 선언한다.
- 배열을 순서대로 출력하는 print함수를 선언한다.

```
int main() {
    FILE *fp;                 //파일포인터
    int count = 0;            //배열의 개수
    Student *data;            //학생 데이터 포인터
    char name[20], phone[20];
    int num;
    int i;

    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {               //파일 끝까지 반복하며 임시변수에 담으면서 count를 증가
        fscanf(fp, "%s%d%s", name, &num, phone);
        count++;
    }
    rewind(fp);                       //파일포인터를 앞으로 옮김
```

4. 필요한 변수들을 선언한다.

- fp는 파일포인터 name, phone, num은 파일로부터 데이터를 입력받고 count는 파일 데이터의 개수를 센다.
- data는 Student 구조체 포인터로 학생 수에 맞게 동적할당하여 사용한다.

5. data.txt파일을 읽기 형식으로 open한다.
만약 파일이 존재하지 않는다면 file not open을 출력하고 프로그램을 종료한다.

6. 만약 파일이 존재한다면 파일 feof함수를 이용하여 파일을 끝까지 입력받는다. 임시 변수 name, num, phone에 파일 데이터를 입력받고 데이터의 개수를 나타내는 count를 증가시킨다.

7. rewind함수를 이용하여 파일포인터를 다시 앞으로 옮긴다.

```
data = (Student*)malloc(sizeof(Student)*count);     //data포인터 동적할당
for(i=0; i<count; i++) {
    fscanf(fp, "%s%d%s", name, &data[i].num, phone);    //파일로부터 데이터 입력 학번은 직접 입력받음
    data[i].name = (char*)malloc(sizeof(char)*(strlen(name) + 1));  //name포인터 동적할당
    strcpy(data[i].name, name);                                     //문자열 복사

    data[i].phone = (char*)malloc(sizeof(char)*(strlen(phone) + 1));    //phone포인터 동적할당
    strcpy(data[i].phone, phone);                                       //문자열 복사
}
printf("<정렬 전> ₩n");
print(data, count);     //정렬 전 배열 출력

shell_sort(data, count);    //쉘정렬 함수 호출

printf("₩n<정렬 후>₩n");
print(data, count);     //정렬 후 배열 출력

for (i = 0; i < count; i++) {   //이름포인터와 phone포인터 동적할당 해제
    free(data[i].name);
    free(data[i].phone);
}
free(data);         //data배열 동적할당 해제
fclose(fp);         //파일포인터 close
return 0;
```

8. 위에서 구한 데이터의 개수 count변수를 이용하여 data포인터를 동적할당하여 생성한다.

9. 파일 데이터의 개수를 구했으므로 feof가 아니라 count를 이용하여 for문으로 데이터를 다시 입력받는다.
학번은 바로 입력이 가능하기 때문에 직접 입력을 받고 name과 phone은 임시 변수에 입력 받은 후 문자열의 길이 + 1로 data안의 name과 phone 포인터를 동적할당한 후 strcpy로 복사한다.

10. 데이터를 모두 입력 받았다면 print함수를 이용하여 정렬 전 배열을 출력하고, shell_sort함수를 호출하여 쉘정렬을 수행한다.

11. 쉘 정렬이 끝난 후 다시 print함수를 이용하여 정렬 후에 배열의 변화를 확인한다.

12. 모든 동작이 끝나면 for문을 이용해 count만큼 반복하며 data배열에서 동적할당한 name포인터와 phone포인터를 메모리 해제한다.
그리도 data배열도 free를 이용하여 동적할당을 해제해주고 fclose로 파일 포인터를 닫은 후 프로그램을 종료한다.

```
void inc_insertion_sort(Student list[], int first, int last, int gap) { //일정 
    int i, j;
    Student key;                                        //정렬을 하기 위해 사용
    for (i = first + gap; i <= last; i = i + gap) {     //gap만큼 이동하며 쉘끼리
        key = list[i];                                  //
        for (j = i - gap; j >= first && key.num > list[j].num; j = j - gap) { /
            list[j + gap] = list[j];                //list[j]의 값을 뒤로 이동
        }
        list[j + gap] = key;                //
    }
}
```

13. inc_insertion_sort함수는 일정한 간격으로 떨어져 잇는 요소들을 삽입 정렬한다. 매개 변수로는 정렬할 배열과 시작인덱스, 끝 인덱스, 부분리스트의 개수 gap을 전달받는다.

14. I를 시작값 + 갭의 크기부터 끝 인덱스 까지 반복한다. I를 증가시킬 때는 gap만큼 증가한다. 그리고 비교을 위해 사용할 key 변수에 list의 값을 저장한다.

15. 내부 for 반복문은 I-gap부터 first까지 반복하며 j를 gap씩 감소시키며 반복한다. 반복을 하며 key의 학번이 더 큰지 비교하고, 만약 key의 학번이 더 크다면 배열의 [j+gap]인덱스에 배열의 [j]인덱스를 저장한다.

16. 내부 반복문이 종료되면 [j+gap]인덱스에 key를 저장한다.

```
void shell_sort(Student list[], int n) {    //쉘 정렬
    int i, gap;
    for (gap = n / 2; gap > 0; gap = gap / 2) {//gap을
        if ((gap % 2) == 0)gap++;           //만약 ga
        for (i = 0; i < gap; i++) {         //부분리스
            inc_insertion_sort(list, i, n - 1, gap);
        }
    }
}
```

17. shell_sort함수는 쉘 정렬을 하는 함수이다. 매개변수는 정렬할 배열 list와 배열의 크기 n을 전달받는다.

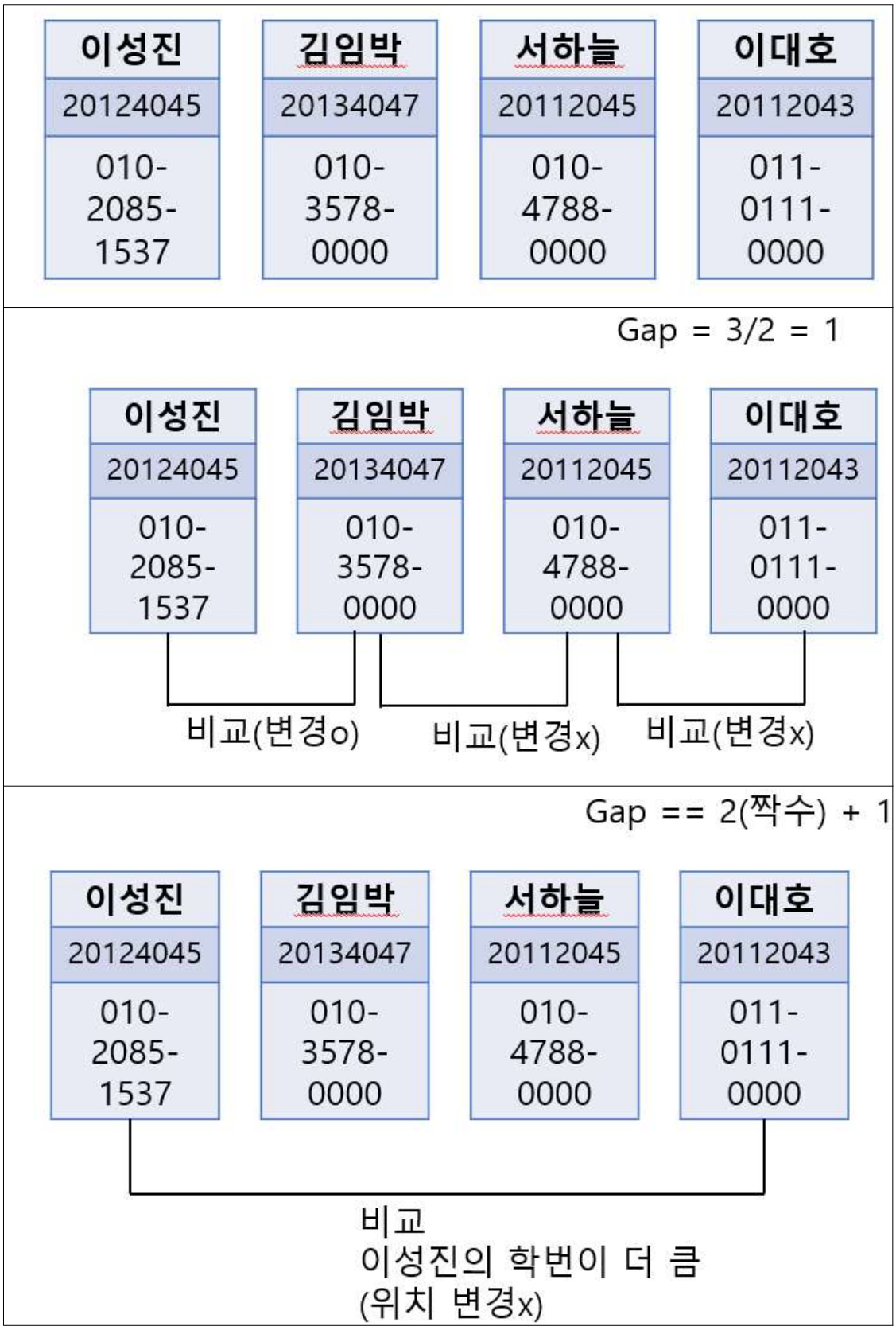
18. 처음 gap의 크기를 정렬할 배열의 크기의 /2로 시작하고 gap을 /2하며 gap이 1이 될 때까지 반복한다.

19. 만약 gap이 짝수가 나온다면 gap을 1증가시킨다.
gap만큼 다시 for문을 반복하며 요소 삽입 정렬 함수 inc_insertion_sort 함수를 호출한다.

```
void print(Student list[], int count) {     //배열 출력 함수
    int i;
    for (i = 0; i < count; i++) {       //배열의 개수 count만큼 반복하며
        printf("%s %d %s₩n", list[i].name, list[i].num, list[i].phone);
    }
}
```

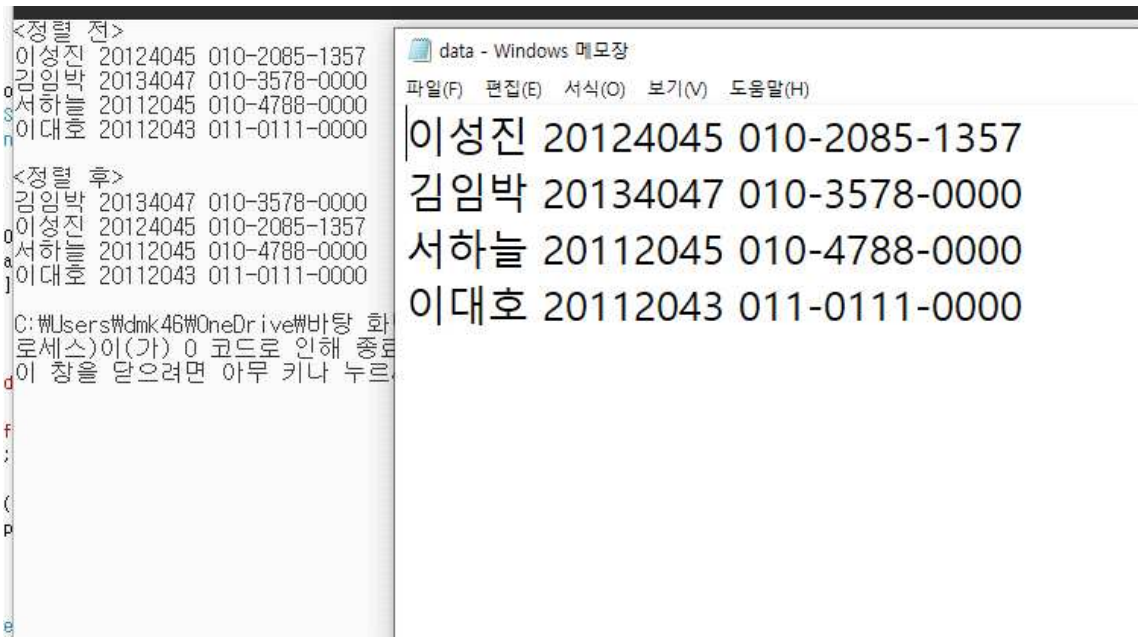
20. print함수는 배열에 있는 데이터를 순서대로 출력하는 함수이다.
배열의 크기 count만큼 for문을 반복하며 배열에 있는 이름, 학번, 전화번호를 순서대로 출력한다.

쉘 정렬의 진행 과정


이성진
20124045
010-2085-1537
김임박
20134047
010-3578-0000
서하늘
20112045
010-4788-0000
이대호
20112043
011-0111-0000
Gap = 3/2 = 1
이성진
20124045
010-2085-1537
김임박
20134047
010-3578-0000
서하늘
20112045
010-4788-0000
이대호
20112043
011-0111-0000
비교(변경o)
비교(변경x)
비교(변경x)
Gap == 2(짝수) + 1
이성진
20124045
010-2085-1537
김임박
20134047
010-3578-0000
서하늘
20112045
010-4788-0000
이대호
20112043
011-0111-0000
비교
이성진의 학번이 더 큼
(위치 변경x)


정렬 완료
김임박
20134047
010-3578-0000
이성진
20124045
010-2085-1537
서하늘
20112045
010-4788-0000
이대호
20112043
011-0111-0000

1.4 실행 창


<정렬 전>
이성진 20124045 010-2085-1357
김임박 20134047 010-3578-0000
서하늘 20112045 010-4788-0000
이대호 20112043 011-0111-0000
<정렬 후>
김임박 20134047 010-3578-0000
이성진 20124045 010-2085-1357
서하늘 20112045 010-4788-0000
이대호 20112043 011-0111-0000
C:₩Users₩dmk46₩OneDrive₩바탕 화
로세스)이(가) 0 코드로 인해 종료
이 창을 닫으려면 아무 키나 누르
data - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
이성진 20124045 010-2085-1357
김임박 20134047 010-3578-0000
서하늘 20112045 010-4788-0000
이대호 20112043 011-0111-0000

1.5 느낀 점

이번 문제는 파일에 있는 데이터를 읽어와 쉘 정렬을 이용하여 내림차순 정렬을 하는 문제였습니다. 지금까지 쉽고 간단한 배열만을 사용해왔기 때문에 처음에는 적응이 잘 되지 않기도 했지만 이번 기회를 계기로 쉘 정렬에 대해 자세히 알 수 있었습니다.

처음에는 리스트를 일정한 기준에 따라 분류하여 여러개의 부분 리스트를 만든다는 것이 이해가 되지 않았습니다. 하지만 직접 그림을 그려보며 코드를 진행해보니 쉘정렬을 하는 방법에 대해 이해를 할 수 있었습니다. 특히 간격이 1이 될 때까지 간격을 1/2로 계속 줄여가면서 정렬을 할 수 있다는 것이 새롭게 느껴졌습니다.

이번 과제를 계기로 제가 알고만 있던 버블 정렬보다 정렬하는 방법이 훨씬 많다는 것을 알 수 있었습니다. 이번 기회를 통해 모든 정렬 방법을 익힐 수 있도록 노력하겠습니다.

2. 정렬

합병 정렬 프로그램

- data.txt에 저장되어 있는 데이터를 불러와 합병정렬을 이용하여 오름차순으로 정렬하여 출력하시오.

2.1 문제 분석

조건 1 배열을 이용하여 작성
조건 2 리스트의 분할과 합병 등 합병정렬 되는 과정을 함께 출력.
조건 3 합병정렬을 이용하여 오름차순으로 정렬.

- 이 문제는 합병 정렬을 이용하여 파일에 있는 데이터를 오름차순 정렬하고, 출력하는 문제이다. 합병 정렬은 순환 호출을 이용하여 구현한다. 리스트를 숫자 하나가 남을 때까지 분할하고, 분할하는 과정이 끝나면 이를 다시 합병하는 과정에서 정렬을 수행한다. 합병 정렬은 분할한 배열들을 합병하는 과정에서 임시 배열이 하나 더 필요하기 때문에 전역변수로 배열 하나를 더 생성한다. 이번 문제의 경우 메모리의 크기를 아끼기 위해 임시 배열 변수와 원래 배열을 포인터로 선언하여 데이터의 개수에 맞게 동적할당하여 사용할 것이다.

이번 문제는 합병정렬이 되는 과정이 함께 출력되어야 하기 때문에 합병정렬 함수 merge_sort함수에서 왼쪽리스트, 오른쪽 리스트의 값을 출력한 후 합병된 sorted배열을 출력하여 정렬이 잘 되었는지 확인한다.

합병 정렬 함수의 총 비교 연산은 최대 nLogn번 필요하고 이동연산은 2nLogn이 필요하다. 따라서 비교연산과 이동연산 모두 O(nLogn)의 복잡도를 가지고 있다. 합병정렬의 장점은 최악, 평균, 최선 모두 O(nLogn)이기 때문에 정렬되는 시간이 동일하다는 장점이 있지만, 임시 배열이 필요하기 때문에 레코드의 크기가 크면 시간낭비가 심하다는 단점이 있다. 하지만 만약 연결리스트를 이용하여 정렬을 할 경우 훨씬 빠르게 정렬이 가능하다.

2.2 소스 코드

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

int *sorted;        //합시킬 저장할 추가 배열

void merge(int list[], int left, int  mid, int right);  //두개 리스트 합병 함수
void merge_sort(int list[], int left, int right);       //두개 리스트 함수
void Print(int arr[], int count);       //배열 출력 함수
int main() {
    FILE *fp;
    int i, count=0, temp;
    int *arr;


    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("File not open");
        return 0;
    }
    while (!feof(fp)) {
        fscanf(fp, "%d", &temp);    //파일의 데이터를 임시 변수 temp에 입력
        count++;            //파일 데이터의 개수 증가
    }
    rewind(fp);
    arr = (int*)malloc(sizeof(int)*count);      //count로 arr과 sorted포인터 동적할당
    sorted = (int*)malloc(sizeof(int)*count);

    for (i = 0; i < count; i++) {//count만큼 반복하여 파일 데이터 입력
        fscanf(fp, "%d", &temp);
        arr[i] = temp;      //arr배열에 데이터 저장
    }

    printf("정렬하지 않은 리스트₩n");
    Print(arr, count);          //합병 정렬 전 배열 출력

    merge_sort(arr, 0, count-1);        //합병 정렬 함수 호출

    printf("₩n정렬후리스트₩n");
    Print(arr, count);          //합병 정렬 후 배열 출력

    fclose(fp);     //파일포인터 닫음
    free(arr);      //arr포인터 해제
    free(sorted);   //sorted포인터 해제
    return 0;

}
void merge(int list[], int left, int  mid, int right) { //두개 리스트 합병 함수
    int i, j, k, l;
    i = left; j = mid + 1; k = left;

    while (i <= mid && j <= right) {    //합할 정렬된 리스트를 합병
        if (list[i] <= list[j]) {       //만약 왼쪽 리스트 보다 오른쪽 리스트가 더 크다면
            sorted[k++] = list[i++];    //왼쪽 리스트의 데이터를 sorted에 넣은 후 증가
        }
        else {                      //왼쪽 리스트 보다 오른쪽 리스트가 더 작다면
            sorted[k++] = list[j++];    // sorted에 오른쪽 데이터를 넣은 후 증가
        }
    }
    if (i > mid) {          //만약 i가 mid보다 더 크다면(오른쪽 리스트에 데이터가 남아있다면)
        for (l = j; l <= right; l++) {
            sorted[k++] = list[l];  //오른쪽 리스트에 남은 값을 sorted에 모두 복사
        }
    }
    else {                  //만약 i가 mid보다 더 작다면(왼쪽 리스트에 데이터가 남아있다면)
        for (l = i; l <= mid; l++) {
            sorted[k++] = list[l];  //왼쪽 리스트에 남은 값을 sorted에 모두 복사
        }
    }
    printf("₩nleft list : ");
    for (l = left; l <= mid; l++) {     //왼쪽 리스트 출력
        printf("%d ", list[l]);
    }
    printf("₩nright list : ");
    for (l = mid + 1; l <= right; l++) {    //오른쪽 리스트 출력
        printf("%d ", list[l]);
    }
    printf("₩nSorted List: ");
    for (l = left; l <= right; l++) {   //합병된 리스트 출력
        printf("%d ", sorted[l]);
        list[l] = sorted[l];
    }
    printf("₩n");
}
void merge_sort(int list[], int left, int right) {      //두개 리스트 함수
    int mid;

    if (left < right) {
        mid = (left + right) / 2;           //리스트의 중간 추출
        merge_sort(list, left, mid);        //앞쪽 크기의 반의 추가 배열로 추출
        merge_sort(list, mid + 1, right);
        merge(list, left, mid, right);  //정렬된 두개 배열을 하나의 배열이 통합
    }
}
void Print(int arr[], int count) {      //배열 출력 함수
    int i;
    for (i = 0; i < count; i++) {
        printf("%d> ", arr[i]);         //배열의 개수 count만큼 반복하여 출력
    }
    printf("₩n");
}
```

2.3 소스 코드 분석

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int *sorted;        //임시로 저장할 추가 배열

void merge(int list[], int left, int  mid, int right);  //부분 리스트 합병 함수
void merge_sort(int list[], int left, int right);       //부분 리스트 함수
void Print(int ary[], int count);       //배열 출력 함수
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. 정렬하는 배열을 임시로 저장할 sorted배열을 포인터로 선언한다. sorted변수는 메모리의 낭비가 없도록 데이터의 개수를 이용하여 동적할당하여 사용한다.

3. 필요한 함수들을 선언한다.

- merge함수는 2개의 인접한 배열을 합병하는 함수이다.
- merge_sort함수는 리스트를 균등 분할하는 함수이다.
- print함수는 배열을 순서대로 출력하는 함수이다.

```
int main() {
    FILE *fp;
    int i, count=0, temp;
    int *ary;


    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {
        fscanf(fp, "%d", &temp);    //파일의 데이터를 임시 변수 temp에 담음
        count++;            //파일 데이터의 개수 증가
    }
    rewind(fp);
    ary = (int*)malloc(sizeof(int)*count);      //count로 ary와 sorted포인터 동적할당
    sorted = (int*)malloc(sizeof(int)*count);
```

4. 필요한 변수들을 선언한다.

- fp는 파일포인터, count는 데이터 파일의 개수를 세는 변수이다.
- temp는 파일로부터 데이터를 임시로 받는 변수이다.
- ary는 포인터로 선언되어 count의 개수에 맞게 동적할당하여 파일 데이터를 저장한다.

5. 파일을 읽기 형식으로 open하고 만약 파일이 존재하지 않는다면 file not open 메시지를 출력하고 종료한다.
만약 파일이 존재한다면 파일 데이터를 임시변수 temp에 담으면서 파일 데이터의 개수인 count를 증가시킨다.

6. rewind함수를 이용하여 파일 포인터를 다시 앞으로 옮기고 ary포인터와 sorted포인터를 데이터의 개수 count를 이용하여 동적할당한다.

```
    for (i = 0; i < count; i++) {//count만큼 반복하며 파일 데이터 입력
        fscanf(fp, "%d", &temp);
        ary[i] = temp;      //ary배열에 데이터 저장
    }

    printf("정렬되지 않은 리스트₩n");
    Print(ary, count);          //합병 정렬 전 배열 출력

    merge_sort(ary, 0, count-1);        //합병 정렬 함수 호출

    printf("₩n정렬된리스트₩n");
    Print(ary, count);          //합병 정렬 후 배열 출력

    fclose(fp);     //파일포인터 닫음
    free(ary);      //ary포인터 해제
    free(sorted);   //sorted포인터 해제
    return 0;

}
```

7. count만큼 반복하며 다시 temp에 파일 데이터를 입력받은 후
ary배열에 temp를 저장한다.

8. 먼저 정렬되지 않은 리스트를 Print함수를 이용하여 출력하고
합병 정렬 함수인 merge_sort함수를 호출하여 합병정렬을 수행한다.

9. 정렬이 끝나면 Print함수로 정렬된 리스트를 출력하고
fclose로 파일포인터를 닫은 후 동적할당한 ary배열과 sorted배열을
메모리 해제한다.

```
void merge(int list[], int left, int  mid, int right) { //부분 리스트 합병 함수
    int i, j, k, l;
    i = left; j = mid + 1; k = left;

    while (i <= mid && j <= right) {    //분할 정렬된 리스트 합병
        if (list[i] <= list[j]) {       //만약 왼쪽 리스트 보다 오른쪽 리스트가 더 크다면
            sorted[k++] = list[i++];    //왼쪽 리스트의 데이터를 sorted에 담은 후 i증가
        }
        else {                      //왼쪽 리스트 보다 오른쪽 리스트가 더 작다면
            sorted[k++] = list[j++];    // sorted에 오른쪽 데이터를 담은 후 j증가
        }
    }
    if (i > mid) {          //만약 i가 mid보다 더 크다면(오른쪽 리스트에 데이터가 남아있다면)
        for (l = j; l <= right; l++) {
            sorted[k++] = list[l];  //오른쪽 리스트에 남은 값을 sorted에 모두 복사
        }
    }
    else {                  //만약 i가 mid보다 더 작다면(왼쪽 리스트에 데이터가 남아있다면)
        for (l = i; l <= mid; l++) {
            sorted[k++] = list[l];  //왼쪽 리스트에 남은 값을 sorted에 모두 복사
        }
    }
    printf("\nleft list : ");
    for (l = left; l <= mid; l++) {     //왼쪽 리스트 출력
        printf("%d ", list[l]);
    }
    printf("\nright list : ");
    for (l = mid + 1; l <= right; l++) {    //오른쪽 리스트 출력
        printf("%d ", list[l]);
    }
    printf("\nSorted List: ");
    for (l = left; l <= right; l++) {   //합병된 리스트 출력
        printf("%d ", sorted[l]);
        list[l] = sorted[l];
    }
    printf("\n");
}
```

10. merge함수는 인접한 배열 리스트를 합병하는 함수이다.
매개변수로는 정렬할 배열과, 왼쪽 끝 인덱스, 가운데 인덱스, 오른쪽 끝 인덱스를 전달받는다.

11. 변수 l는 분할한 배열의 왼쪽 리스트의 인덱스를 나타내고 j는 오른쪽 리스트의 인덱스를 나타낸다. k는 left부터 시작하여 sorted배열에 차례대로 값을 삽입한다.

12. while문을 이용하여 l는 mid까지, j는 right까지 반복한다. 만약 반복 중간에 이 값에 도달하면 반복이 종료된다.
왼쪽 리스트의 값과 오른쪽 리스트의 값을 비교한다. 만약 왼쪽 리스트의 값이 더 크면 리스트의 값을 sorted에 삽입하고 왼쪽 리스트의 인덱스 i를 증가시킨다. 반대로 오른쪽 리스트의 값이 더 크면 오른쪽 리스트의 값을 sorted에 삽입하고 오른쪽 리스트의 인덱스 j를 증가시킨다.

13. 만약 중간에 반복이 종료되었다면 어느 한 리스트가 끝에 도달했다는 의미이기 때문에 왼쪽 리스트 또는 오른쪽 리스트에 값이 남아있는 상태이다. 그러므로 리스트에 아직 남아있는 값들을 sorted에 삽입한다.
오른쪽 리스트에 값이 남아 있다면 j부터 right까지 sorted에 삽입하고 왼쪽 리스트에 값이 남아있다면 I부터 mid까지 sorted에 삽입한다.

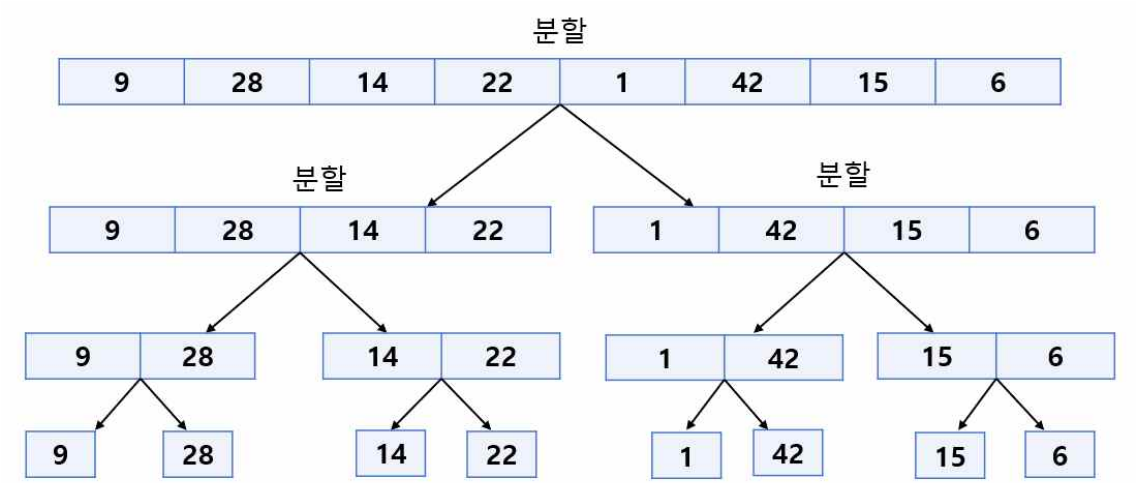
14. 합병정렬을 수행하며 진행과정을 함께 출력한다.
먼저 왼쪽 리스트를 출력하기 위해 left부터 mid까지 리스트의 데이터를 출력한다. 그리고 오른쪽 리스트를 출력하기 위해 mid+1부터 right까지 반복하며 데이터를 출력한다.
마지막으로 두 리스트가 합병된 리스트를 출력하기 위해 left부터 right까지 반복하며 sorted배열을 출력한다.

```
void merge_sort(int list[], int left, int right) {      //부분 리스트 함수
    int mid;

    if (left < right) {
        mid = (left + right) / 2;           //리스트의 균등 분할
        merge_sort(list, left, mid);        //같은 크기의 2개의 부분 배열로 분할
        merge_sort(list, mid + 1, right);
        merge(list, left, mid, right);  //정렬된 부분 배열을 하나의 배열에 통합
    }
}
void Print(int ary[], int count) {      //배열 출력 함수
    int i;
    for (i = 0; i < count; i++) {
        printf("<%d> ", ary[i]);        //배열의 개수 count만큼 반복하며 출력
    }
    printf("₩n");
}
```

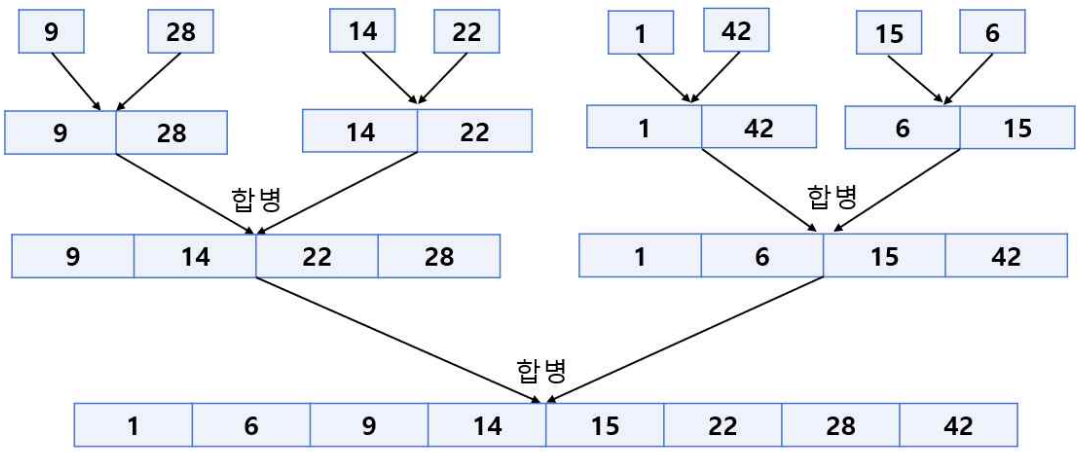
15. merge_sort함수는 주어진 배열을 2등분 하여 나눈 부분배열에 대해 다시 merge_sort함수를 호출하며 순환호출하여 수행한다.
mid값을 구하기 위해 left와 right를 더하여 2로 나누고 merge_sort를 다시 호출하여 left부터 mid 리스트와 mid+1부터 right리스트로 나눈 후 이를 다시 합치기 위해 merge함수를 호출한다.

16. Print함수는 배열의 값을 순서대로 출력하는 함수이다.
입력받은 데이터의 개수 count만큼 반복하며 배열에 있는 값을 출력한다.

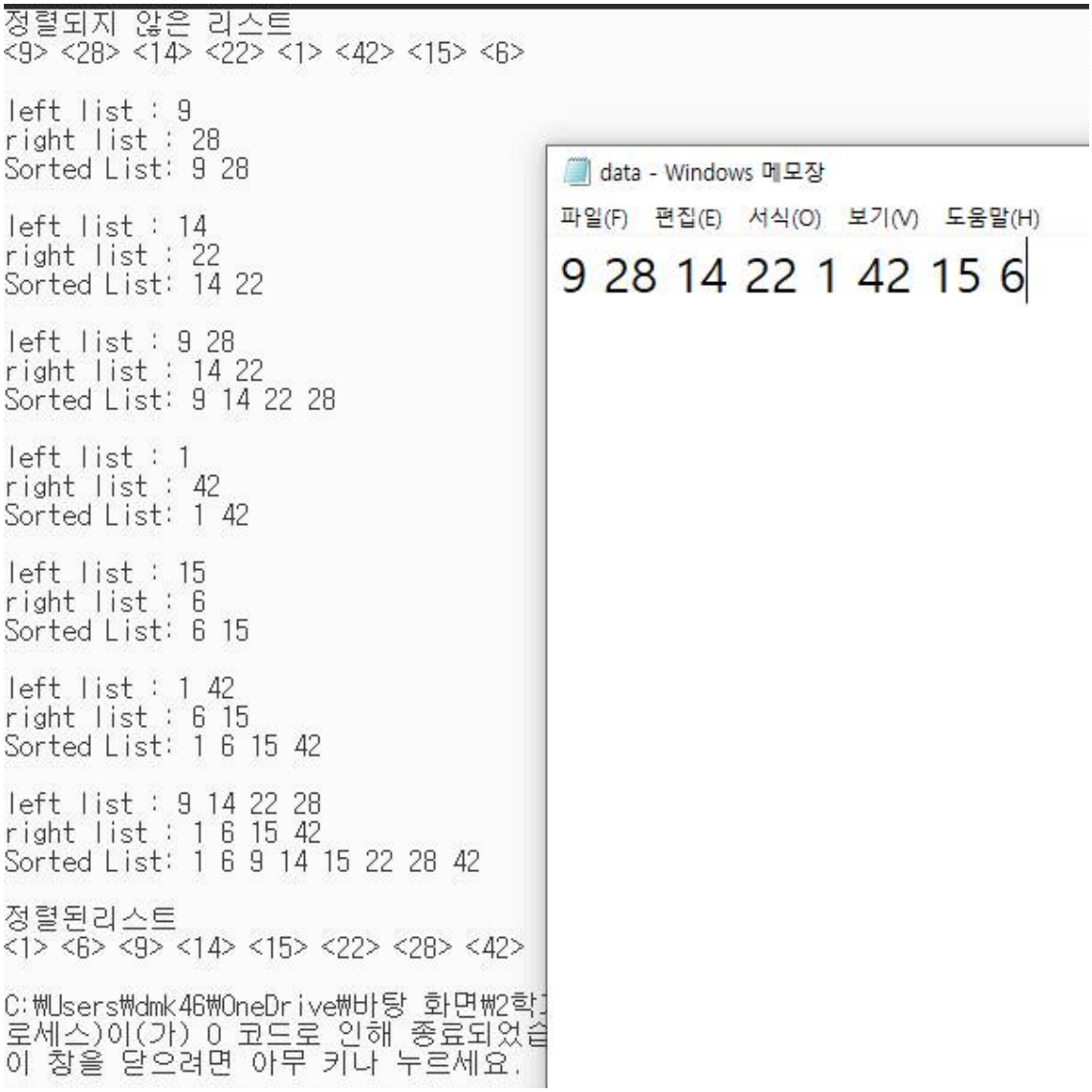
리스트의 분할 과정



리스트의 합병 과정



2.4 실행 창


정렬되지 않은 리스트
<9> <28> <14> <22> <1> <42> <15> <6>
left list : 9
right list : 28
Sorted List: 9 28
left list : 14
right list : 22
Sorted List: 14 22
left list : 9 28
right list : 14 22
Sorted List: 9 14 22 28
left list : 1
right list : 42
Sorted List: 1 42
left list : 15
right list : 6
Sorted List: 6 15
left list : 1 42
right list : 6 15
Sorted List: 1 6 15 42
left list : 9 14 22 28
right list : 1 6 15 42
Sorted List: 1 6 9 14 15 22 28 42
정렬된리스트
<1> <6> <9> <14> <15> <22> <28> <42>
C:₩Users₩dmk46₩OneDrive₩바탕 화면₩2학
로세스)이(가) 0 코드로 인해 종료되었습
이 창을 닫으려면 아무 키나 누르세요.
data - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
9 28 14 22 1 42 15 6

2.5 느낀 점

이번 문제는 합병 정렬을 이용하여 파일에 있는 데이터를 오름차순 정렬하고, 동시에 정렬하는 과정을 출력하는 문제였습니다. 이번 문제는 책에 있는 그림을 보고 이해는 되었지만 어떤 방법으로 프로그래밍을 해야 될지 잘 몰랐습니다. 하지만 교수님의 수업을 듣고 순환을 이용하여 프로그래밍하는 방법에 대해 알 수 있었습니다.

처음에는 합병 정렬은 정렬을 수행하기 위해 임시 배열공간이 필요하다는 것을 배우고 만약 레코드의 크기가 크면 시간이 오래 걸린다는 단점 때문에 잘 사용되지 않을 것이라고 생각했었는데, 그렇다고 하더라도 앞에서 배운 버블정렬이나 선택정렬보다는 빠르고 만약 연결리스트를 이용하여 정렬하면 훨씬 빠르다는 것을 배운 후 배열로만 생각하는 것이 아니라 다른 방법으로도 저장하는 방법이 많다는 것을 알 수 있었습니다.

이번 기회를 계기로 각각의 상황에 맞는 정렬방법을 사용해야겠다고 다짐했습니다. 이를 위해서는 모든 정렬을 하는 방법을 알고 있어야 하기 때문에 정렬에 대한 공부를 계속 할 것이라고 생각했습니다.

3. 정렬

퀵 정렬 프로그램

- **data.txt에 저장되어 있는 데이터를 불러와 퀵 정렬을 이용하여 오름차순으로 정렬하여 출력하시오.**

3.1 문제 분석

조건 1 동적할당을 이용하여 파일에 있는 데이터를 저장
조건 2 퀵 정렬을 이용하여 데이터를 오름차순으로 정렬
조건 3 퀵 정렬이 진행되는 과정을 함께 출력

- 이 문제는 퀵 정렬을 이용하여 파일에 있는 데이터를 오름차순 정렬하는 문제이다. 퀵 정렬은 합병 정렬과 비슷하게 전체 리스트를 2개의 부분 리스트로 분할하고, 각각의 부분 리스트를 다시 퀵정렬하는 분할-정복법을 사용한다. 퀵 정렬은 피벗을 사용한다. 피벗은 리스트 안에 존재하는 한 요소를 이용하여 선택한다. 피벗보다 작은 요소들은 피벗의 왼쪽으로 옮겨지고 큰 요소는 오른쪽으로 옮겨진다. 퀵 정렬 함수 또한 부분 리스트에서 순환 호출로 수행한다. 부분 리스트에서도 피봇을 정하고, 다시 부분리스트로 나누는 과정이 반복된다.

이 문제는 메모리를 아끼기 위해 파일에 있는 데이터의 개수를 알아내서 포인터를 개수에 맞게 동적할당하여 사용한다. 또한 퀵 정렬이 수행되는 과정을 함께 출력해야 하기 때문에 피봇값과 교환을 수행할 low값과 high값을 함께 출력한 후 배열 데이터 또한 순서대로 출력한다. low값은 왼쪽부터 시작하여 가장 처음 만나는 피봇보다 큰 수이고 high값은 오른쪽에서 시작하여 왼쪽으로 이동하며 가장 처음 만나는 피벗보다 작은 수이다. 이 high값과 low값을 서로 변경한다.

퀵 정렬은 평균적으로 O(nLogn)의 시간 복잡도를 가지고 최악의 경우 O(n^2)가 될 수도 있지만 다른 정렬 알고리즘과 비교했을 때 가장 빠르다.

3.2 소스 코드

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#define SWAP(x,y,t) ( (t) = (x), (x)=(y), (y) = (t))    //x와 y의 자리를 교환하는 SWAP매크로 정의

int partition(int list[], int left, int right);    //피소트 분할 함수
void quick_sort(int list[], int left, int right);   //퀵정렬 함수
void Print(int list[], int count);  //배열 출력함수

int main() {
    FILE *fp;
    int i, tmp, count=0;
    int *aru;                       //배열을 저장할 aru포인터 선언

    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("File not open");
        return 0;
    }
    while (!feof(fp)) {             //파일 끝(EOF) 만날때까 데이터의 개수 증가
        fscanf(fp, "%d", &tmp);
        count++;
    }
    rewind(fp);             //파일포인터를 앞으로 돌림
    aru = (int*)malloc(sizeof(int)*count);  //aru포인터 동적할당
    for (i = 0; i < count; i++) {
        fscanf(fp, "%d", &tmp);
        aru[i] = tmp;       //aru배열에 파일 데이터 값 삽입
    }
    printf("< 정렬되지 않은 피소트 >₩n");
    Print(aru, count);      //정렬 전 aru배열 출력

    quick_sort(aru, 0, count-1);        //퀵정렬 함수 호출

    printf("₩n< 정렬된 피소트 >₩n");
    Print(aru, count);      //정렬 후 aru배열 출력

    free(aru);              //aru배열 동적할당 해제
    fclose(fp);         //파일포인터 close
    return 0;
}
int partition(int list[], int left, int right) {        //피소트 분할 함수
    int pivot, temp;
    int low, high;
    int i;

    low = left;                         //low에 left값 저장
    high = right + 1;                   //high에 right다음 값 저장
    pivot = list[left];                 //피봇은 list[left]값
    printf("₩n-Pivot : %d₩n", pivot);

    do {
        do {
            low++;      //list[low]가 pivot보다 커질때까지 low를 증가
        } while (list[low] < pivot);

        printf("low : ");
        for (i = left; i <= right; i++) {
            if (list[i] == list[low]) {
                printf("%d ", list[low]);       //low값 출력
                break;
            }
        }
        if (i == right + 1) {
            printf("over ");
        }


        do {
            high--;     //list[high]가 pivot보다 작아질때까지 high를 감소
        } while (list[high] > pivot);

        printf("high : %d ", list[high]);     //high값 출력

        for (i = left; i <= right; i++) {
            printf("%d > ", list[i]);       //list의 정렬 과정 출력
        }
        printf("]₩n₩n");

        if (low < high)SWAP(list[low], list[high], temp);   //만약 high가 더 크다면 list[low]와 list[high]의 값을 교환
    } while (low < high);
    SWAP(list[left], list[high], temp); // list[low]와 list[high]의 값을 교환
    return high; //high 리턴
}
void quick_sort(int list[], int left, int right) {  //퀵정렬 함수
    if (left < right) {
        int q = partition(list, left, right);   //pivot의 위치 반환 받음

        quick_sort(list, left, q - 1);          //left에서 피봇 바로 앞까지를 다시으로 순환호출
        quick_sort(list, q + 1, right);         //피봇 위치 바로 다음부터 right까지 순환호출
    }
}
void Print(int list[], int count) {     //배열 출력함수
    int i;
    for (i = 0; i < count; i++) {
        printf("%d > ", list[i]);   //배열의 개수만큼 반복하여 배열 출력
    }
}
```

3.3 소스 코드 분석

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#define SWAP(x,y,t) ( (t) = (x), (x)=(y), (y) = (t))    //x와 y의 자리를 교환하는 SWAP매크로 정의
```

1. 소스코드를 작성한 날짜, 이름, 프로그램명을 작성하고, 필요한 헤더를 포함한다.

2. x와 y의 위치를 변경하는 SWAP매크로를 정의한다.

```
int partition(int list[], int left, int right);     //리스트 분할 함수
void quick_sort(int list[], int left, int right);   //퀵정렬 함수
void Print(int list[], int count);  //배열 출력함수
```

3. 필요한 함수들을 선언한다.

- partition함수는 피봇을 기준으로 2개의 부분리스트로 나누는 함수이다.
- quick_sort함수는 순환호출을 이용하여 배열을 정렬하는 함수이다.
- Print함수는 배열의 데이터를 순서대로 호출하는 함수이다.

```
int main( ) {
    FILE *fp;
    int i, tmp, count=0;
    int *ary;                           //배열을 저장할 ary포인터 선언

    fp = fopen("data.txt", "r");        //파일을 읽기 형식으로 open
    if (!fp) {
        printf("file not open");
        return 0;
    }
    while (!feof(fp)) {             //파일 끝까지 반복하며 데이터의 개수 증가
        fscanf(fp, "%d", &tmp);
        count++;
    }
    rewind(fp);             //파일포인터를 앞으로 옮김
    ary = (int*)malloc(sizeof(int)*count);  //ary포인터 동적할당
    for (i = 0; i < count; i++) {
        fscanf(fp, "%d", &tmp);
        ary[i] = tmp;       //ary배열에 파일 데이터 값 삽입
    }
    printf("< 정렬되지 않은 리스트 >₩n");
    Print(ary, count);      //정렬 전 ary배열 출력

    quick_sort(ary, 0, count-1);        //퀵정렬 함수 호출

    printf("₩n< 정렬된 리스트 >₩n");
    Print(ary, count);      //정렬 후 ary배열 출력

    free(ary);              //ary배열 동적할당 해제
    fclose(fp);         //파일포인터 close
    return 0;
```

4. 필요한 변수들을 선언한다.

- fp는 파일포인터, tmp는 파일로부터 데이터를 받아올 변수이다.
- count는 파일 데이터의 개수를 세고 ary 포인터는 count를 이용하여 동적할당한다.

5. 파일을 읽기 형식으로 open한다. 만약 파일이 존재하지 않는다면 file not open메시지를 출력하고 프로그램을 종료한다.
만약 파일이 존재한다면 임시변수 tmp에 파일 데이터를 입력받고, 파일 데이터의 개수를 나타내는 count를 증가시킨다.

6. rewind함수를 이용하여 파일 포인터를 앞으로 옮긴 후 ary포인터를 count를 이용하여 동적할당 생성한다.
그리고 for문을 이용하여 count만큼 반복하며 ary에 파일 데이터를 삽입한다.

7. Print함수를 이용하여 정렬되기 전의 배열을 출력한 후 퀵 정렬 함수인 quick_sort함수를 호출하고, 정렬 후의 배열을 Print함수를 이용하여 다시 출력한다.

8. 동적할당한 ary배열을 free를 이용하여 메모리 해제하고, fclose를 이용하여 파일포인터를 닫은 후 프로그램을 종료한다.

```
int partition(int list[], int left, int right) {        //리스트 분할 함수
    int pivot, temp;
    int low, high;
    int i;
    low = left;                                 //low에 left값 저장
    high = right + 1;                           //high에 right다음 값 저장
    pivot = list[left];                         //피봇을 list[left]값으로 설정
    printf("₩n-Pivot : %d-₩n", pivot);
    do {
        do {
            low++;      //list[low]가 pivot보다 커질때까지 low를 증가
        } while (list[low] < pivot);
        printf("low : ");
        for (i = left; i <= right; i++) {
            if (list[i] == list[low]) {
                printf("%d ", list[low]);       //low값 출력
                break;
            }
        }
        if (i == right + 1) {
            printf("over ");
        }
        do {
            high--;     //list[high]가 pivot보다 작아때까지 high를 감소
        } while (list[high] > pivot);

        printf("high : %d  [", list[high]);     //high값 출력

        for (i = left; i <= right; i++) {
            printf("%d > ", list[i]);       //list의 정렬 과정 출력
        }
        printf("]₩n₩n");

        if (low < high)SWAP(list[low], list[high], temp);   //만약 high가 더 크다면 list[low]와 list[high]의 값을 교환
    } while (low < high);
    SWAP(list[left], list[high], temp); // list[low]와 list[high]의 값을 교환
    return high; //high 리턴
```

9. partition함수는 데이터가 들어있는 배열 list의 left부터 right까지의 리스트를, 피봇을 기준으로 2개의 부분 리스트로 나누는 함수이다.
매개 변수로는 배열과 리스트의 왼쪽 인덱스, 오른쪽 인덱스를 전달받는다.

10. low변수에는 left값을 삽입하고, hight변수에는 right+1값을 삽입한다. 그리고 pivot은 리스트의 가장 왼쪽 값으로 설정한다.

11. 외부 반복문은 low가 high보다 작다면 do while문을 이용하여 반복을 수행한다. 내부 반복문은 먼저 list[low]가 피봇보다 커질때까지 반복을 수행한다. 만약 list[low]가 피봇보다 작다면 low값을 증가시키며 오른쪽으로 이동한다.

12. left부터 right까지 반복하며 list[low]값이 리스트 내에 존재하는지 확인한다. 만약 같은 값이 있다면 low값을 출력하고, 만약 존재하는 값이 없다면 over를 출력한다.

13. high값은 list[high]값이 피봇보다 작아질 때까지 반복하며 감소시킨다. 만약 list[high]값이 피봇보다 작아진다면 반복을 종료하고 list[high]값을 출력한다.

14. high값까지 모두 출력했다면 for문을 left부터 right까지 반복하며 list 배열 값을 출력한다. 이를 통해 퀵정렬을 하는 과정에서 배열이 변화하는 과정을 확인할 수 있다.

15. 만약 low보다 high 값이 더 크다면 list[low]와 list[hight]의 값을 SWAP 매크로를 통해 서로 교환한다.
만약 low가 더 크다면 반복을 종료하고 list[left]값과 list[high]값을 서로 교환한 후 high 값을 리턴한다.
partition함수의 반환 값은 피봇의 위치가 된다.

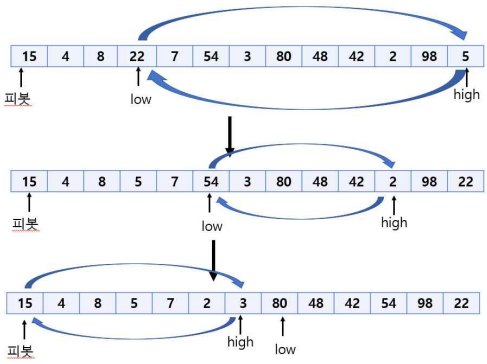
```
void quick_sort(int list[], int left, int right) {  //퀵정렬 함수
    if (left < right) {
        int q = partition(list, left, right);   //pivot의 위치 반환 받음

        quick_sort(list, left, q - 1);          //left에서 피봇 바로 앞까지를 대상으로 순환호출
        quick_sort(list, q + 1, right);         //피봇 위치 바로 다음부터 right까지 순환호출
    }
}
void Print(int list[], int count) {     //배열 출력함수
    int i;
    for (i = 0; i < count; i++) {
        printf("%d > ", list[i]);   //배열의 개수만큼 반복하며 배열 출력
    }
}
```

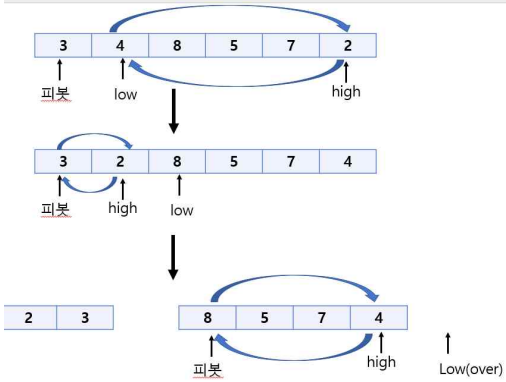
16. quick_sort함수는 순환을 이용하여 퀵 정렬을 수행하는 함수이다.
매개변수로는 정렬할 배열 list와 배열의 처음위치 인덱스, 끝 위치 인덱스를 전달받는다.

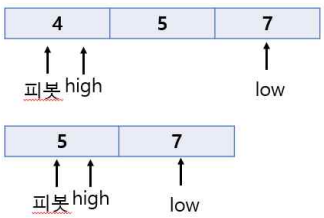
17. 만약 left가 right보다 작다면 partition함수를 이용하여 피봇의 위치를 반환받은 뒤, quick_sort함수를 순환 호출하여 left부터 피봇 앞, 피봇 다음부터 right까지 두 개의 부분 리스트로 나누어 정렬을 수행한다.

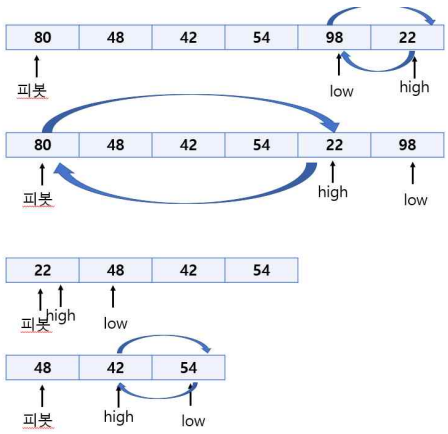
18. Print함수는 배열의 데이터를 순서에 맞게 출력하는 함수이다.
배열의 개수 count만큼 반복하며 list배열 안에 있는 데이터를 순서대로 출력한다.

퀵 정렬 알고리즘의 진행 과정



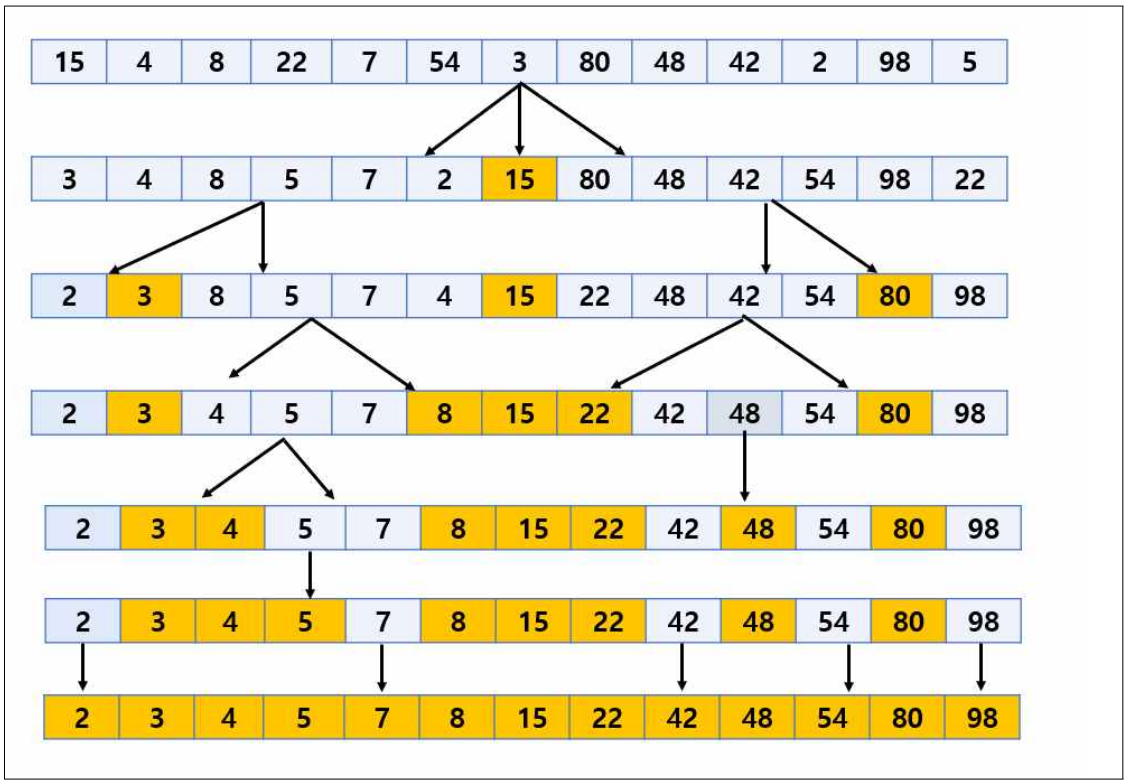






최종 배열

2	3	4	5	7	8	15	22	42	48	54	80	98


15 4 8 22 7 54 3 80 48 42 2 98 5
3 4 8 5 7 2 15 80 48 42 54 98 22
2 3 8 5 7 4 15 22 48 42 54 80 98
2 3 4 5 7 8 15 22 42 48 54 80 98
2 3 4 5 7 8 15 22 42 48 54 80 98
2 3 4 5 7 8 15 22 42 48 54 80 98
2 3 4 5 7 8 15 22 42 48 54 80 98

3.4 실행 창

```
< 정렬되지 않은 리스트 >
15 > 4 > 8 > 22 > 7 > 54 > 3 > 80 > 48 > 42 > 2 > 98 > 5 >
-Pivot : 15-
low : 22 high : 5  [15 > 4 > 8 > 22 > 7 > 54 > 3 > 80 > 48 > 42 > 2 > 98 > 5 > ]

low : 54 high : 2  [15 > 4 > 8 > 5 > 7 > 54 > 3 > 80 > 48 > 42 > 2 > 98 > 22 > ]

low : 80 high : 3  [15 > 4 > 8 > 5 > 7 > 2 > 3 > 80 > 48 > 42 > 54 > 98 > 22 > ]


-Pivot : 3-
low : 4 high : 2  [3 > 4 > 8 > 5 > 7 > 2 > ]

low : 8 high : 2  [3 > 2 > 8 > 5 > 7 > 4 > ]


-Pivot : 8-
low : over high : 4  [8 > 5 > 7 > 4 > ]


-Pivot : 4-
low : 5 high : 4  [4 > 5 > 7 > ]


-Pivot : 5-
low : 7 high : 5  [5 > 7 > ]


-Pivot : 80-
low : 98 high : 22  [80 > 48 > 42 > 54 > 98 > 22 > ]

low : 98 high : 22  [80 > 48 > 42 > 54 > 22 > 98 > ]


-Pivot : 22-
low : 48 high : 22  [22 > 48 > 42 > 54 > ]


-Pivot : 48-
low : 54 high : 42  [48 > 42 > 54 > ]


< 정렬된 리스트 >
2 > 3 > 4 > 5 > 7 > 8 > 15 > 22 > 42 > 48 > 54 > 80 > 98 >
C:₩Users₩dmk46₩OneDrive₩바탕 화면₩2학기 수업자료₩자료구조2₩실습₩week13_3 quick_s
```

3.5 느낀 점

이번 문제는 퀵정렬을 이용하여 파일에 있는 데이터를 저장하고 오름차순으로 정렬하여 출력하는 문제였습니다. 퀵 정렬은 지금 우리가 현재 사용하는 정렬 중에 가장 많이 사용한다고 배웠기 때문에 지금까지 배운 정렬 중에서 모두 익혀야 되겠지만 퀵 정렬 알고리즘만큼은 꼭 익혀야겠다고 생각했습니다.

처음에는 피봇을 기준으로 2개의 부분리스트로 나누는 partition함수가 잘 이해되지 않았는데, 직접 그림을 그리면서 코드하나하나 진행해보니 피봇을 이용하여 나눈다는 의미에 대해 알 수 있는 계기가 되었습니다. 이번 과제를 통해 피봇이 구해지는 과정과 low, high값을 출력해보며 이 값들이 구해지는 과정에 대해 직접 확인해보는 것이 알고리즘을 이해하는데 더 도움이 되었던 것 같습니다.

앞으로 정렬이 필요하다면 버블정렬같이 쉽고 간단한 정렬만 쓰는 것이 아니라 이런 복잡한 정렬을 직접 써보며 연습이 필요할 것 같다고 생각했습니다. 그렇기 때문에 퀵 정렬에 대해 잘 알아야 하고 공부를 많이 해야겠다고 생각했습니다.

4. 정렬 하며 느낀 점

저는 지금까지 버블정렬만을 이용하여 정렬을 수행했지만 상당히 좋지 않은 알고리즘이라는 것을 알고 있으면서도 다른 정렬 방법에 대해 잘 모르기 때문에 계속 사용해왔습니다. 하지만 이번 자료구조 시간에 배운 내용을 통해 버블 정렬 말고도 많은 정렬 방법이 있다는 것을 알 수 있었고 각자 상황에 맞게 사용하는 정렬도 모두 다르다는 것을 알 수 있었습니다.

상황에 맞는 정렬을 사용하려면 모든 정렬에 대해 알고 있어야 하기 때문에 이번 기회를 계기로 정렬에 대해 공부하고, 앞으로 정렬을 사용해야 되는 상황이 생긴다면 그에 맞는 정렬을 사용할 수 있도록 노력하겠습니다.