강의명: 자료구조

숙제 번호: 8

숙제 제목: Sort I| (정렬 ||)

학생 이름: 김성현

학번: 201910783

1. Merge sort(합병 정렬)

1.1

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "merge.h"

// ======================================================================

int sorted[MAX\_LIST\_LENGTH];

void merge(int list[], int left, int mid, int right)

{

//i is index that means sorted left part array.

//j is index that means sorted right part array.

//k is index that means result sorted array, sort[].

int i = left, j = mid + 1, k = left, l;

while(i <= mid && j <= right) { //merge that divided left, right array.

if(list[i] <= list[j])

sorted[k++] = list[i++]; //if left is lower than right, copy left one.

else

sorted[k++] = list[j++]; //if right is lower than left, copy right one.

}

if(i > mid) //copy if rest is right part(mid is middle).

for(l = j; l <= right; l++)

sorted[k++] = list[l];

else //copy if rest is left part.

for(l = i; l <= mid; l++)

sorted[k++] = list[l];

//copy to original array list[] from temp sorted array sorted[].

for(l = left; l <= right; l++)

list[l] = sorted[l];

}

// ======================================================================

\*이렇게 프로그램한 이유:

위 프로그램의 함수 void merge(int list[], int left, int mid, int right)는 index값 left, mid, right를 가지고 merge하는, 즉 정렬된 2개의 부분 배열을 합병하는 함수입니다. 이 merge함수는 호출되는 장소인 merge\_sort함수에서 정말 다단계처럼 반으로, 또 반으로 나눠진 배열들을 한 단계씩 합칠 때마다 호출되어 왼쪽 부분과 오른쪽 부분을 비교해서 무조건 작은 것들부터 다른 새로운 array에 차곡차곡 넣어서 비교-작은거 넣기의 작업을 반복하는 함수로써, 이렇게 반복하다가 왼쪽이나 오른쪽 중 어느 한 곳이 비게 되면 남은 쪽은 이미 정렬이 다 되어 있으므로 남은 항목들을 일괄적으로 차곡차곡 쌓아놓은 새로운 array의 뒷편에 이어 붙혀 오름차순 정렬이 완성되게 하는 함수입니다. 입력으로는 원래 리스트 array인 list[]와, 왼쪽 절반 부분의 array의 첫 시작 부분 index인 left와, 왼쪽 절반 부분의 끝이자 오른쪽 절반 부분 시작 바로 전 index인 mid, 그리고 마지막으로 오른쪽 부분의 끝 index를 가리키는 right가 있고, 출력으로는 별다른 거 없습니다. 애초에 포인터로 list를 받기 때문에 임시로 차곡차곡 쌓아서 완성시킨 임시 array를 마무리로 이 원래 list array에 그대로 복사시키면 되는 것이라 출력은 없다고 할 수 있습니다. 이러한 합병 알고리즘의 구현을 위해서 저는 일단 함수가 시작하기도 전에 전역함수로 차곡차곡 작은 숫자부터 쌓을 수 있는 공간이 있는 임시 array인 sorted를 MAX\_LIST\_LENGTH의 크기만큼 선언하여 딱 필요한 만큼만 공간을 전역변수로 확보하여 어느 곳에서도 접근할 수 있게 하였습니다. 그 다음 본격적으로 merge함수를 시작했는데, 먼저 입력으로 들어온 index정보들을 바탕으로 비교-합병을 위한 index변수들을 선언하여 각 부분에 대한 탐색-비교 준비를 하였습니다. 변수 i는 index로써 왼쪽 정렬된 배열의 첫 시작부분을 가리키는 index로 의미하기 위해 left를 넣었고, 변수 j에는 오른쪽 정렬된 배열의 첫 시작부분을 가리키는 index로 의미하기 위해 mid +1, 변수 k는 임시 array인 sorted의 각 단계 저장 현황을 표시하기 위해 배열의 첫 시작 index를 가리키는 의미도 지닌 left를 넣어 주었고, 나중에 정렬이 완성된 sorted배열을 원래 리스트array로 옮길 때 index별로 딱 딱 맞춰서 넣기 위한 index변수 l까지 선언하였습니다. 그 다음으로는 크게 3개부분으로 나눌 수 있는데, <왼쪽-오른쪽 비교해서 작은 거 임시 sorted에 차곡차곡 넣기> / <왼쪽 혹은 오른쪽에 남아있는 것들 일괄적으로 다 sorted에 이어서 넣기> / <이렇게 완성된 sorted배열을 원래 리스트 array에 그대로 복사하기>의 3개 부분으로 나눌 수 있습니다. 첫 번째로 < 왼쪽-오른쪽 비교해서 작은 거 임시 sorted에 차곡차곡 넣기>를 위해 큰 while반복문을 열어서 조건으로 left를 넣어서 증가시키면서 왼쪽 index들을 살펴볼 i가 왼쪽의 끝자락인 mid까지만 커지게 하면서(&&) 오른쪽 끝 부분인 right까지만 커지게 하도록 조건을 주어 각 부분을 벗어나지 않도록 하였습니다. 그 다음 if문으로 리스트의 왼쪽 i번째 index와 오른쪽 j번째 index를 비교하여 만약 왼쪽이 더 작거나 같을 경우 sorted배열에 왼쪽 index인 i번째 list항목을 넣고, k와 i를 증가시켜 각각의 현황이 증가되었음을, 이 경우가 아닌 경우는 오른쪽 j번째가 더 작으므로 j번째 index를 sorted에 넣고 각 k와 j변수를 증가시켜 각각 다음 index를 가리키게 하였습니다. 이렇게 왼-오른쪽 비교가 끝나게 되면 어느 한쪽만 남게 되는 경우가 있을 텐데, 이걸 위해 2번째 부분인 <왼쪽 혹은 오른쪽에 남아있는 것들 일괄적으로 다 sorted에 이어서 넣기>를 만들었습니다. if문을 만들어 조건으로 왼쪽 배열 부분의 현상황을 말하는 index인 i가 왼쪽 배열 끝 부분인 mid보다 크면 그것은 왼쪽이 다 빠져나가서 오른쪽이 남아있다는 것을 말하는 것이었으므로 변수l에 오른쪽 남은 index의 사작을 가리키는 j를 넣어서 이를 오른쪽 배열의 끝인 right까지 for문을 돌리면서 sorted에 k번째 index부터 이어서 남은 오른쪽 배열을 일괄 복사하였고, 이 경우가 아닌 경우는 왼쪽이 남아 있는 것이기에 l에 왼쪽 남은 배열의 시작 index인 i를 왼쪽 배열의 끝인 mid까지 for문을 돌려 sorted의 k번째 index부터 이어서 일괄 복사하도록 하였습니다.

그 다음 마지막으로 <이렇게 완성된 sorted배열을 원래 리스트 array에 그대로 복사하기>을 하여 l에 배열의 시작인 left를 넣고 이를 배열의 완전 끝이기도한 right까지 for문을 돌려 list에 l을 index로 하여 sorted의 항목들을 다 복사시키는 걸로 하여 원래 리스트에 제대로 정렬하는 것까지 마무리하였습니다.

1.2

s1910783@oak:hw08$ gcc test-merge.c merge.c -o test-merge

s1910783@oak:hw08$ ./test-merge

|+ [0:27][1:10][2:12][3:20][4:25][5:13][6:15][7:22]

||+ [0:27][1:10][2:12][3:20]

|||+ [0:27][1:10]

||||+ [0:27]

||||- [0:27]

||||+ [1:10]

||||- [1:10]

|||- [0:10][1:27]

|||+ [2:12][3:20]

||||+ [2:12]

||||- [2:12]

||||+ [3:20]

||||- [3:20]

|||- [2:12][3:20]

||- [0:10][1:12][2:20][3:27]

||+ [4:25][5:13][6:15][7:22]

|||+ [4:25][5:13]

||||+ [4:25]

||||- [4:25]

||||+ [5:13]

||||- [5:13]

|||- [4:13][5:25]

|||+ [6:15][7:22]

||||+ [6:15]

||||- [6:15]

||||+ [7:22]

||||- [7:22]

|||- [6:15][7:22]

||- [4:13][5:15][6:22][7:25]

|- [0:10][1:12][2:13][3:15][4:20][5:22][6:25][7:27]

\*merge sort 과정 단계 설명(with 수행한 화면 & 슬라이드 12-16):

Merge\_sort는 말그대로 합병 정렬을 위한 전체 행동 틀이 짜여진 함수로, 크게 3개부분으로 나누어 1.왼쪽정렬recursive / 2.오른쪽정렬 recursive / 3.총정렬합병으로 기본 아이디어가 구축되어 있습니다. 일단 맨처음 배열을 정확히 반으로 갈라 왼쪽, 오른쪽으로 나누고, 그 다음 왼쪽과 오른쪽을 각각 merge\_sort를 recursive하게 돌아서 정렬되게 만들어, 간단하게 설명하자면 한쪽을 분할하고, 그걸 또 분할하고 각각 생긴걸 또 분할하고 그렇게 쭉가다가 분할의 최종으로 하나씩 남게되면 그때부터 차례차례recursive를 닫아가면서 다시 합쳐지면서 정렬이 되고 합병이 되는 것입니다. 이를 슬라이드 12-16에서 설명하자면,

스크린샷이(가) 표시된 사진

자동 생성된 설명

분할1단계에서는 첫 배열을 반 나누고, 분할2단계는 전단계에서 나눠져서 나온 각 배열들을 또 각각 반으로 나누고, 그렇게 해서 3단계까지 가서 딱 하나의 항목들이 되도록 분할을 해주고, 그 다음부터 각 분할된 것들을 합병해줍니다. 합병1단계에선 가장 자잘하게 분할되어 있는 것들, 그냥 각 항목 별로 분할되어 있으니 2개씩 다시 묶어주면 되고, 합병2단계에선 각 배열들마다 2개씩 가지고 있으니 각 왼쪽 부분 오른쪽 부분을 서로 비교하여 작은 것부터 순서대로 넣어 정렬시키고, 마지막 합병 3단계에서는 총 4개씩 배열이 되어 있으니 각각 비교하여 서로 파생되어 나온 것들을 결론적으로 정렬되서 합병된다는 것을 알 수 있습니다. 그다음 화면 출력의 각 줄의 의미에 대해 설명하자면,

|+ [0:27][1:10][2:12][3:20][4:25][5:13][6:15][7:22] //original

||+ [0:27][1:10][2:12][3:20] //L

|||+ [0:27][1:10] //L-L

||||+ [0:27] //L-L-L

||||- [0:27] //L-L-L합병

||||+ [1:10] //L-L-R

||||- [1:10] //L-L-R 합병

|||- [0:10][1:27] //L-L-R + L-L-L정렬 합병

|||+ [2:12][3:20] //L-R

||||+ [2:12] //L-R-L

||||- [2:12] //L-R-L 합병

||||+ [3:20] //L-R-R

||||- [3:20] //L-R-R 합병

|||- [2:12][3:20] //L-R-R + L-R-L 합병

||- [0:10][1:12][2:20][3:27] //L-L + L-R 합병

||+ [4:25][5:13][6:15][7:22] //R

|||+ [4:25][5:13] //R-L

||||+ [4:25] //R-L-L

||||- [4:25] //R-L-L 합병

||||+ [5:13] //R-L-R

||||- [5:13] //R-L-R 합병

|||- [4:13][5:25] //R-L-L + R-L-R 합병

|||+ [6:15][7:22] //R-R

||||+ [6:15] //R-R-L

||||- [6:15] //R-R-L 합병

||||+ [7:22] //R-R-R

||||- [7:22] //R-R-R 합병

|||- [6:15][7:22] //R-R-L + R-R-R 합병

||- [4:13][5:15][6:22][7:25] //R-L + R-R 합병

|- [0:10][1:12][2:13][3:15][4:20][5:22][6:25][7:27] //L + R 합병

옆에 주석에 달린 설명처럼 왼쪽 배열은 L, 오른쪽 배열은 R로 표현했고, 왼쪽 배열을 또 분할한 것의 왼쪽 배열은 L-L, 이걸 또 분할한 것의 왼쪽 배열을 L-L-L로 표현하고 합병할 때는 합병하는 두 개를 +로 표시하여 합병이라는 표시와 함께 표기하였습니다. 보라색은 전체적인 왼쪽 배열 부분, 즉 전체 배열에서의 왼쪽 부분에서 파생되어지는 부분, 파랑색은 전체 배열에서의 오른족 부분에서 파생되어지는 부분을 의미하고 마지막 빨간색은 최종적으로 합병되어 정렬되어진 결과를 의미합니다. 자세히 보면 ||와 같은 깊이 정렬 깊이와 함께 +는 분할, -는 합병이라는 것을 알 수 있습니다. 또한 이러한 출력 결과를 슬라이드 12-16과 연관시켜 본다면, +에서 깊이가 더해갈수록 분할 1->2->3단계로 더더욱 깊어가고, -에서 깊이가 더욱 낮아질수록 합병 1->2->3단계로 간다는 것을 알 수 있습니다.

2. Quick sort(퀵 정렬)

2.1

// ======================================================================

#include <stdio.h>

#include <stdlib.h>

#include "quick.h"

// ======================================================================

int partition(int list[], int left, int right)

{

int low, high, pivot, temp;

low = left; //1

high = right + 1; //2

pivot = list[left]; //3

do {

do //4

low++;

while(low<=right && list[low]<pivot);

do //5

high--;

while(high>=left && list[high]>pivot);

if(low<high) SWAP(list[low], list[high], temp); //6

} while(low<high); //7

SWAP(list[left], list[high], temp);

return high; //8

}

// ======================================================================

\*이렇게 프로그램한 이유:

이 함수는 이른바 퀵정렬 때 필요한 분할, int partition(int list[], int left, int right)함수로 간단하게 설명하자면 입력으로 리스트 배열 list[]와 들어온 배열 list[]의 가장 왼쪽 index를 가리키는 변수 left와 가장 오른쪽 index를 가리키는 변수 right가 들어옵니다. 이러한 값들이 들어오면 partition함수는 분할 정복 방법을 사용하는데, 리스트 배열 list[]에서 가장 왼쪽 항목, 즉 left가 가리키는 index의 “숫자”를 pivot이라는 기준으로 하여 이 pivot의 값보다 작은 항목들은 pivot의 왼쪽 부분에, pivot의 값보다 큰 항목들은 pivot의 오른쪽 부분에 넣어 2개의 부분 리스트로 비 균등 분할하는 함수입니다. 이러한 기능을 구현하기 위해서 저는 먼저 가장 왼쪽 index로써 차차 올라가며 pivot과의 비교를 위한 도구로 쓰일 변수 low와 동일한 의미로 가장 오른쪽 index보다 1큰 값의 index로 차차 내려가면서 pivot과의 비교를 위한 도구로 쓰일 변수 high, 그리고 각 단계에서 들어온 배열list[]에서 pivot값을 넣어줄 변수 pivot, 그리고 정렬이 다 끝나고 pivot이 제대로 된 자리를 찾아가기 위해 1개 오버된 index값 low와 바꿀 때 쓸 함수 SWAP에 사용할 변수 temp를 선언하였습니다. 그 다음엔 앞서 말한 것처럼 low에는 들어온 배열의 가장 왼쪽 index값인 left를 주어 차차 올라갈 준비를, high에는 가장 오른쪽 index값인 right에 1 더한 값을 주어 가장 오른쪽에서 차차 내려올 준비를 해주고, pivot에는 pivot으로 쓰일 가장 왼쪽 값인 list[left]값을 주어 pivot값 또한 준비해주었습니다. 그 다음은 반복문do~while문큰거 1개를 열어 전체적인 low와 high가 각각 왼쪽과 오른쪽을 의미하는 index들 임으로 low가 반드시 high보다 작다는 조건을 주어 이를 위반하였을 경우 더 이상 진행하지 못하고 하고, do~while문을 안에 2개 더 열어 하나는 low가 조건에 맞추어 1씩 증가되는데, 여기서 이 조건은 low가 이 배열을 벗어나지 않게 right까지만 증가하게 하고 low index값에 해당하는 리스트의 항목이 pivot보단 작을 경우에는 왼쪽에 정상적으로 놔두면 됨으로 넘어가서 pivot보다 클 경우에는 정지되서 나중에 조건에 안맞아서 멈췄을 때에 해당하는 high값과 SWAP되도록 하였고, 동일맥락으로 나머지 do~while문 하나에도 high가 차차 감소하게 되는데 조건으로 high가 left까지만 감소하게 하고 얘는 오른쪽 부분이니 pivot보다 더 큰 값일때만 반복문을 진행, 만약 pivot값보다 작은 값이 나올 경우 반복문이 멈추고 바로 밑에 독립적으로, 큰 do~while문 안에 있는 if-SWAP문을 이용해 미리 멈춰있던low 값과 바꾸어 주도록 하였습니다. 여기서 이 두개의 SWAP은 if문 조건 안에 두어 조건으로 반드시 low가 high보다 작을 경우, 즉 왼쪽과 오른쪽이 서로의 영역에 있을 경우에만 SWAP되도록 하였습니다. 그 다음 큰 do~while문을 빠져나오고 나서 최종 마무리로 분할이 끝났으면 pivot이 맨 앞에 있고 그 다음 왼쪽 부분, 오른쪽 부분이 되있으므로 서로의 영역을 침범하여 멈춰있는 high값을 pivot값과 SWAP하여 pivot값이 딱 중간에 오도록 하였습니다.

2.2

s1910783@oak:hw08$ gcc test-quick.c quick.c -o test-quick

s1910783@oak:hw08$ ./test-quick

|+ [0:5][1:3][2:8][3:4][4:9][5:1][6:6][7:2][8:7]

pivot [0:5]->[4:5]

||+ [0:1][1:3][2:2][3:4]

pivot [0:1]->[0:1]

|||+

|||-

|||+ [1:3][2:2][3:4]

pivot [1:3]->[2:3]

||||+ [1:2]

||||- [1:2]

||||+ [3:4]

||||- [3:4]

|||- [1:2][2:3][3:4]

||- [0:1][1:2][2:3][3:4]

||+ [5:9][6:6][7:8][8:7]

pivot [5:9]->[8:9]

|||+ [5:7][6:6][7:8]

pivot [5:7]->[6:7]

||||+ [5:6]

||||- [5:6]

||||+ [7:8]

||||- [7:8]

|||- [5:6][6:7][7:8]

|||+

|||-

||- [5:6][6:7][7:8][8:9]

|- [0:1][1:2][2:3][3:4][4:5][5:6][6:7][7:8][8:9]

\*quick sort과정 각 단계 설명(with 수행한 화면 & 슬라이드12-26):

먼저 간단하게 quick sort의 구조에 대해 설명해드리자면, 먼저 임의의 변수 q에 partition수행 후 pivot의 index을 저장하여 이를 다시 왼쪽(의 끝이자 right가 되는 q-1), 오른쪽(의 첫번째 이자 left가 되는 q+1)으로 거듭해나가서 또 정렬하도록 하는 구조입니다. 이를 슬라이드 12-26과 함께 설명하자면,

스크린샷이(가) 표시된 사진

자동 생성된 설명

첫번째 전체 배열에서 가장 왼쪽 값인 5를 pivot으로 하여 그 다음 라인에 pivot보다 작은 왼쪽 부분과 pivot, pivot보다 큰 오른쪽 부분으로 정렬이 되고 그 다음 또 왼쪽 부분과 오른쪽 부분이 각각 퀵정렬이 되어 1, 9가 각각 pivot이 되어 그 다음 3번째 라인에 정렬이 되고, 또 그렇게 각 부분에서 나눠진 왼쪽, 오른쪽 부분에서 또 각각 pivot이 선정되어 정렬이 되고, 그 다음은 정렬이 다 되었고 각 배열의 구성이 하나씩이니 그냥 바로 다음 줄에 정렬이 완성되는 것을 볼 수 있습니다.

이제 수행 화면 출력 결과에 대해서 살펴보자면,

|+ [0:5][1:3][2:8][3:4][4:9][5:1][6:6][7:2][8:7] //정렬 안된 레어 상태

pivot [0:5]->[4:5] //pivot이 중간 위치로 찾아감.

||+ [0:1][1:3][2:2][3:4] //첫 퀵정렬 후 생겨진 왼쪽 부분 배열

pivot [0:1]->[0:1] //pivot이 제 위치를 찾아감.

|||+

|||- //pivot인 1보다 큰 것들이 그대로 되어 있으므로 아무것도 변화 x.

|||+ [1:3][2:2][3:4] //퀵 정렬 후 또 생긴 왼쪽 배열

pivot [1:3]->[2:3] //위의 왼쪽 배열에서 또 pivot이 중간으로 찾아감.

||||+ [1:2] //정렬된 왼쪽

||||- [1:2] //순서 맞으니 그대로 결과

||||+ [3:4] //정렬된 거 오른쪽

||||- [3:4] //수서 맞으니 그대로 결과

|||- [1:2][2:3][3:4] //최종적으로 계층적으로 올라가서 정렬 완료되었음.

||- [0:1][1:2][2:3][3:4] //최종적으로 계층적으로 올라가서 정렬 완료되었음.

||+ [5:9][6:6][7:8][8:7] //첫 퀵 정렬 후 생겨진 오른쪽 부분 배열

pivot [5:9]->[8:9] //pivot이 제 위치를 찾아감.

|||+ [5:7][6:6][7:8] //pivot이 제 위치를 찾아감으로 생긴 왼쪽 배열

pivot [5:7]->[6:7] //또 pivot이 제 자리를 찾아감.

||||+ [5:6] //정렬 후 왼쪽

||||- [5:6] //정렬 할께 없으니 그대로 끝

||||+ [7:8] //정렬 후 오른쪽

||||- [7:8] //정렬 할께 없으니 그대로 끝

|||- [5:6][6:7][7:8] //위 상위 배열 정렬 끝

|||+

|||-

||- [5:6][6:7][7:8][8:9] //첫 정렬 후 나온 오른쪽 배열 정렬 끝

|- [0:1][1:2][2:3][3:4][4:5][5:6][6:7][7:8][8:9] //최종 정렬 끝

위 결과를 슬라이드 12-26의 각 라인별 색상에 맞추어 본 결과 왼쪽 쭉 가서 정렬 끝내고 그 다음 오른쪽 쭉 가서 정렬 끝내는 과정이 동일함을 알 수 있다.

끝.