5주차 [과제]

이름 : 유재영

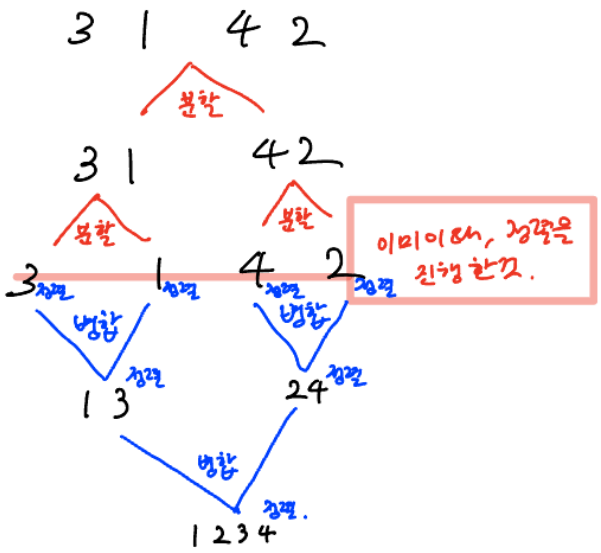
학번 : 20181650

제출일 : 2023.04.10

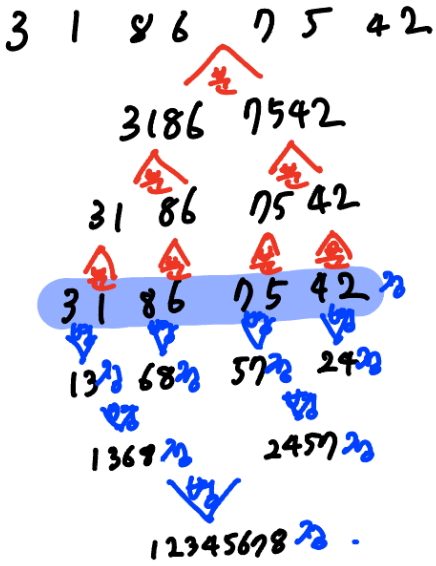
담당교수 : 최해철 교수님

과제 1. 병합정렬 구현 (내림 차순)

병합정렬은 전형적인 분할-정복 알고리즘으로써 아래와 같은 순서로 진행된다.



좀 더 많은 데이터에 대해서는 아래와 같이 진행된다.



이때, 데이터가 각각 하나씩으로 나누어졌을 때 정렬 후 병합 과정이 진행되는 것이므로, 각각 하나씩 나누어진 과정에서도 **정렬은 진행된 것**이다.

위 과정을 바탕으로 작성한 코드를 하나씩 분석을 진행하면,

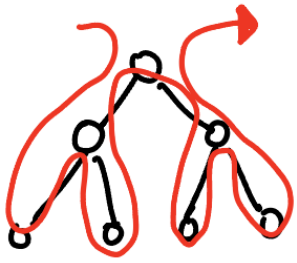
```
80  void mergeSort(int data[], int left, int right) {
81
82      if (left < right) {
83          int middle = (left + right) / 2;
84          mergeSort(data, left,  right: middle);
85          mergeSort(data,  left: middle + 1, right);
86          merge(data, left, middle, right);
87      }
88  }
```

83행은 `data[]`를 `middle = (left + right) / 2;`을 기준점으로 반으로 나누어

84,85행 `mergeSort()`를 **<u>두 번 재귀 호출</u>**하게 된다.

진행 순서는 `mergeSort()`를 끝까지
진행 후, 다시 재귀적으로 돌아와 다음
`mergeSort()`를 실행하고,
최종적으로 `merge()`를 호출하게 된다.

<u>그림과 같이 진행 됨 →</u>



호출되는 `merge()`를 분석 해보면,

```
52  void merge(int data[], int left, int middle, int right) { // 내림차순 정렬
53      int *tmp = (int *) malloc( size: sizeof(int) * (right - left + 1));
54      int i, j, k, l;
55      i = left;
56      j = middle + 1;
57      k = left;
58      while (i <= middle && j <= right) {
59          if (data[i] >= data[j]) {
60              tmp[k++] = data[i++];
61          } else {
62              tmp[k++] = data[j++];
63          }
64      }
65      if (i > middle) {
66          for (l = j; l <= right; l++) {
67              tmp[k++] = data[l];
68          }
69      } else {
70          for (l = i; l <= middle; l++) {
71              tmp[k++] = data[l];
72          }
73      }
74      for (l = left; l <= right; l++) {
75          data[l] = tmp[l];
76      }
77      free(tmp);
78  }
```

53행은 merge가 진행되는 사이즈 만큼 **tmp 배열에 동적 할당**을 통해 공간을 할당 해주게 되며

58행 while문에서는 middle값을 기준으로 좌우로 나누어 비교며,
i 가 middle에 도착하거나 혹은 **j가 right에 도착할 때** 까지 동적 할당된 tmp 배열을 채워가게 된다.

65행~73행 if 문에서는, **남아있는 값**을 tmp 배열에 채워나가게 된다.

74행 for문에서는 `data[]` 배열에 `tmp[]` 배열의 내용을 복사 진행하며

77행에서 동적 할당 해제를 선언한다.

실행 결과는 아래와 같다.

```
Run:  230404
/Users/yujaeyeong/Developments/algorithm/uni
랜덤 수의 개수를 입력하시오. : 11
정렬 전 배열 : 12 97 44 47 71 8 14 71 46 24 27
정렬 후 배열 : 97 71 71 47 46 44 27 24 14 12 8
Process finished with exit code 0
```

오름 **차순으로 구현**하려면 **59행** if문을 `if (data[i] <= data[j])` 로 수정하면 된다.

```
while (i <= middle && j <= right) {
    if (data[i] <= data[j]) {
        tmp[k++] = data[i++];
    } else {
        tmp[k++] = data[j++];
    }
}
```

실행 결과는 아래와 같다.

```
Run:  230404
/Users/yujaeyeong/Developments/algorithm/uni
랜덤 수의 개수를 입력하시오. : 11
정렬 전 배열 : 16 35 50 89 53 10 74 86 8 43 32
정렬 후 배열 : 8 10 16 32 35 43 50 53 74 86 89
Process finished with exit code 0
```

합병 정렬 실습과 과제 진행 간, 의문점이 하나 생겼다.

`mergeSort()` 학습 간 참고 자료로 2학년때 학습했던 자료구조 책을 사용했는데,

`merge()` 부분 메소드의 내부 변수에서 이상한 점을 발견했다.

```
void merge(int data[], int left, int middle, int right) {
    int *tmp = (int *) malloc(sizeof(int) * (right - left + 1));

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

    ...

    for (l = left; l <= right; l++) {
        data[l] = tmp[l];
    }
    free(tmp);
}
```

해당 k = left 부분인데, 의문을 가지게 된 이유는,

배열에 할당된 크기가 '2'라고 한다면 접근할 수 있는 인덱스 번호는 각각 0,1 일 것인데, 넘어서는 값 2,3 등으로 접근해도 오류가 발생하지 않는 이유가 무엇일까?

2,3으로 접근?? 에 대한 설명을 위해, 다음과 같이 **printf**문을 추가적으로 작성하였다.

```
void merge(int data[], int left, int middle, int right) {

    int *tmp = (int *) malloc(sizeof(int) * (right - left + 1));

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

    printf("merge 호출 횟수 : %d\n", ++count);
    printf("할당된 공간의 크기 : %d\n", right - left + 1);

    while (i <= middle && j <= right) {
        printf("비교 대상 값 1번 : %d 2번 : %d\n", data[i], data[j]);
        if (data[i] <= data[j]) {
            printf("1번 출력 / 접근하는 인덱스 번호 : %d\n", k);
            tmp[k++] = data[i++];
            printf("2번 출력 / 접근하는 인덱스 번호의 저장 후 값 : %d\n",tmp[k - 1]);
        } else {
            printf("3번 출력 / 접근하는 인덱스 번호 : %d\n", k);
            tmp[k++] = data[j++];
            printf("4번 출력 / 접근하는 인덱스 번호의 저장 후 값 : %d\n",tmp[k - 1]);
```

```
        }
    }

    if (i > middle) {
        for (l = j; l <= right; l++) {
            tmp[k++] = data[l];
        }
    } else {
        for (l = i; l <= middle; l++) {
            tmp[k++] = data[l];
        }
    }

    for (l = left; l <= right; l++) {
        printf("tmp[l]의 값 : %d\n", tmp[l]);
        data[l] = tmp[l];
    }
    printf("\n");
    free(tmp);
}
```

아래는 n=8로 호출 했을 때 위 함수로 출력되는 결과이다.

```
merge 호출 횟수 : 2
할당된 공간의 크기 : 2
비교 대상 값 1번 : 64 2번 : 43
3번 출력 / 접근하는 인덱스 번호 : 2
4번 출력 / 접근하는 인덱스 번호의 저장 후 값 : 43
tmp[l]의 값 : 43
tmp[l]의 값 : 64
```

```
merge 호출 횟수 : 5
할당된 공간의 크기 : 2
비교 대상 값 1번 : 13 2번 : 85
1번 출력 / 접근하는 인덱스 번호 : 6
2번 출력 / 접근하는 인덱스 번호의 저장 후 값 : 13
tmp[l]의 값 : 13
tmp[l]의 값 : 85
```

둘 다 접근하는 인덱스 번호가, 할당된 공간의 크기를 넘어선다.

그런데도 정상적으로 작동하는 이유가 무엇인지 고민해보아도 찾아낼 수가 없었다.

교수님께 질문을 드린 후, 해당 과제를 수정해볼 예정이다.

교수님과의 결론 : ~~Mac북이 메모리 관리를 잘하는것이다.~~

교수님께서, **다른 프로그램 여러개를 돌린 뒤에 실행시키면 메모리 참조 오류가 날 것**이라고 하셨다.

해당 방식으로 코드를 작성하게 되면, 치명적인 오류를 발생시키는 것이므로,

그래서 문제가되는 해당 부분을 아래와 같이 수정하였다.

```
52  void merge(int data[], int left, int middle, int right) {
53
54      int *tmp = (int *) malloc( size: sizeof(int) * (right - left + 1));
55
56      int i, j, k, l;
57      i = left;
58      j = middle + 1;
59      k = 0;
60      while (i <= middle && j <= right) {
61          if (data[i] >= data[j]) {
62              tmp[k++] = data[i++];
63          } else {
64              tmp[k++] = data[j++];
65          }
66      }
67      if (i > middle) {
68          for (l = j; l <= right; l++) {
69              tmp[k++] = data[l];
70          }
71      } else {
72          for (l = i; l <= middle; l++) {
73              tmp[k++] = data[l];
74          }
75      }
76      for (l = left; l <= right; l++) {
77          data[l] = tmp[l-left];
78      }
79      free(tmp);
80  }
```

59행에서 `k = 0` 으로 시작하여 `tmp[]` 배열에 저장을 진행하며,

76행에서 최종적으로 `data[]` 에 저장하는 과정을 `tmp[0]` 부터 접근할 수 있도록 `tmp[l-left]` 로 설정해주어 해결하였다.

과제2. 힙정렬 구현(오름 차순)

먼저 힙은 **완전 이진 트리**로서 다음의 성질을 만족한다.

- 각 노드의 값은 자신의 children의 값보다 크지 않음(작거나 같음) → 최소 힙
- 각 노드의 값은 자신의 children의 값보다 작지 않음(크거나 같음) → 최대 힙
- 맨 아래 층을 제외하고는 완전히 채워져 있음
- 맨 아래 층은 왼쪽부터 꽉 채워져 있음

힙 정렬은 주어진 배열을 힙으로 만든 다음, 차례로 하나씩 힙에서 제거함으로써 정렬하게 된다.

책에서 구현하려고 하는 힙 정렬 과정을 나열하면

1. 주어진 배열을 Heap으로 만듬
2. Heap에서 가장 작은 값을 차례로 하나씩 Heap에서 제거함으로써 Heap의 크기를 줄임
3. Heap에 원소가 남아 있으면 goto 1번
4. Heap에 아무 원소가 남지 않으면 Heap 정렬 종료

   → 정렬 순서는 Heap에서 제거된 순서

책에서 요구하는 과정은 아래와 같다.

1. 주어진 배열을 Heap으로 만드는 과정 → `buildHeap()`
2. Heap에서 최소/최대 원소를 제거하고 나서 Heap의 성질을 만족하게 수선하는 과정 → `Heapify()`

해당 부분을 구현하는데 있어서 햇갈리는 부분과 어려움이 많았지만 4시간 가량 부딪혀 정답을 찾게되었다.

과제로 요구한 오름차순을 구현하기 위해서는, **먼저 힙을 최대 힙으로 구현한 뒤, 힙 정렬 과정을 거쳐야 했다.**

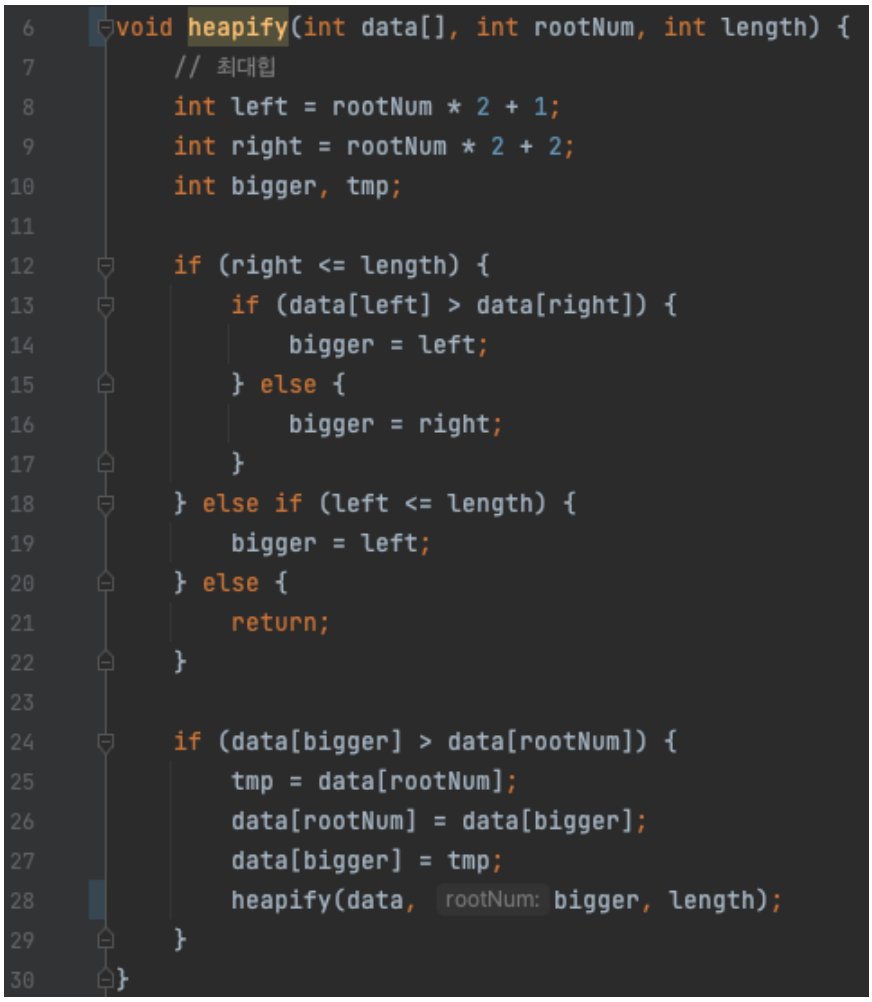
힙 정렬간 root에 있는 값과 배열의 마지막에 있는 값의 swap을 진행 한 뒤 해당 마지막 값은 제외하고 다시 heap 정렬을 진행하도록 설계되어 있으므로, **최대 힙으로 구현하는 것이**

정답이었다.

또한, **배열 인덱스 번호는 0번 부터 시작**인데, 책에서는 번호를 1번에서 시작하여 해결하는 과정으로 해결하여,
고민이 필요했다.

구현한 코드는 아래와 같다.

`heapify()` - 최대 힙



```
6   void heapify(int data[], int rootNum, int length) {
7       // 최대힙
8       int left = rootNum * 2 + 1;
9       int right = rootNum * 2 + 2;
10      int bigger, tmp;
11
12      if (right <= length) {
13          if (data[left] > data[right]) {
14              bigger = left;
15          } else {
16              bigger = right;
17          }
18      } else if (left <= length) {
19          bigger = left;
20      } else {
21          return;
22      }
23
24      if (data[bigger] > data[rootNum]) {
25          tmp = data[rootNum];
26          data[rootNum] = data[bigger];
27          data[bigger] = tmp;
28          heapify(data, rootNum: bigger, length);
29      }
30  }
```

구현 과정에서 루트의 인덱스 번호를 0으로 설정하고 진행하였기 때문에,

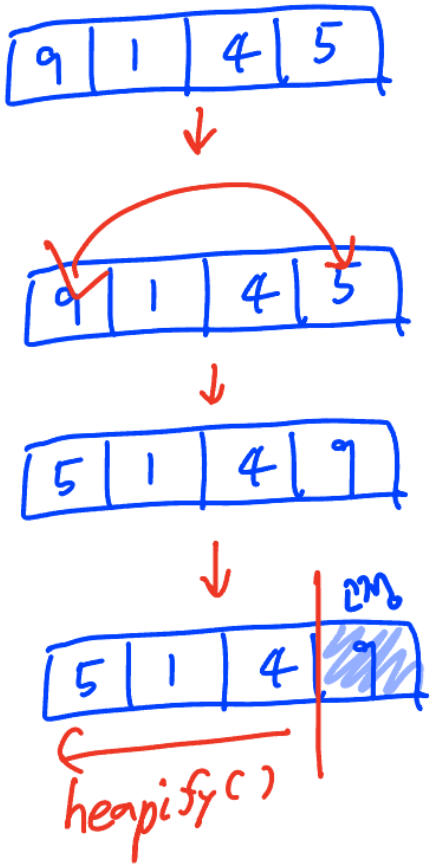
8,9행 에서 자식 노드를 접근할 때 `rootNum * 2 + 1` / `rootNum * 2 + 2` 로 접근하게 된다.

12~22행 에서 좌측과 우측 자식 노드가 존재하는지를 **length**와 비교하여 판단하고, 큰 값의 인덱스 번호를 bigger에 저장하게 된다. 만약 자식 노드가 존재하지 않을 경우 **21행**의 `return;` 을 통해서 탈출하게 된다.

24행은 `data[rootNum]`보다 `data[bigger]` 이 큰 값인 경우 서로 교체 과정이 **25~27행**에 구현 되었고,

28행은 힙을 만족하는지 확인하기 위해 교체된 배열 인덱스 번호를 rootNum으로 `heapify()`가 호출되었다.

24행~29행은 아래와 그림과 같은 역할을 하는 부분이다.



학습간 추가 적으로 구현해본 최소 힙은 아래와 같다.

```java
void heapify(int data[], int rootNum, int length) {
    //최소 힙
    int left = rootNum * 2 + 1;
    int right = rootNum * 2 + 2;
    int smaller, tmp;

    if (right <= length) {
        if (data[left] < data[right]) {
            smaller = left;
        } else {
            smaller = right;
        }
    } else if (left <= length) {
        smaller = left;
    } else {
        return;
    }

    if (data[smaller] < data[rootNum]) {
        tmp = data[rootNum];
        data[rootNum] = data[smaller];
        data[smaller] = tmp;
        heapify(data, rootNum: smaller, length);
    }
}
```

크게 달라지는 부분은 없이, **13행과 24행에서 비교 연산자의 방향이 바뀌게 된다.**

해당 최소힙을 활용해 내림차순으로 정렬을 진행할 수 있다.

주어진 배열을 먼저 힙 형태로 만드는 `buildHeap()` 은 아래와 같다.

```java
void buildHeap(int data[], int length) {
    for (int i = (length - 1) / 2; i >= 0; i--) {
        heapify(data, rootNum: i, length);
    }
}
```

위 과정에서는, **힙 조건만 만족시키면 되므로** 말단 노드들을 제외한 노드부터 `heapify()`를 진행 시키게 된다.

최종적인 힙 정렬 `heapSort()` 는 아래와 같다.

`heapSort()` - 힙 정렬 과정

```
73  void heapSort(int data[], int length) {
74
75      int tmp;
76      buildHeap(data, length);
77
78      for (int i = length; i >= 1; i--) {
79          tmp = data[i];
80          data[i] = data[0];
81          data[0] = tmp;
82          heapify(data, rootNum: 0, length: i - 1);
83      }
84  }
```

78행 for문 내부에서 `heapify()`를 호출하게 되는데, 이때 맨 마지막 배열 인덱스 번호에 최대 힙의 최대 값인
루트 노드를 저장하고, 마지막 배열 인덱스 번호를 제외하고 `heapify()`를 진행하게 된다.

최종적으로 오름 차순으로 정렬이 진행되는 것이다.

실행 결과는 아래와 같다.

```
Run:  230404
/Users/yujaeyeong/Developments/algorithm/univ
랜덤 수의 개수를 입력하시오. : 11
정렬 전 배열 : 91 61 73 39 26 37 26 81 81 72 97
정렬 후 배열 : 26 26 37 39 61 72 73 81 81 91 97
Process finished with exit code 0
```