알고리즘 과제(03)

201204025 김대래

과목 명 : 알고리즘
담당 교수 : 김동일
분반 : 02 분반

Index

1. 실습 환경

OS : MacOs Mojave 10.14
Language : C
Tool : Neovim, gcc

2. 과제 개요

임의의 나열된 수가 저장된 파일(test_'test case'.txt)을 입력 받아 각각의 정렬 방법(Max Heap, Min Heap, Counting)에 따라 정렬하여 결과를 파일로 출력한다.

입력 예시 : test_100.txt / test_1000.txt

출력 예시 : max_heap.txt / min_heap.txt / counting.txt

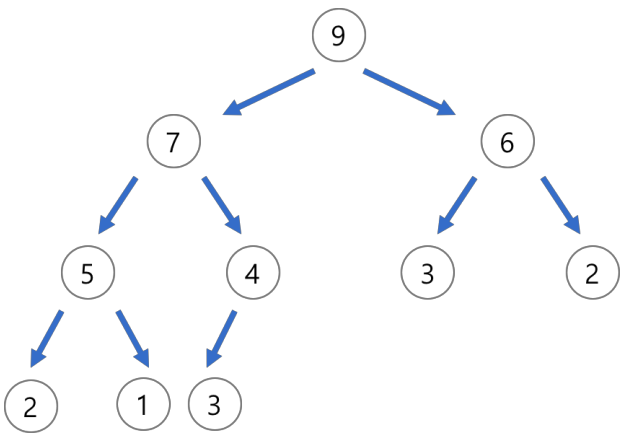
3. 주요 개념

3.1. 힙 정렬(Heap Sort)

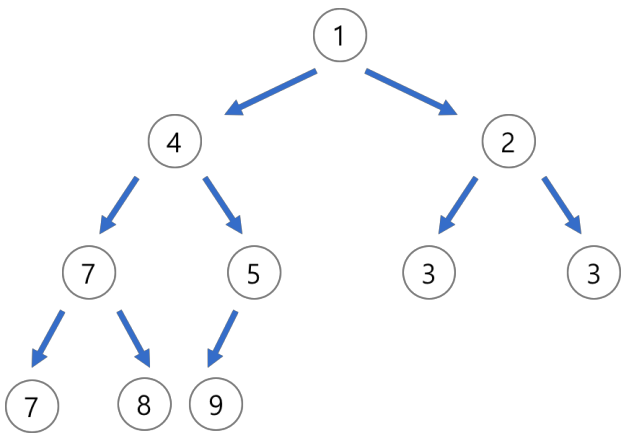
3.1.1. 개념

힙 정렬(Heapsort)이란 최대 힙 트리(Max Heap)나 최소 힙 트리(Min Heap)를 구성해 정렬을 하는 방법으로서, 내림차순 정렬을 위해서는 최대 힙을 구성하고 오름차순 정렬을 위해서는 최소 힙을 구성하면 된다.

트리의 높이에 의존적이기 때문에 heapify의 복잡도는 $O(logn)$이며 n개의 데이터를 정렬할 때 시간 복잡도는 $O(nlogn)$이다.



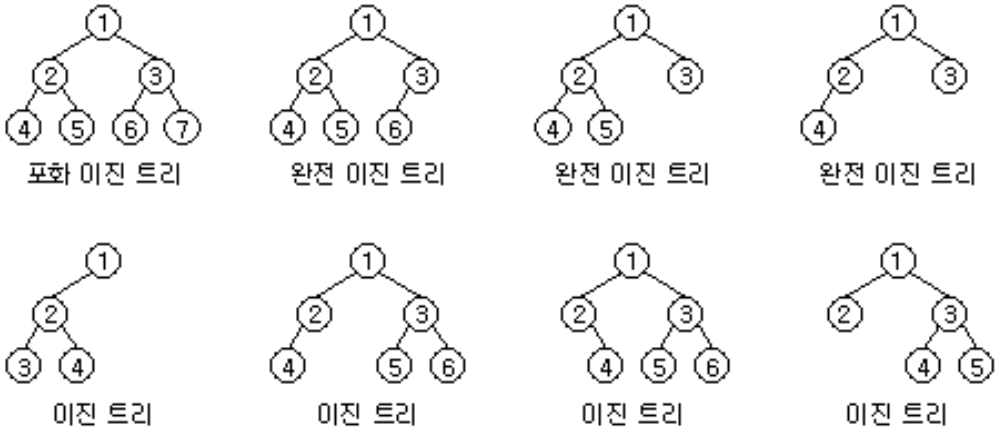
-최대 힙(max heap)-



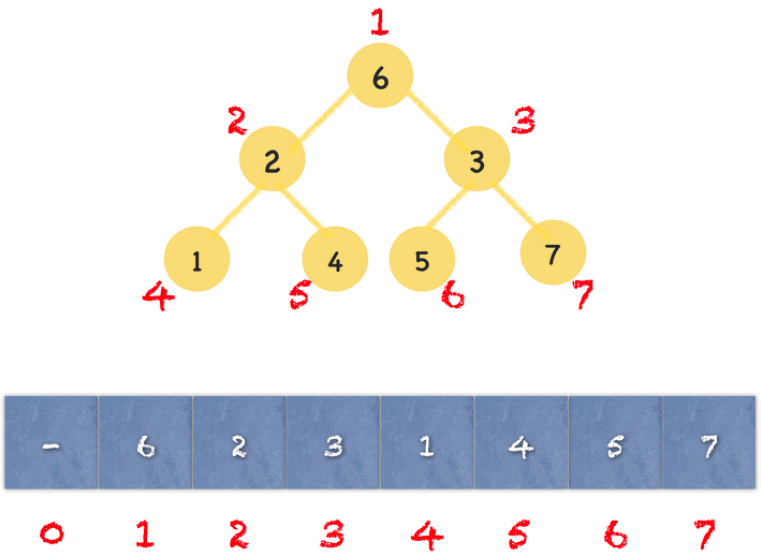
-최소 힙(min heap)-

3.1.2. 힙 정렬 조건

힙 정렬을 하기 위해서는 반드시 완전 이진 트리의 구조를 갖추고 있어야 한다.



완전 이진 트리의 구조를 갖추고 있기 때문에 배열의 index를 이용하여 트리 구조로 표현할 수 있다.



3.1.3. 정렬 순서

힙 정렬을 하기 위하여 먼저 **(1)최대/최소 힙으로 만들어(BuildHeap)** 줘야 한다. 최대 힙의 경우 부모가 자식보다 값이 커야하며, 최소 힙의 경우 반대로 부모가 자식보다 값이 작아야한다.
만약, 그렇지 않다면 **(2)힙의 구조로 데이터를 변경(Heapify)**한다.
힙의 구조를 갖췄다면 **(3)루트 노드(index = 0)를 배열의 맨 뒤로 보내고** (2)으로 돌아가서 반복한다.

*(3)을 반복할 때는 뒤에서 부터 차례대로 값을 교환한다.

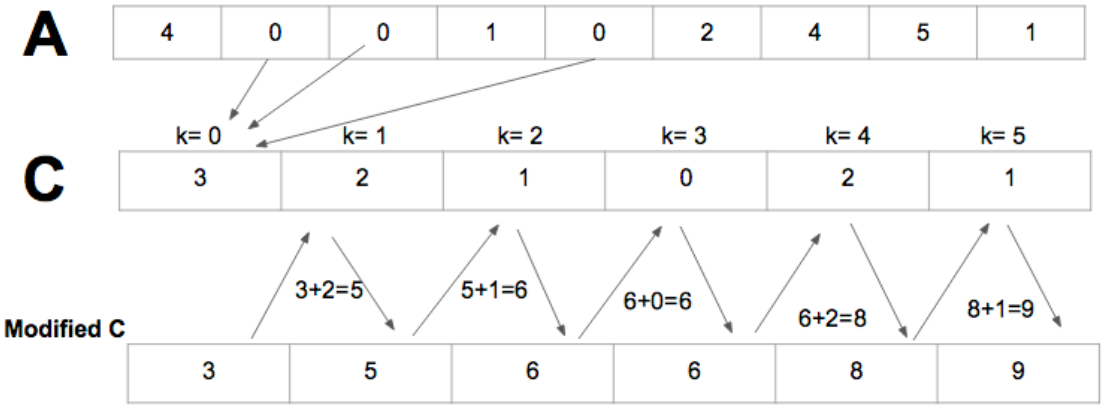
3.2. 계수 정렬(Counting Sort)

3.2.1. 개념

계수 정렬은 배열의 각 숫자가 몇 번 등장했는지 횟수를 이용하여 정렬하는 방식이다.

각 숫자의 누적합을 이용하여 해당 번호가 몇 번째 index에 위치해야 하는지 알 수 있다.

배열의 크기인 n만큼 반복하며 횟수를 측정하여 배열의 최댓값 상수 k까지 반복하며 정렬하기 때문에 $O(k+n)$의 복잡도로 상수 시간인 k를 빼면 $O(n)$의 복잡도를 갖는다.



3.2.2. 계수 정렬의 한계

배열의 크기 n보다 배열이 원소 값이 월등히 클 경우 (ex. n= 10 k = 3213) $O(n^2)$보다 큰 경우가 발생할 가능성이 있다.

```
                COUNTING-SORT(A, B, k)
Θ(k)    1  for i = 0 to k
        2      C[i] = 0
Θ(n)    3  for j = 1 to A.length
        4  C[A[j]] = C[A[j]] + 1
        5  ▷ C[i] contains the number of elements equal to i.
Θ(k)    6  for i = 1 to k
        7      C[i] = C[i] + C[i - 1]
        8  ▷ C[i] contains the number of elements less than or equal to i.
Θ(n)    9  for j = A.length downto 1
       10      B[C[A[j]]] = A[j]
       11      C[A[j]] = C[A[j]] - 1
```

4. 과제 3-1 Max Heap Sort

4.1. MaxHeapSort(int arr[])

```
void MaxHeapSort(int arr[]) {
    int heap_size = test_num;
    BuildMaxHeap(arr);
    int i;
    for (i = test_num - 1; i > 0; i--) {
        swap(arr, 0, i);
        heap_size--;
        MaxHeapify(arr, heap_size, 0);
    }
}
```

I. Max Heap 구조로 만들기 위해 BuildMaxHeap()함수 사용
II. heap의 구조를 갖췄다면 root 노드(index 0)를 정렬이 안 된 가장 뒤의 원소와 교환
III. MaxHeapify()함수를 통해 다시 heap의 구조로 전환

4.2. BuildMaxHeap(int arr[])

```
void BuildMaxHeap(int arr[]) {
    int heap_size = test_num;
    int i;
    for (i = (test_num / 2) - 1; i >= 0; i--) {
        MaxHeapify(arr, heap_size, i);
    }
}
```

I. 처음 배열을 Max Heap의 구조로 만들기 위한 함수
II. leaf 노드의 부모부터 순서대로 MaxHeapify()함수를 통해 heap 구조로 전환

4.3. MaxHeapify(int arr[], int heap_size, int i)

```
void MaxHeapify(int arr[], int heap_size, int i) {
    int largest = i;
    int left = Left(i);
    int right = Right(i);
    if (left < heap_size && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < heap_size && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(arr, i, largest);
        MaxHeapify(arr, heap_size, largest);
    }
}
```

I. 부모 노드 i 를 가장 큰 값으로 설정한 뒤 자녀(left, right)와의 값을 비교하며 큰 값을 찾아 저장
II. 만약 부모가 큰 값이 아니라면 자리를 바꾸고 다시 heapify를 통해 힙의 구조를 완성하며 반복(재귀)

5. 과제 3-2 Min Heap Sort

5.1. MinHeapSort(int arr[]) / BuildMinHeap(int arr[])

```
void MinHeapSort(int arr[]) {
    int heap_size = test_num;
    BuildMinHeap(arr);
    int i;
    for (i = test_num - 1; i > 0; i--) {
        swap(arr, 0, i);
        heap_size--;
        MinHeapify(arr, heap_size, 0);
    }
}

void BuildMinHeap(int arr[]) {
    int heap_size = test_num;
    int i;
    for (i = (test_num / 2) - 1; i >= 0; i--) {
        MinHeapify(arr, heap_size, i);
    }
}
```

- Max Heap과 동일한 과정
- 실질적으로 정렬을 하는 것은 heapify를 통해 가장 작은 값을 root 노드로 만들어 Min heap 구조를 갖춤

5.2. MinHeapify(int arr[], int heap_size, int i)

```
void MinHeapify(int arr[], int heap_size, int i) {
    int smallest = i;
    int left = Left(i);
    int right = Right(i);
    if (left < heap_size && arr[left] < arr[smallest]) {
        smallest = left;
    }
    if (right < heap_size && arr[right] < arr[smallest]) {
        smallest = right;
    }
    if (smallest != i) {
        swap(arr, i, smallest);
        MinHeapify(arr, heap_size, smallest);
    }
}
```

I. 전반적으로 Max Heapify와 동일한 구조를 갖고 있지만 Min Heap의 구조를 갖추기 위하여 최솟값을 찾아 저장

6. 과제 3-3 Counting Sort

6.1. Max(int arr[])

```
int Max(int arr[]) {
    int arrSize = test_num;
    int i;
    int max = arr[0];
    for (i = 1; i < arrSize; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    return max;
}
```

A. count 배열의 크기를 정하기 위해 실행하는 함수
B. 배열의 값 중 최댓값을 찾아 반환

6.2. CountingSort(int arr[])

```
void CountingSort(int arr[]) {
    //array init
    int arrSize = test_num;
    int max_val = Max(arr);
    int *sortedArr = (int *)malloc(sizeof(int) * arrSize);
    int *countingArr = (int *)malloc(sizeof(int) * (max_val + 1));

    int i, j;
    for (i = 0; i <= max_val; i++) {
        countingArr[i] = 0;
    }
    //각 원소의 수
    for (j = 0; j < arrSize; j++) {
        countingArr[arr[j]]++;
    }

    //누적 합
    for (i = 1; i <= max_val; i++) {
        countingArr[i] = countingArr[i] + countingArr[i - 1];
    }

    for (j = arrSize-1; j >= 0; j--) {
        sortedArr[countingArr[arr[j]]] = arr[j];
        countingArr[arr[j]]--;
    }

    //copy sotred array to original array
    for (i = 1; i <= arrSize; i++) {
        arr[i-1] = sortedArr[i];
    }
}
```

I. 정렬할 배열 arr를 인자로 받아 정렬하여 저장 후 값을 다시 copy해 올 sortedArr 선언
II. 배열의 가장 큰 값을 크기로 하는 countingArr 배열 선언
III. 원소가 하나도 없을 수 있기 때문에 counting Arr 0으로 초기화
IV. arr배열의 값을 index로 가지는 countingArr의 값을 1씩 올려주며 횟수를 측정
V. countingArr의 처음부터 끝까지 각 수의 누적합을 계산
VI. 정렬할 배열 arr의 마지막 원소부터 누적합을 보고 해당 index에 정렬 배열에 저장
VII. 모두 정렬된 배열 sortedArr를 arr에 복사

7. 실행

7.1. 정렬 결과 확인

7.1.1. Max Heap Sort

```
~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 100
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 0
Max Heap Sort Time : 0.000038

~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 cat max_heap.txt
0
0
7
9
10
12
13
14
18
20
27
```

7.1.2. Min Heap Sort

```
~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 ls
[Al]실 습 4주 차                    test_100.txt
hw03.c                              test_1000.txt
test

~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 100
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 1
Min Heap Sort Time : 0.000029

~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 cat min_heap.txt
199
195
192
186
185
```

7.1.3. Counting Sort

```
~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 100
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 2
Counting Sort Time : 0.000011

~/Desktop/Algorithm/[Al]03_C_201204025
→  [Al]03_C_201204025 cat counting.txt
0
0
7
9
10
12
13
14
18
20
27
```

8. 결과

8.1. 알고리즘 속도 비교

```
~/Desktop/Algorithm/[Al]03_C_201204025
➜  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 1000
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 0
How many times Heapify : 9552
Max Heap Sort Time : 0.000298

~/Desktop/Algorithm/[Al]03_C_201204025
➜  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 1000
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 1
How many times Heapify : 9584
Min Heap Sort Time : 0.000392

~/Desktop/Algorithm/[Al]03_C_201204025
➜  [Al]03_C_201204025 ./test
>> test case num (100 / 1000 / 10000)
>> Input : 1000
>> Input Sort Mode
 [0 : Max Heap Sort, 1 : Min Heap Sort, 2 : Conuting Sort ] : 2
Counting Sort Time : 0.000053
```

- 동일한 파일(test case : 1000)로 정렬
- 같은 복잡도($O(nlogn)$)를 갖는 Max Heap과 Min Heap은 비슷한 결과
- Max Heap이 조금 더 빠르게 나온 이유는 heapify 횟수를 측정해 본 결과 Min Heap보다 상대적으로 적은 횟수를 실행하기 때문
- 가장 빠른 Counting Sort는 $O(k+n)$의 복잡도를 갖는데 실행 조건에서의 k가 n보다 크지만 월등히 크지 않기에 $O(n)$의 복잡도를 가짐