Heap, Priority Queue, Fenwik tree, Dijkstra's Algorithm 에 대한 조사

V2018114 임철호

Priority Queue(우선순위 큐)

입력된 순서와 상관없이 우선순위에 따라 데이터를 꺼내는 자료구조

우선순위 큐를 구현 방법
- 배열
- 연결 리스트
- 힙

	삭제 순서
스택	후입선출 (LIFO)
큐	선입선출 (FIFO)
우선순위 큐	우선순위가 가장 높은 데이터

배열
-데이터 삽입 및 삭제과정에서 데이터를 한 칸씩 당기거나 밀어야 하는 연산을 계속 하여야 한다.

연결 리스트
-삽입의 위치를 찾기 위해 첫번째 노드에서부터 시작해 마지막 노드에 저장된 데이터와 우선순위를 비교 진행해야한다.

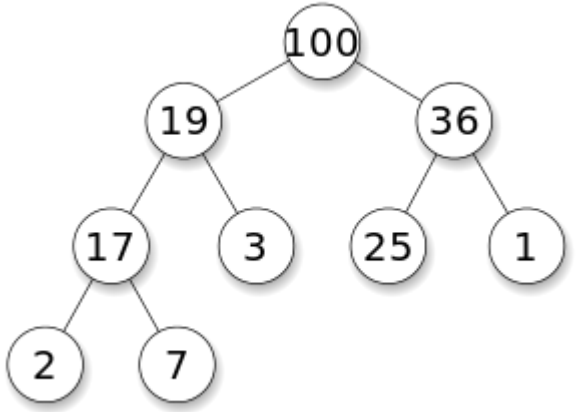
라는 단점이 있기에 우선순위 큐 사용시 힙을 주로 사용한다.

HEAP

- Heap은 최댓값 및 최솟값을 찾아내는 연산을 빠르게 하기 위해 고안된 완전이진트리(Complete binary tree)를 기본으로 한 자료구조

- 최대힙 – 부모 노드가 자식 노드 보다 크거나 같은 것
- 최소힙 – 부모 노드가 자식 노드 보다 작거나 같은 것

- 부모는 자식보다 우선순위가 높은 데이터가 배치된다.



최대 힙

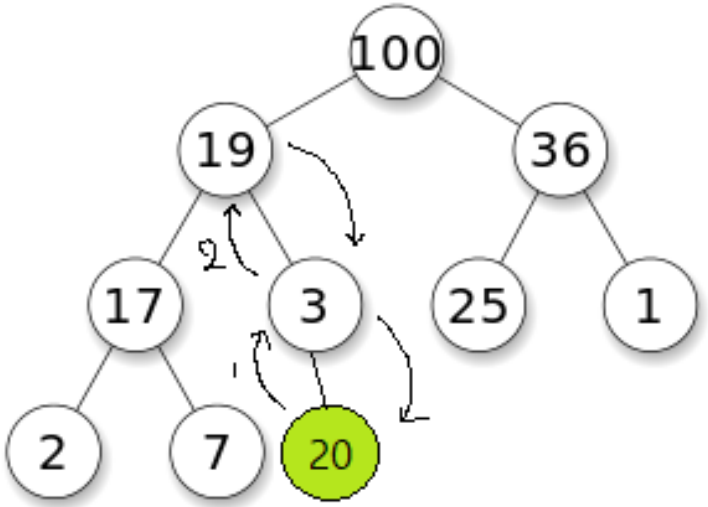
HEAP의 데이터 삽입

- 가장 말단에 데이터를 삽입한다.

- 삽입한 노드와 부모 노드를 비교

- 삽입한 데이터가 우선순위라면 부모와 교환한다.

- 더 이상 우선순위가 되지 않을 때 까지 과정을 반복한다



HEAP의 데이터 삭제

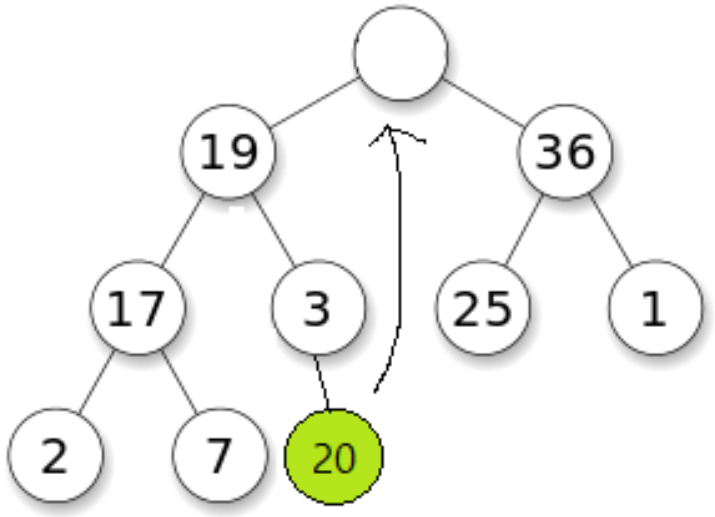
HEAP의 데이터 삭제는 항상 루트에서 일어난다.

- 루트 노드를 제거한다.

- 루트 노드가 있던 자리에 가장 말단의 노드를 삽입한다.

- 루트에 들어간 노드와 자식 노드를 비교한다.

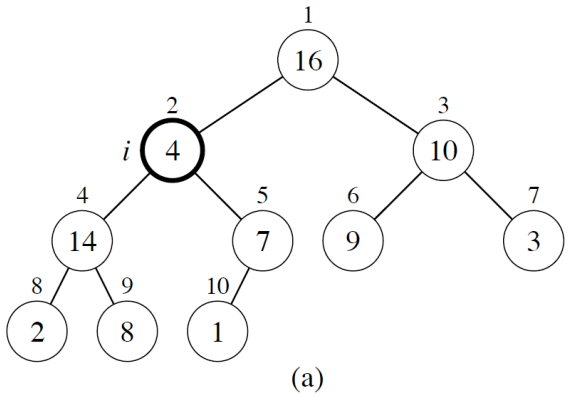
-우선 순위 조건을 만족 할때 까지 비교를 반복한다.

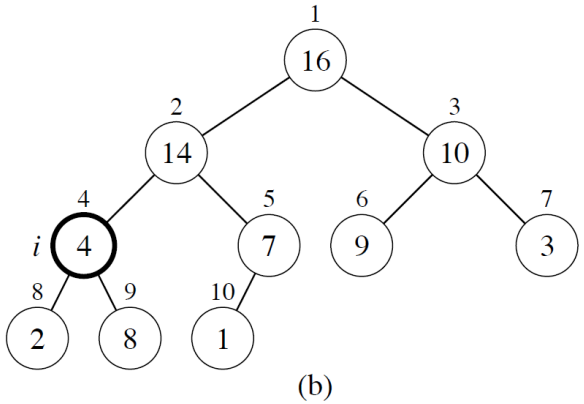


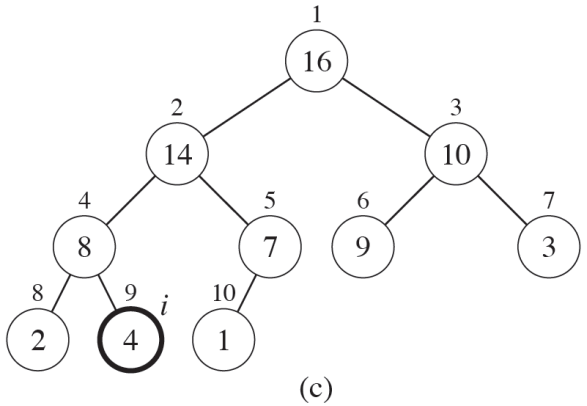
HEAP Sorting

- 힙을 이용한 정렬 방법

- HEAP Sorting은 추가적인 메모리를 요구하지 않는다.







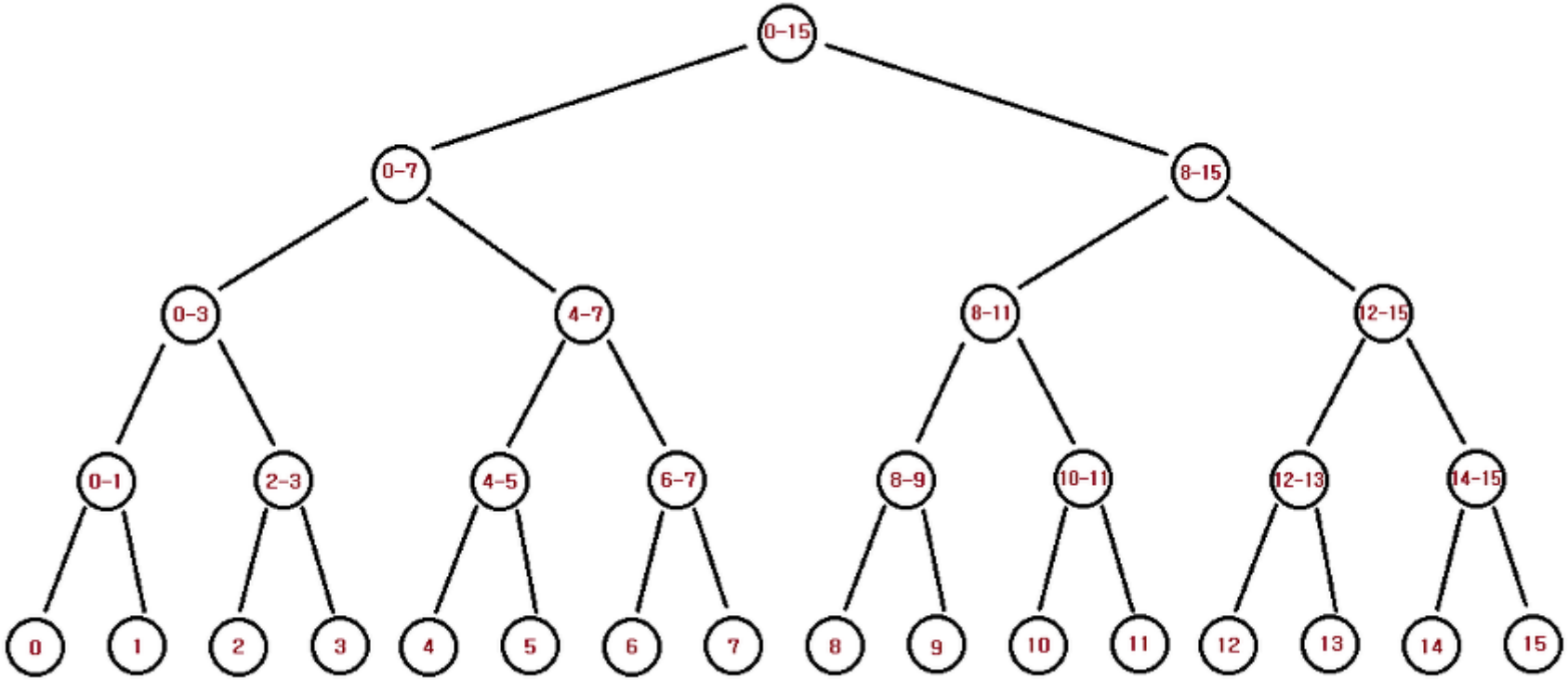
최대 힙으로 정렬하는 과정

부모 노드와 자식 노드를 비교하여 가장 큰 것을 부모 노드로 교체하고 이 과정을 더 이상 교체할 것이 없을 때 까지 반복한다.

Fenwick tree

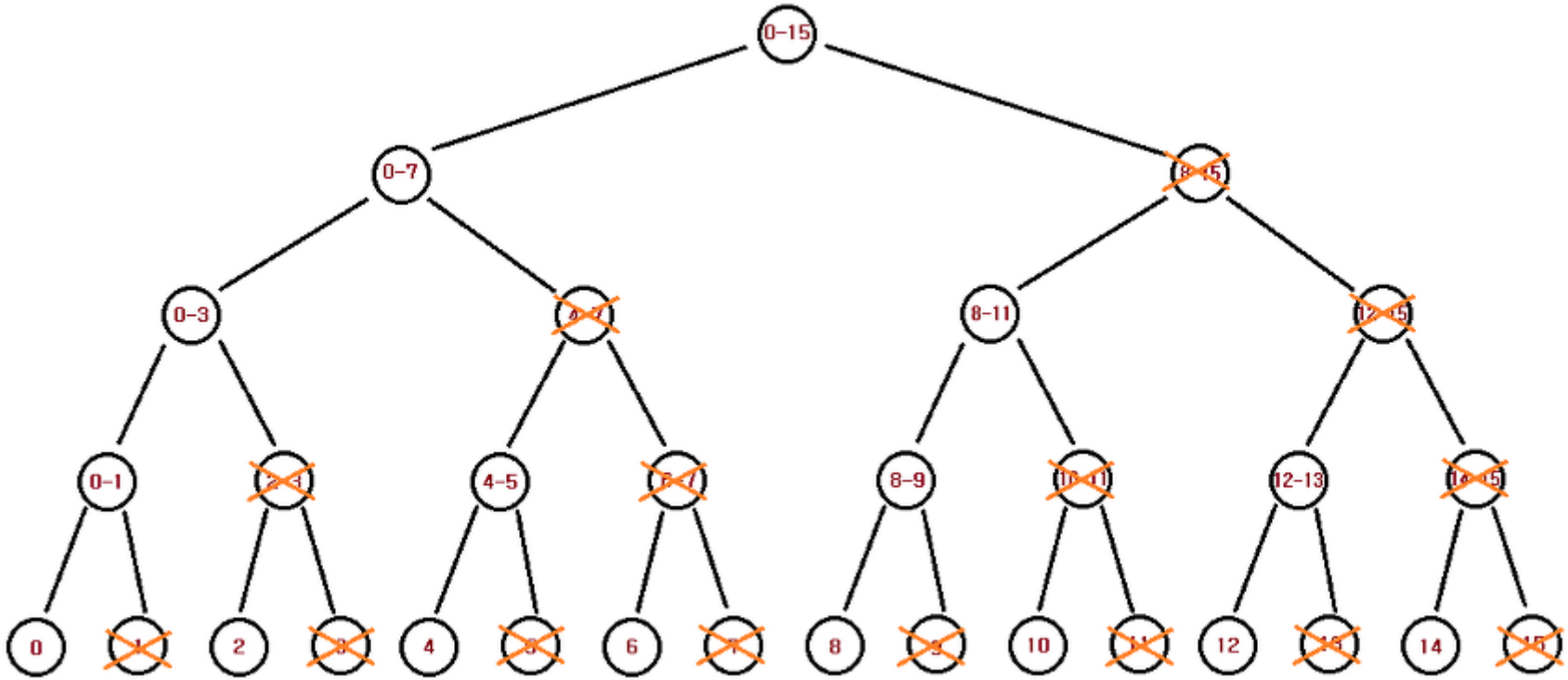
- 효율적으로 구간의 합을 구할 수 있는 자료구조
- 세그먼트 트리보다 효율적으로 메모리를 절약 할 수 있다.
- NUM += (NUM & - NUM) 으로 나타낼 수 있다.

SEGMENT TREE?



구간을 보존하고 있는 트리, 주어진 쿼리에 대해 빠르게 응답하기 위해 만들어진 자료구조

Fenwick tree

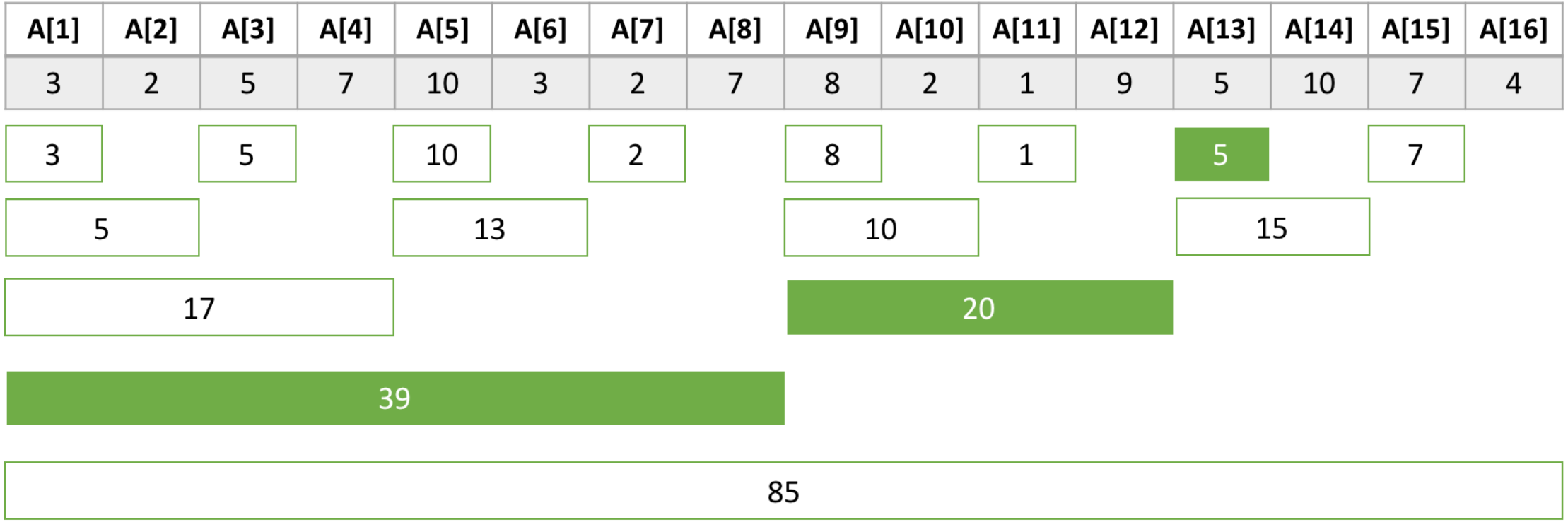


전 그림과 같은 세그먼트 트리를 위 그림 처럼 Fenwick tree로 변경한다.

A = [3, 2, 5, 7, 10, 3, 2, 7, 8, 2, 1, 9, 5, 10, 7, 4]인 경우에,
각각의 Tree[i]가 저장하고 있는 값은 그림과 같게 된다.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]	A[14]	A[15]	A[16]
3	2	5	7	10	3	2	7	8	2	1	9	5	10	7	4

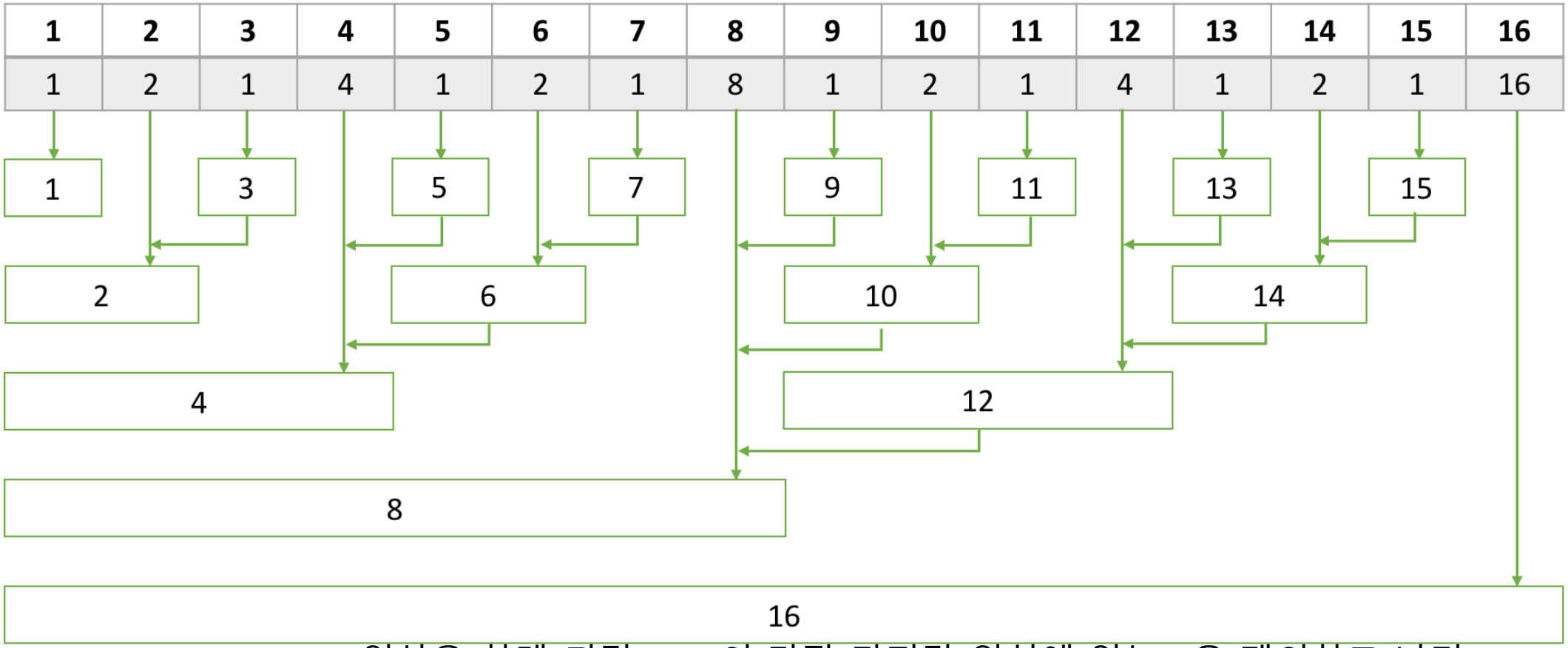
3	5	10	2	8	1	5	7
5	13			10		15	
17				20			
39							
85							



만약 A[1] 부터 A[13] 까지의 합을 구한다면 13의 이진수는 1101이다.
A[1] + …..+ A[13] = Tree[1101] + Tree[1100] + Tree[1000] 과 같다.

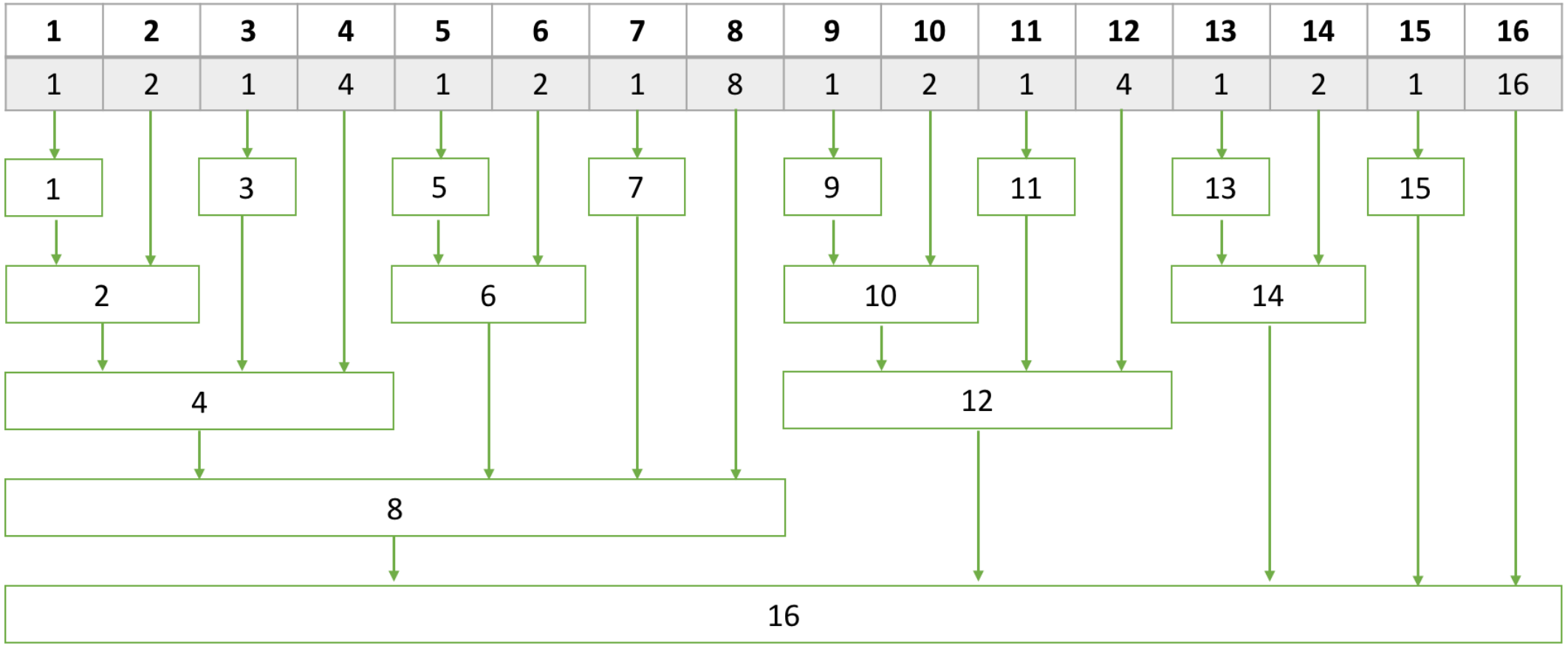
1101 -> 1100 -> 1000는 마지막 1의 위치를 빼면서 찾을 수 있다.

모든 i에 대해서 A[1] +……A[i]를 구하는 과정은 아래의 그림과 같다.



num += num & -num 연산을 하게 되면 num의 가장 마지막 위치에 있는 1을 제외하고 나머지는 다 0으로 표기된다. 만들어진 값을 num에 더하면 그 바로 아래 있는 박스로 가게 되고 그 작업을 i까지 반복하면 된다.

또한 어떤 구간의 합은 A[1] +…. +A[j]에서 A[1] + … + A[i-1]을 뺀 값과 같다.
따라서 Sum(j) – Sum(i-1)과 같다.



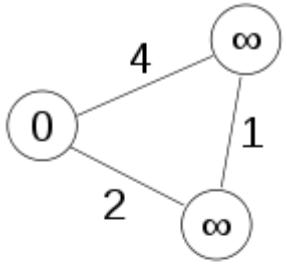
또한 i를 업데이트 할 경우 Tree[i]의 마지막 1의 값을 더해주면 된다.

예를 들어 7의 정보를 업데이트 하려 한다면 7의 이진수인 111의 마지막에 1을 더해주면 된다.

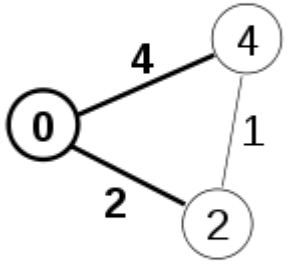
111 -> 1000 -> 10000 인 7과 8 그리고 16의 값을 업데이트 하면 된다.

Dijkstra's Algorithm

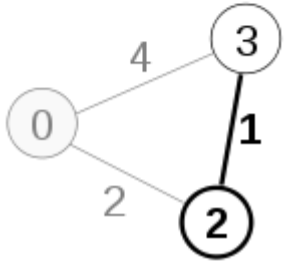
- 어떤 변도 음수 가중치를 갖지 않는 유향 그래프에서 주어진 출발점과 도착점 사이의 최단 경로 문제를 푸는 알고리즘



- **변 경감(edge relaxation)**이라고 불리는 기본 연산을 바탕으로 한다



- s에서 u까지의 최단 경로($d[u]$)를 이미 알고 있고, u에서 v까지 길이가 $w(u,v)$인 변 (u, v)가 존재할 때, s에서 v까지의 최단 경로는 u까지의 최단 경로에 변 (u, v)를 추가함으로써 얻을 수 있다. 이 경로의 비용은 $d[u]+w(u, v)$가 되며, 이 비용이 현재의 $d[v]$ 값보다 낮으면 $d[v]$를 새로운 값으로 바꾼다.



- 경감 연산은 모든 변 (u, v)에 대해 한번씩 경감이 적용되어 모든 $d[v]$가 최단 경로의 비용을 나타내게 되었을 때 끝난다.

문 제

농부 창호가 키우는 소들은 각자 자신의 목장에 얌전히 있다. 소가 없는 텅 빈 목장도 있다. 각각의 목장은 다른 목장들과 하나 혹은 더 많은 길로 연결이 되어 있다. 때때로 두 목장은 한 개 이상의 길로 연결되어 있을 수도 있으며, 적어도 한 개 이상의 목장들은 헛간으로 가는 길이 존재한다. 물론 소는 어떤 방향으로도 갈 수 있고, 모든 소들은 같은 속도로 걷게 된다.

목장들에는 'a' 부터 'z' 까지와, 'A' 부터 'Y' 까지로 번호가 붙여져 있다. 소 한 마리는 대문자 알파벳이 번호 매겨진 목장에 있다(목장 하나 당 소 한 마리 이다). 그리고 소가 없는 목장은 소문자로 번호가 매겨져 있다. 헛간은 그 헛간에 소가 있건 없건, 'Z' 로 번호가 매겨져 있다.

각각 목장과 목장, 혹은 목장과 헛간을 잇는 길의 거리가 주어질 때, 헛간으로 가장 먼저 올 수 있는 소와 그 거리를 구하는 프로그램을 작성하자.

입력 형식

첫째줄에는 P(1≤P≤10,000)개의 길 들이 입력된다. 둘째줄 부터 P+1 번째 줄까지, 한 줄에 두 문자와 숫자 하나가 입력 되는데, 이는 서로 연결 되는 두 목장(혹은 헛간)의 번호와, 그 길의 길이(1≤길이≤1,000)를 뜻한다.

출력 형식

가장 먼저 헛간에 도달하는 소가 원래 있던 목장의 번호와 그 소가 걷는 거리를 출력하라.

입력 예

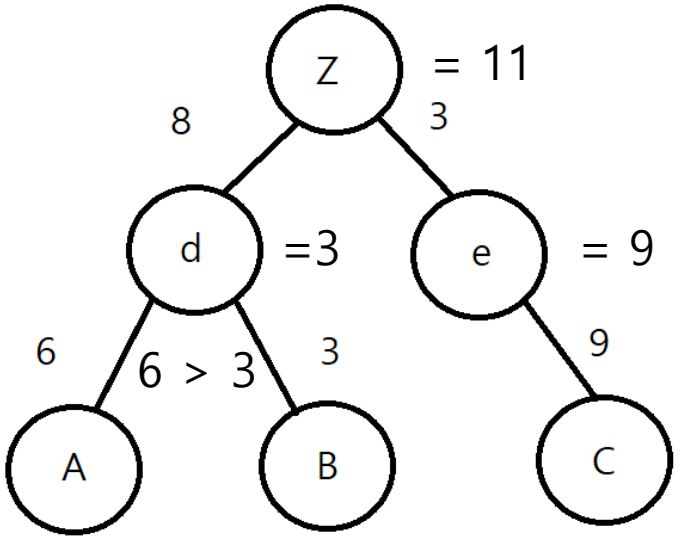
```
5
A d 6
B d 3
C e 9
d Z 8
e Z 3
```

출력 예

```
B 11
```

입력 예로 본 알고리즘

3 + 8 < 9 + 3



A = 6 + 8 = 14
B = 3 + 8 = 11
C = 9 + 3 = 12

A > B
B > C

B = 11

A에서 d로 가는 거리를 처음에 d 에 저장하고 B에서 d로 가는 거리를 저장한다. 그후 그 둘을 비교하여 더 작은 값을 d에 저장한다.

마찬가지로 Z에서도 d에서 Z까지의 거리를 더해주고 e에서 Z까지의 거리를 더해준 후 그중 가장 낮은 값을 Z에 저장한다. Z까지 가는 모든 경로의 수가 계산 되었을 때 가장 작은 값과 출발 지점을 출력한다.