Lecture 10: 범위 탐색을 위한 자료구조 Part I.

이번 장에는 기록된 특정 하나의 원소가 아니라 저장된 원소가 아닌 일정한 범위에 대한 query에 대하여 그 안에 포함된 자료를 검색하는 자료구조에 대하여 살펴보고자 한다. 구체적인 내용으로는 Range Tree, Binary Indexed, Priority Tree에 대하여 살펴본다. 범위의 차원이 1차원이면 쉽지만 범위의 차원이 2차 이상이면 간단히 처리할 수 없다. 다차원 범위 탐색은 여러 응용 상황에서 발생한다. 예를 들어 시간적으로 12시에서 16시, 즉 $t = [12, 16]$에 활동한 사람 중에서 나이가 $[30, 45]$인 사람을 모두 찾는 문제는 2차원 범위가 사용된다.

10.1 구간 탐색 문제의 일반적 구조

이번 장에서는 어떤 기하공간에 주어진 범위에 포함된 기하객체를 탐색하기 위하여 븍별히 고안된 자료구조를 살펴보고자 한다. 기본적으로 이런 류의 자료구조는 모든 자료가 주어지고 더 이상 변화가 없는 Static version이 있고, 중간에 새로운 자료가 들어오거나 삭제, 또는 기존의 자료의 key 값이 변화하는(updated) 경우가 있다.

전자의 경우에는 비용이 들지만 좋은 전처리 과정으로 이후의 query를 빠르게 처리할 수 있다. 예를 들어 일차원 배열에서 특정 구간 데이터 값을 구하는 문제$\sum_i^j A[k]$가 전형적인 예이다. 이 문제는 accumulated sum 배열을 전처리를 통하여 미리 만들어 두면 그 배열의 두 원소의 차이(difference)를 이용하여 쉽게 $O(1)$에 구할 수 있다. 이것은 2차원 공간에도 마찬가지이다.

그런데 만일 중간에 어떤 값 $A[i]$이 변화하거나, 또는 삭제될 경우, 또는 작업 도중에 새로운 값이 추가(insert)될 경우이다. 이 경우라면 자료구조의 많은 부분을 갱신해야 한다. 결국 범위 자료 처리를 위한 동적 자료구조는 이런 작업을 얼마나 최소화시킬 수 있는가의 문제로 귀납되게 된다.

(a) Range tree(범위 트리)의 입력은 '점(points)'이다. 질문은 어떤 구간(axis-parallel) range space이다. 답은 query 공간에 '포함'된 모든 점을 report하는데 특화된 자료구조이다.

(b) Priority Search Tree의 입력은 점(points)이다. 그리고 각 data item은 어떤 aux 정보인 priority 정보를 따로 가지고 있다. 일반적인 priority queue Q는 현재의 Q에 포함된 데이터 중에서 highest priority item을 report하고 이것을 Q에서 제거하는 작업이다. 그런데 특정 원소를 찾을 수는(억지로 찾을 수는 있겠지만) 없다. 그런데 우리는 이 searching 작업이 가능한 특별한 queue를 만들고자 한다. 즉 주어진 Q에서 어떤 범위 $[l, r]$에 있는 원소 중에서 가장 highest priority item을 찾는 작업을 가능하게 하는 것이 이 자료구조이다.

(c) Binary indexed tree의 모든 item에는 items-count가 붙어있다. 우리는 어떤 두 개의 index $l < r$ 사이에 포함된 모든 item들의 합(weighted sum)을 빠르게 보고하는데 최적화된 자료구조를 구성해야 한다. 그리고 이 문제에서는 특정 원소의 item-count를 실시간으로 변화시킬 수 있ㄷ자. 삭제, 삽입은 불가능하지만 item count update는 가능하게 한다. 이 작업을 최적으로 할 수 있는 자료구조를 만들어야 한다. 입력은 range이며 출력은 sum이다.

(d) Segment tree(선분 트리)는 어떤 구간(intervals)을 저장한다. 그리고 주어진 점(point)에 대하여 어떤 구간이 포함되어 있는지 그 구간을 보고하는 작업에 최적화되어 있다. 즉 입력은 " 점" 이며 출력은 interval 이다.

(e) Interval tree(구간 트리)는 segment tree와 같이 구간(interval)을 입력으로 받아 저장한다. 그러나 segment tree 와 달리 입력은 interval이며 출력은 주어진 구간에 걸쳐있는 모든 interval을 출력하는 것이다. 즉 입력, 출력 모두 interval이 segment tree와 다른 점이다. 그리고 입력 interval이 degenerated(퇴화된) 구간, 즉 점일 경우에는 segment tree의 기능을 포함한다. 즉 Interval tree는 segment tree를 포함한다.

10.2 1차원 범위 트리(Range Tree)

1차원 직선상에 n 개의 실수 $S = \{a_1, a_2, \cdots, a_n\}$ 이 주어졌을 때, 주어진 폐구간(query interval) $I = [b_1, b_2]$ 에 포함되는 S 의 모든 원소를 계산하는 문제를 Range query 라고 부른다. Range query 에는 이와 같이 모든 점을 계산하는 range reporting 문제와 모든 점의 개수 만을 계산하는 range counting 문제로 나눌 수 있다.

Range query 문제에서 집합 S 의 모든 원소가 동적으로 insert, delete 될 수 있기 때문에 단순한 전처리 과정으로 처리할 수는 없다. 따라서 엄청난 수의 연속된 range query를 효율적으로 처리할 수 있는 자료구조가 만들어져야 한다. 이러한 작업을 효율적으로 처리하는 자료구조 일반을 range tree라고 한다. 이 range의 차원에 따라서 다양한 자료구조가 만들어 질 수 있다.

문제 10.2.1 일차원 점 집합 $S = \{a_1, a_2, \cdots, a_n\}$가 있다. 이 S에 새로운 점이 추가되거나 삭제되지는 않는다. 즉 *static range query* 문제를 생각해보자. 만일 어떤 *range query* $I = [b_1, b_2]$에 대하여 그 안에 포함되는 점의 갯수만을 출력하는 문제는 $O(1)$에 풀어보시오. □

우리는 먼저 1차원 Range Tree를 살펴보자. 1차원 Range Tree 는 Red-Black Tree(RBT) 와 같은 균형트리(Balanced Binary Tree)를 근간으로 다음과 같이 만들어진다.

(1) Range Tree 의 단말노드는 n 개의 실수 $\{a_1, a_2, \ldots, a_n\}$ 를 원소로 한다.

(2) Range Tree 는 내부노드는 $n-1$ 개의 실수 $\{a_1, a_2, \ldots, a_{n-1}\}$ 를 원소로 한다. 예를 들어, S = {2, 3, 5, 9, 14, 17, 21, 24, 27, 29, 35, 47, 58, 64 }를 원소로 하는 range tree의 예는 다음 그림-1과 같다.

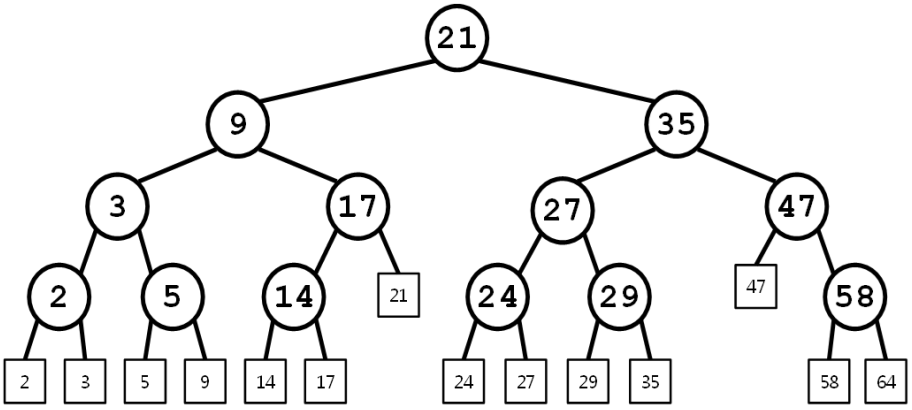


Figure 1: 14개의 실수로 구성된 range tree의 예. 가장 큰 원소인 64는 internal node 에서 빠져있다. 이것은 따로 global variable로 O(1) 공간에 저장해서 처리한다.

$I = [l, r]$에 대하여 그 안에 포함되는 모든 점을 report하는 알고리즘은 다음과 같다.

step 1: range tree 에서 range query의 양끝 값 l, r 에 대하여 각각 search 하면서 최종적으로 도달한 단말 노드를 각각 t_l, t_r로 나타내기로 하자.

step 2: 루트 노드 r 로부터 시작하여 단말 노드로 내려가면서 노드 x 의 key 값이 $I = [l, r]$에 포함되는 첫 노드를 구한다. 이 노드를 `vsplit` 이라고 부르자. 이 노드는 루트에서 시작하여 단말 노드 t_l, t_r로의 경로가 갈라지는 최초의 갈림길 (branching) 노드이다.

step 3: `vsplit` 으로부터 단말 노드 t_l로 내려가면서 어떤 노드 x의 왼쪽 child 로 내려갈 경우에는 이 노드 x의 오른쪽 subtree에 속하는 모든 단말 노드의 점들은 모두 주어진 range에 포함되므로 이 노드들을 모두 report 한다.

step 4: 어떤 노드 x의 오른쪽 child 로 내려갈 경우에는 아무런 작업을 하지 않는다.

step 5: 최종적으로 단말노드 t_l 의 key 가 range에 포함되면 이 노드까지 report하고 작업을 종료한다.

step 6: 마찬가지로 단말노드 t_r 로 내려가면서 어떤 노드 x의 오른쪽 child 로 내려갈 경우에는 이 노드 x 의 왼쪽 subtree 에 속하는 모든 단말노드의 점들은 모두 주어진 range에 포함되므로 이 노드들를 모두 report 한다. 어떤 노드 x의 왼쪽 child 로 내려갈 경우에는 아무런 작업을 하지 않는다. 최종적으로 단말노드 t_r 의 key 가 range에 포함되면 이 노드도 report 한다.

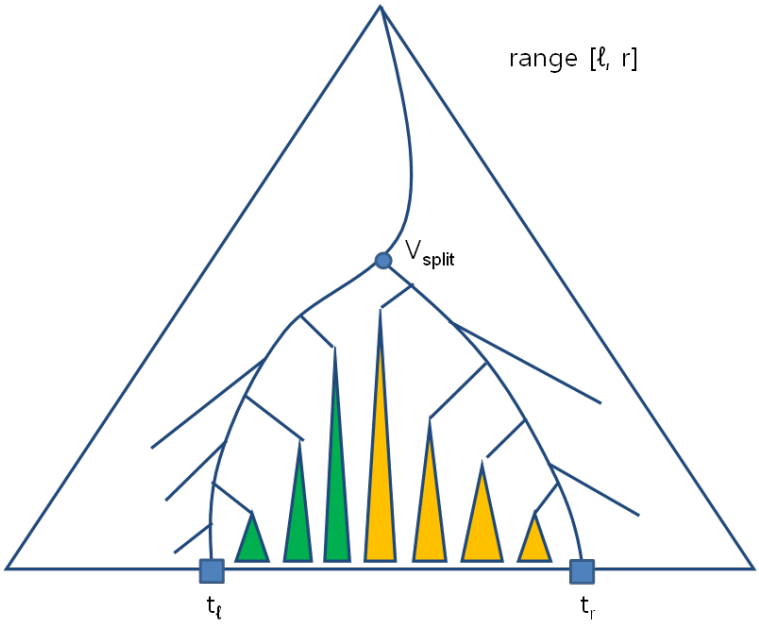


Figure 2: Range query $I = [l, r]$에 포함되는 모든 점이 포함된 모양

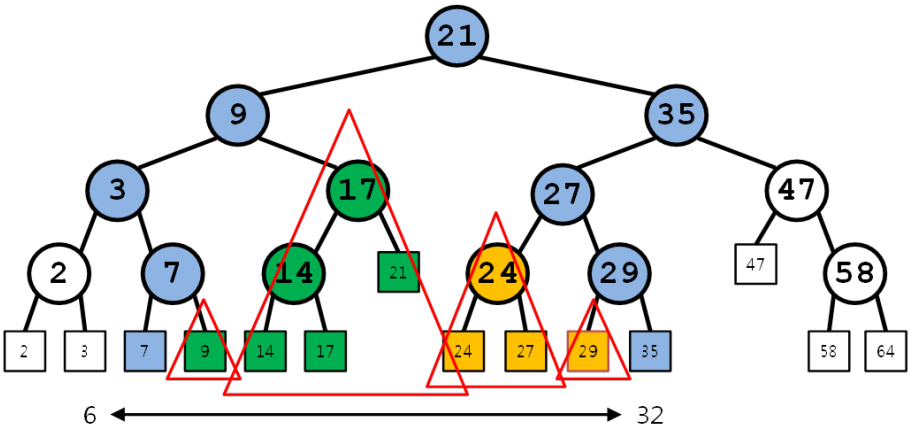


Figure 3: 14개의 실수로 구성된 range tree에서 구간 [6,32]에 속한 원소를 찾는 예. 21과 [6,32]를 비교한 결과 6은 21보다 작으므로 왼쪽 트리를 탐색해야 한다.

위 작업을 장점을 병렬처리를 완벽하게 할 수 있다는 점이다. 기하문제의 특징을 "거리 공간"의 독립성이 보장되기 때문에 항상 병렬처리의 가능성을 가지고 있다는 점이다. 이 점을 항상 염두에 두어야 한다.

문제 10.2.2 n개의 점이 있는 *range reporting problem*의 *time complexity*를 *output sensitive complexity*로 제시하시오. 만일 우리는 각 점 자체가 아니고 그 점들의 갯수만 관심이 있다고 할 경우 필요한 *time complexity*를 밝히시오. □

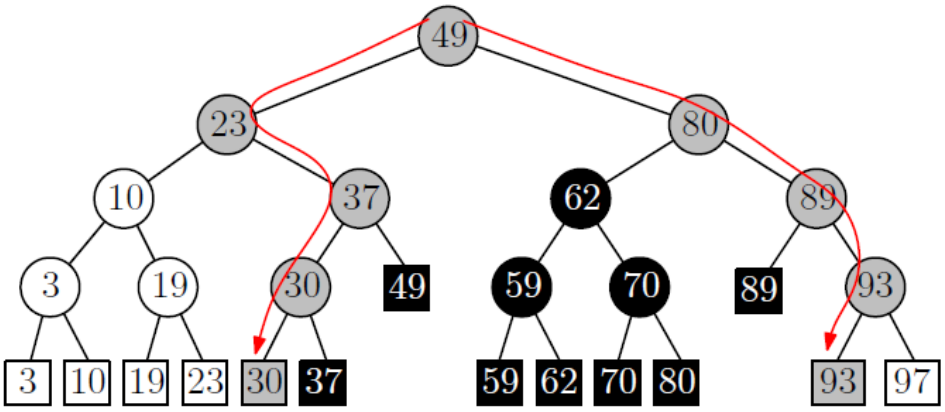


Figure 4: 구간 [25,90] 에 포함된 모든 점을 찾아가는 과정.

Solution) Range Reporting Query 는 $O(\log n + k)$ 시간에 수행할 수 있다. 여기에서 k 는 report 되는 원소의 개수이다. Range Counting Query는 다음과 같이 Range Tree 의 각 내부노드에 그 노드를 루트로 하는 subtree 의 단말노드 개수를 저장하게 되면, Range Counting Query 를 $O(\log n)$ 에 처리할 수 있다. 위의 예에 대하여 아래 그림을 참고해서 보면 된다.

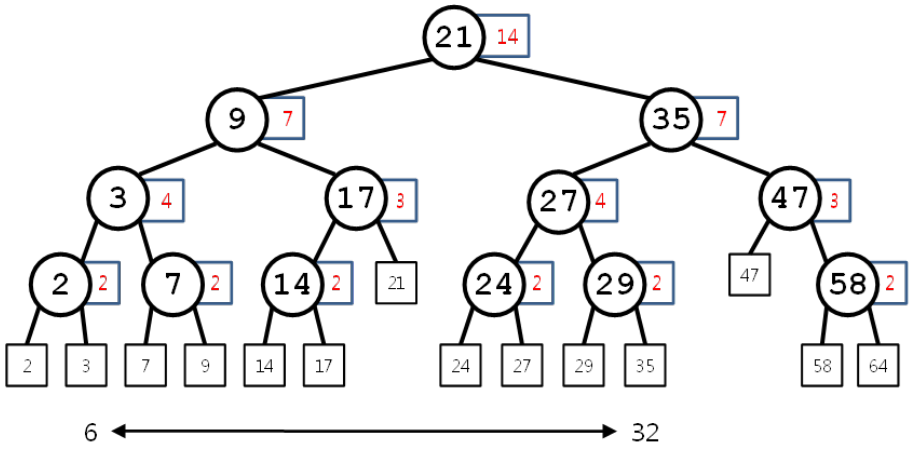


Figure 5: 그림에서 붉은 색의 숫자의 해당 트리의 노드 갯수를 나타낸다. 따라서 만일 각각의 점 좌표가 아니고 갯수만을 보고한다면 하면 이 '붉은' 수의 합을 이용해서 간단하게 할 수 있다.

10.3 2차원 범위 트리(2D-Range Tree)

아래 그림에서와 같이 2차원 평면 상에 n 개의 점 $P = \{p_1, p_2, \ldots, p_n\}$ 이 주어졌을 때, 주어진 직사각형 $R = [a1, a2] \times [b1, b2]$ 에 포함되는 P 의 모든 점를 계산하는 문제를

2 차원 Range Query 라고 부른다.

이러한 range query 문제에서는 집합 P 의 모든 점이 동적으로 insert, delete 될 수 있으며, 주어진 많은 수의 range query 를 효율적으로 처리할 수 있는 자료구조가 만들어져야 한다. 이러한 작업을 효율적으로 처리하는 자료구조를 2차원 range tree라고 하면 대부분의 기하문제, 또는 이를 활용한 DB 탐색 문제에 활용되는 매우 요긴한 자료구조이다.

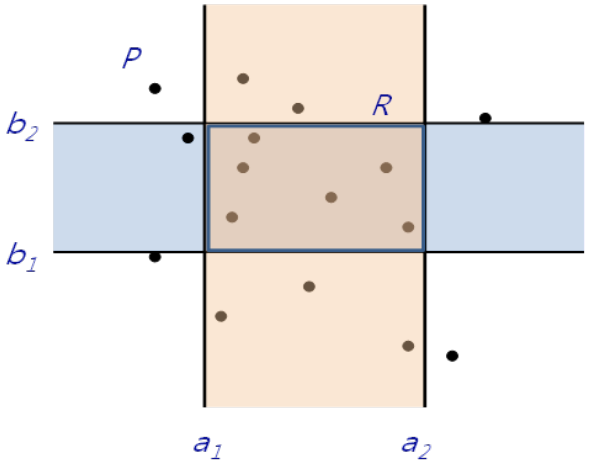


Figure 6: 2차원 공간에서 range searching을 요구하는 문제의 예. 직사각형 안에 들어있는 점을 모두 report 해야 하는 문제이다.

2차원 range tree는 다음과 같이 만들어 진다. 아래 그림-7 참조해서 설명해보자.

먼저 P에 속하는 모든 점들의 x-좌표를 기준으로 1차원 range tree T를 만든다. 이 트리에서는 위 그림에서와 같이 x-좌표가 $[a_1, a_2]$ 인 수직 구간에 포함되는 모든 점을 계산하는 query를 처리할 수 있다.

이제 그 다음 단계를 설명해보자. 이제는 위에서 만든 트리 T의 모든 내부 노드 t 마다, t를 루트로 하는 T의 모든 단말노드에 속하는 점들의 집합 Q_t를 정의할 수 있다. 이 때, Q_t에 속하는 모든 점들에 대하여 y-좌표를 기준으로 1차원 range tree를 만들어 이 트리를 노드 t에 연결하면 된다. 이때 해당하는 점의 집합에서 해당 축의 방향으로 가운데(median) 원소를 root로 두고 양쪽의 거의 동일한 수의 노드에 대해서 각각 recursive하게 range tree를 만든다.

문제 10.3.1 *2차원 range tree의 구성 알고리즘을 formal하게 제시하시오. 그리고 이 알고리즘의 complexity가 $O(n \log n)$ 임을 또한 보이시오.*□

Sol.) 한 쪽 차원으로 range tree를 만드는데 걸리는 시간이 $\Theta(n)$이다. 왜냐하면 tree의 모든 노드에 대하여 $O(1)$의 작업을 하고 그 노드의 수가 $\Theta(2n)$이기 때문이다. 문제는 y축으로는 다양한 크기의 트리가 $O(n)$개 있기 때문에 그것을 따로 계산해야 한다. root

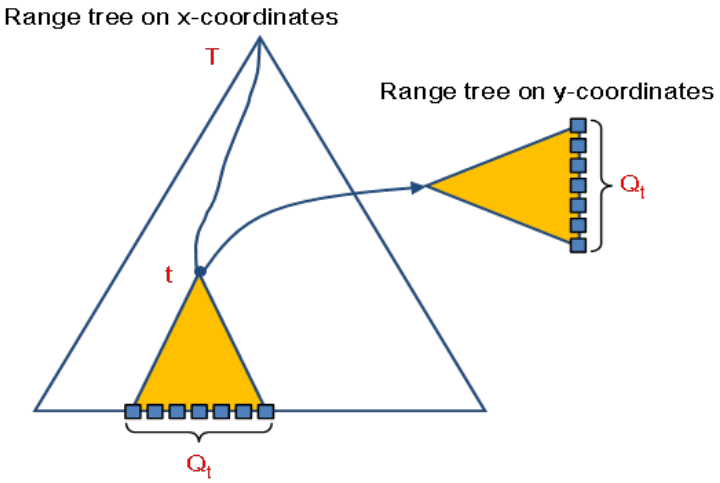


Figure 7: 2개의 일차원 Range tree를 이용해서 2차원 range tree를 구성.

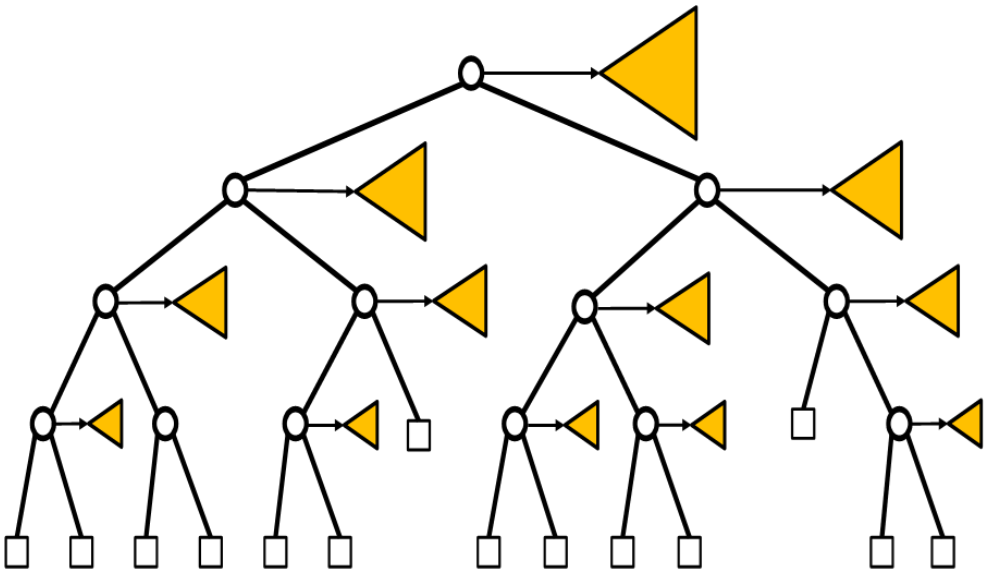


Figure 8: 어떤 internal node가 포함하고 있는 모든 점, 즉 특정 구간에 속한 점에 대하여(within) 그 점들의 다른 차원으로 구성되는 새로운 range tree를 따로 만들어 두 노드를 각각 연결하는 것이다.

의 경우를 보자. n에서 median을 찾아내는데 걸리는 시간은 $O(n)$이다. 각 노드에 대하여 양쪽의 $n/2$개에 대하여 만드는데 걸리는 시간이 $Y(n)$이라고 한다면 다음과 같은 식이 성립한다.

$$Y(n) = Y(\lfloor n/2 \rfloor) + Y(\lfloor n/2 \rfloor) + O(n)$$

이것은 Quick sort의 best case와 동일하므로 $Y(n) = O(n \log n)$이 된다.

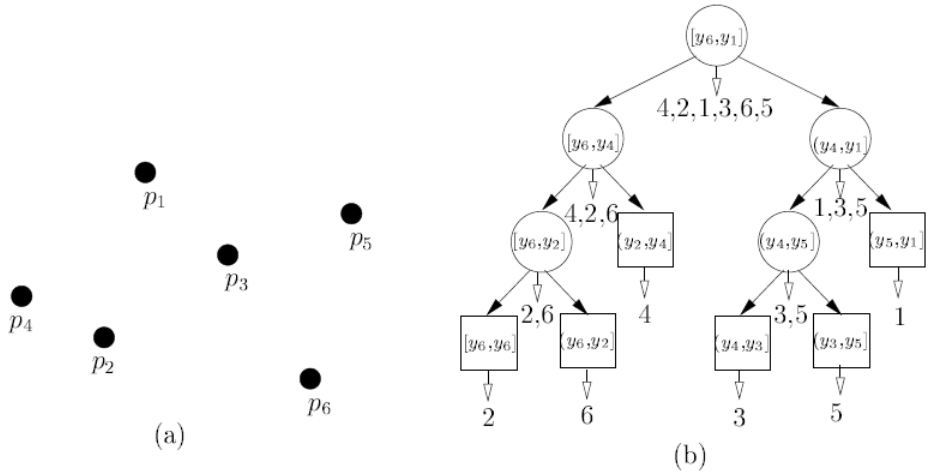


Figure 9: 6개의 점으로 이루어진 2D range Tree의 실제 예

```
Build2dRangeTree(P)
     Ty = P에 속하는 점들의 y-좌표를 기준으로 만든 1차원 range tree
     if (P에 한 개의 점만 있는 경우) :
        한 점으로 이루어진 leaf 노드를 만들고, Ty를 이 노드에 연결
     else :
      P에 속하는 점들을 x좌표의 가장 중앙에 있는 점 pm을 기준으로
             - 그 왼쪽에 있는 점들의 집합 Pl과
             - 오른쪽에 있는 점들의  집합 Pr로 구분하여 나눈다.
      점 pm 을 원소로 하는 노드 v 를 만들고, Ty를 이 노드에 연결한다.

      v->left = Build2dRangeTree(Pl);
      v->right = Build2dRangeTree(Pr);

     return v;
```

문제 10.3.2 *2*차원 *range tree*에서 특정 *2*차원 구간에 속한 점을 *report*하는 알고리즘을 설명하고 그 알고리즘의 시간복잡도와 *space complexity*를 밝히시오. □

Solution) 2차원 range query 가 1차원 range query 와 다른 점은, 1차원 range query 에서 어떤 노드 x 의 각 subtree에 속하는 원소를 report하는 것으로는 완성되지 않는다. 즉 노드 x에 연결된 y-좌표 range tree에 대하여 다시 한 번 더 1차원 range tree를 수행하는 것이다. 알고리즘이 수행되는 시간 복잡도는 보고되는 점의 수가 k개 일 때, x축으로 보고해야 하는 작업이 k번이다. x축으로 볼 때 선택되는 subtree의 갯수가 최대 $O(2\log n)$이고 이 갯수만큼에 대하여 다시 y축으로 원하는 점들을 걸러야 한다. 따라서 output sensitive time은 $O(\log_2 n + k)$ 임을 쉽게 알 수 있다.

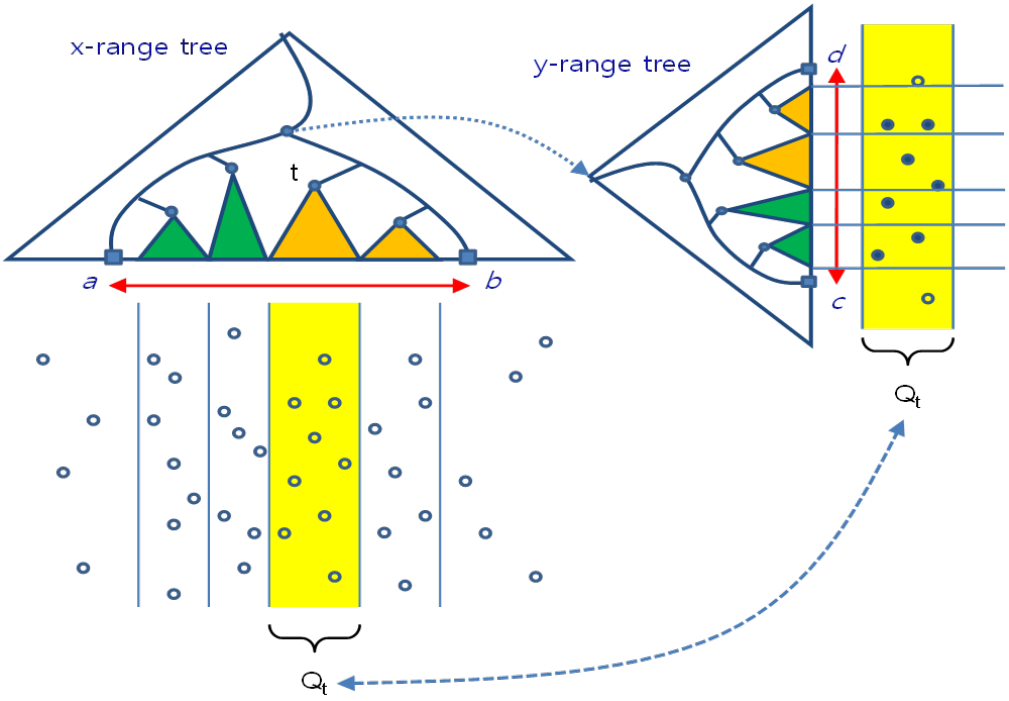


Figure 10: 2차원 Range tree에서 실제 탐색을 수행하는 과정의 예

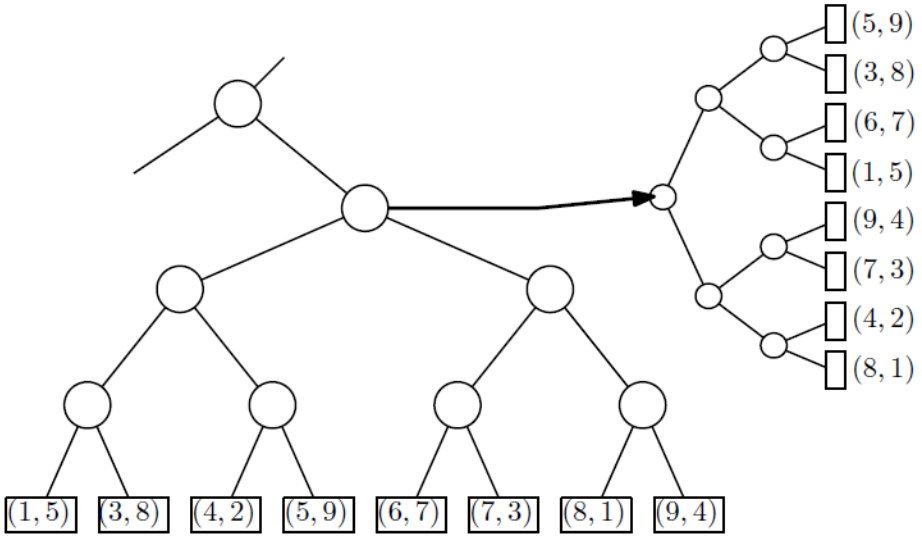


Figure 11: 2차원 Range tree에서 특정 구간에 대한 예. x축으로는 x 좌표로 정렬되어 있으며, 세로축으로는 y 쪽 점으로 데이터가 정렬되어 있다. 따라서 두 축에 대하여 2번 탐색을 하면 된다.

따라서 우리는 다음의 두 정리를 얻을 수 있다.

정리 10.3.1 *[1-D range searching]*
*1*차원의 n개의 점에 대하여 $O(n \log n)$ 시간에 전처리가 될 수 있고 그 크기는 $O(n)$이다. 따라서 *1-D range [counting]* 질의는 $O(\log n + k)$ 시간에 해결된다. ■

정리 10.3.2 *[2-D range searching]*
*2*차원의 점 n개에 대하여 *range search tree*는 $O(n \log n)$ 시간에 전처리가 될 수 있고 그 크기는 $O(n)$ 이다. 그리고 *2D range query*에 대하여 $O(\sqrt{n} + k)$시간에 처리할 수 있다. 여기에서 k는 해답 점의 개수이다. 만일 *counting*만 고려한다면 $O(\sqrt{n})$ 시간에 가능하다.

10.3.1 k-차원 범위 트리(k-D Range Tree)

우리는 앞서의 방법을 그대로 적용하여 2차원을 k차원으로 확장할 수 있다.

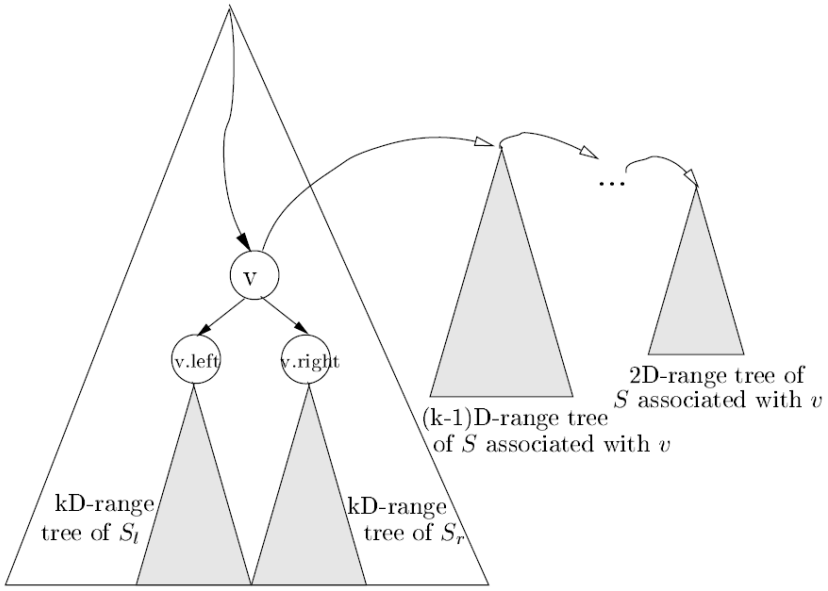


Figure 12: 일차원 Range tree를 사용해서 다차원(high-dimension) range tree를 구성하는 방법.

만일 $T(n,k)$를 n개의 원소를 가진 k차원 공간에서의 query 시간이라고 한다면 다음의 재귀식을 유도할 수 있다.

$$T(n,k) = \begin{cases} O(1) & \text{if } n = 1 \\ O(n \log n) & \text{if } k = 2 \\ 2 \cdot T(n, k-1) + T(n/2, k) + O(n) & \text{otherwize} \end{cases}$$

제일 마지막 recursion이 보여주는 것은 전체 점에 대하여 전체 차원에서 한 차원 낮은 특정 차원으로 정리하고 ($T(n, k-1)$), 그 점에 대하여 같은 차원에 대하여 점의 갯수를 양분하여 처리하는 것이다. 위 식을 정리하면 $T(n,k) = O(n \log^{k-1} n + n \log n)$ 이 됨을 알 수 있다. 그리고 d차원의 자료구조에 대하여 그 크기는 다음과 같이 계산된다. 따라서 $S_d(n)$ of a d-dimensional range tree의 크기 (size) 는 다음을 만족시킨다.

$$S_1(n) = O(n) \text{ for all n}$$
$$S_d(1) = O(1) \text{ for all d}$$
$$S_d(n) \leq 2 \cdot S_d(n/2) + S_{d-1}(n) \quad \text{for } d \geq 2$$

이 재귀식을 정릴하면 다음을 확인할 수 있다.

$$S_d(n) = O(n \log^{d-1} n)$$

정리 10.3.3 *d*-차원의 점은 $O(n \log^{d-1} n)$ 시간안에 크기 $O(n \log^{d-1} n)$ 의 트리로 구현될 수 있으며 모든 *d-dimensional range* 질의는 $O(\log^d n + k)$ 시간에 처리가능하다. 여기에서 k 는 보고되는 정답의 갯수이다. ■

이 문제를 DB에서 가장 보편적으로 사용하는 문제인데, 이것을 k d-tree를 이용해서 풀면 $O(n)$ 의 공간 크기에 시간복잡도는 $O(n^{1-1/d} + k)$ 로 해결할 수 있다.

10.4 2-차원 범위트리와 fractional cascading

우리는 2차원 Range Tree의 Query Time을 fractional cascading 기법을 이용하면 $O(\log_2 n)$ 에서 $o(\log n)$ 으로 낮출 수 있다. 즉$O(\log_2 n)$ 보다는 확실히 낮은 복잡도로 처리할 수 있다. 그리고 우리는 이 방법을 확장하여 d차원 공간에서도 이 문제의 복잡도를 $O(\log^{d-1} n)$ 로 줄일 수 있다. 간단한 문제는 먼저 풀어보자.

문제 10.4.1 두개의 *sorted array A1*과 *A2*가 있고 $A1 \succ A2$이다. $A1 \succ A2$ 의 의미는 배열 *A2*는 *A1*의 *proper subset*이며 순서도 그래도 유지하고 있음을 의미한다. 아래 그림-*10.4*을 참조하시오. 이 두 배열에서 구간 *range*$[a, b]$ 질의를 모두 처리하는 방법을 생각해보자.

가장 trivial한 방법은 a를 binary로 찾아서 b가 나올 때 까지 오른쪽으로 밀고 나가면서 모든 답을 report하는 일이다. 이 일은 A1와 A2가 아무런 관계가 없어도 같은 복잡도 $O(\log n)$에 처리될 수 있다. 그런데 우리는 $A1 \succ A2$ 이라는 이 좋은 관계를 활용하고 못하고 있다. 어떻게 하면 이 "좋은" 특성을 활용할 수 있을지 생각해보자. 특히 A1의 크기가 $O(n)$이고 A2의 크기가 그보다 훨씬 작은 $O(\log n)$이나 $O(\sqrt(n))$인 경우를 생각해보자.

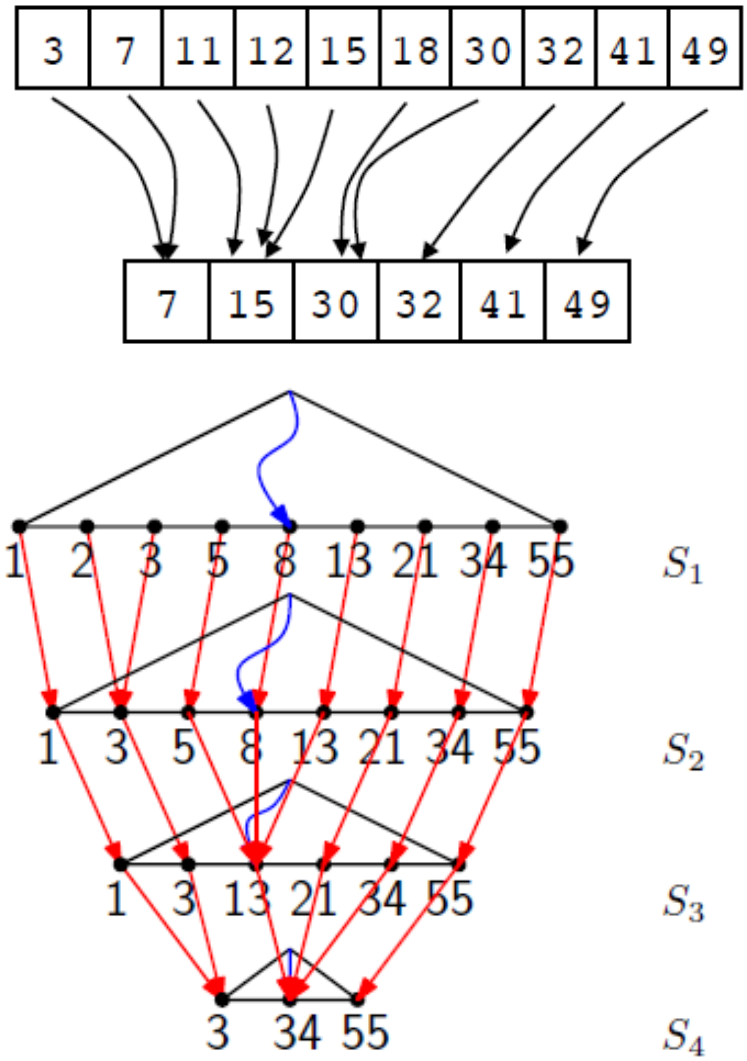


Figure 13: (a) 상위 배열은 하위 배열의 최소 상한의 index를 표시, (b) 같은 방식으로 처리되는 과정

만일 이러한 식으로 배열이 피라미드 형식으로 연결되어 있다고 가정해보자. 한 단계 내려올 때 마다 배열의 크기를 반(1/2)로 줄이고 그때 각각 적절한 값과 link을 연결해보자. 만일 어떤 탐색 구간 `[l,r]`이 주어질 경우 이것을 가장 낮은 배열(가장 작은 배열)에서 부터 올라가면서 찾는아가면 된다. 상위 배열의 점보다 큰 첫 점을 찾는 작업은 두 배열을 scan 하면서 link 시킬 수 있다. 이렇게 되면 한번 끝 점을 찾은 다음에 아래로 내려가면서 시작점을 바로 찾을 수 있다.

`RangeQuery`([x1,x2] by [y1,y2]) 작업의 코드는 다음과 같다.

step(1): Search for xsplit

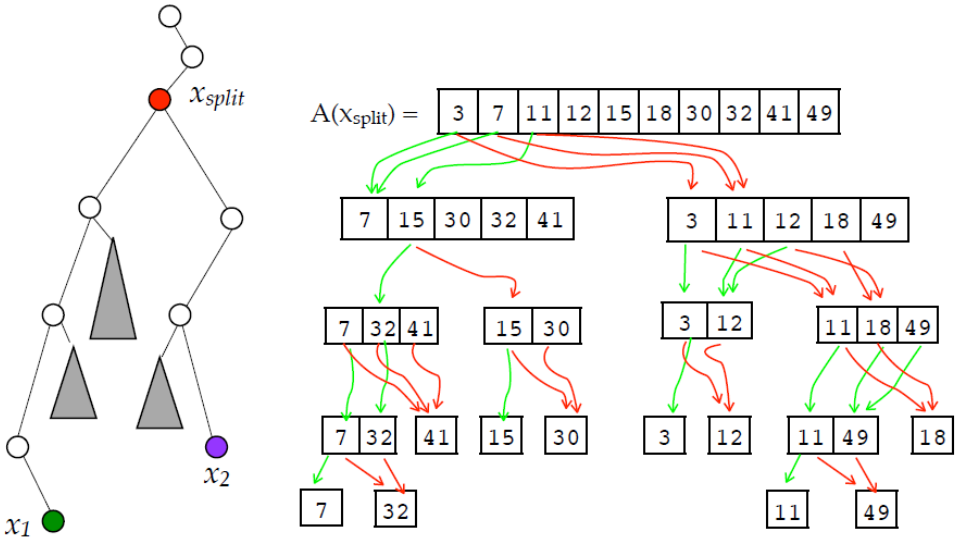


Figure 14: 각 노드에 포함된 점들을 다른 차원 y-축 보조트리에 저장하는대신 y-축의 순서로 정렬된 배열에 각각 저장한다. 따라서 xsplit을 시작하는 점 (point) 에서 원래는 다시 이진탐색 $[y_1, y_2]$를 수행해야하는데 이 작업을 상위 배열의 index를 이용해서 $O(1)$에 찾고 오른쪽으로 밀고 나가서 y_2를 지날 때 까지 지속한다.

step(2): Use binary search to find the first point in A(xsplit) that is larger that y1.

step(3): Continue searching for x1 and x2, following the now diverged paths.

step(4): Let `u1--u2--u3--uk` be the path to x1. While following this path, use the "cascading" pointers to find the first point in each A(ui) that is larger than y1. [similarly with the path `v1--v2--vm` to x2]

step(5): If a child of u_i or v_i is the root of a subtree to output, then use a cascading pointer to find the first point larger than y1, output all points until you pass y2.

이 작업으로 인하여 추가의 $O(\log n)$ binary search 대신에 처음 한번만 이진탐색을 하고 array link를 이용하여 그 다음 단계에서는 바로 찾는다. 따라서 이전의 $O(\log^2 n)$의 시간은 $O(\log n)$이 되므로 2d-rectangle range queries은 $O(\log n + k)$ 시간에 처리되며 d-dimension에 대해서 그 복잡도는 $O(\log^{d-1} n + k)$ 시간에 해결된다.

정리 10.4.1 어떤 자료구조 D_i가 있고 이것에 약간의 변형을 가한 D_{i+1}이 있다. 이 두 자료구조의 차이가 일정이하 일 때, $|D_i - D_{i+1}| \leq c_0$이라면 각 자료구조에 적용하는 어떤 함수 $f()$의 추가작업은 다음으로 *bound*된다.

$$|f(D_i) - f(D_{i+1})| \leq W(c_0) \quad \blacksquare$$

만일 이러한 bounding 함수 $W()$가 존재하지 않는다면 이것을 complex system(복잡계), 또는 critical system이라고 볼 수 있다. 예를 들어 복권번호가 단 한자리 다를지라고 그 결과는 전혀 다르게 나타난다든지, 삶과 죽음도 바로 이런 bounding 함수가 존재하지 않는 전형적인 복잡계(complex system)라고 볼 수 있을 것이다.

10.5 Binary Indexed Tree(BIT)

어떤 구간 또는 범위에 포함된 개체의 수를 확인하는 작업은 매우 자주 요청되는 문제이다. 예를 들어 특성 시간 구간 동안에 통화한 사용자의 수라든지, 접속한 인원을 파악하는 일 등이다. 이 문제를 2차원으로 확장하면 다음과 같이 변형될 수 있다.

만일 입력 데이터가 고정되어 있다면 accumulation 방법으로 $O(1)$만에 특정 구간의 합을 쉽게 구할 수 있다. 그런데 문제는 시간별로 이 데이터가 변할 경우이다. 일단 삽입과 삭제는 제외하고 n개의 데이터의 weight w_i가 시시각각 변할 때 이를 고려하여 range sum query를 처리하는 방식의 자료구조를 만드는 것이다.

문제 10.5.1 사람들의 통화기록이 있다. 그리고 각 사람들은 각각 다양한 종류의 통화상품을 사용하고 있다. 통화상품은 *0*부터 *100*까지로 분류된다. 우리는 특정 시각 $[t_l, t_r]$ 동안 *30*이상 *75*이하의 상품급을 사용한 사용자의 수를 모두 *report*하고 싶어 어떻게 하면 이 작업을 $O(1)$에 할 수 있는지 생각해보자. □

10.5.1 BIT의 기본 아이디어

문제 10.5.2 *2*차원 배열 *D[i][j]*가 있다. 우리는 이 배열의 한 *sub*구간의 합을 전처리 없이, *2*차원 전처리 공간으로 기록할 때의 *complexity*를 구하시오.

$$\sum_{i=n}^{m}\sum_{j=r}^{s} D[i][j]$$

□

문제 10.5.3 만일 *2*차원 *accumulated array*로 구성한다면, 어떤 원소의 값이 바뀔 때 얼마나 많은 원소값을 *update*해야할 지 계산하시오. □

아래 그림과 같이 전체 데이터에서 몇 개의 대표(representative) 데이터로 그것을 요약한다고 할 때 하나의 원소가 update될 때, 그들 중 몇개는 새롭게 update되어야 한다. 많은 경우에는 전체가 모두 update되어야하고 (linear accumulation의 경우) D[1]이 수정되면 모든 누적 값을 고쳐야 한다. 이것을 좀 더 효율적으로 할 수 없을까 하는 것이 주요 연구 주제이다.

3	2	7	2	8
0	5	1	1	0
7	4	2	0	5
2	1	6	8	2
7	2	1	0	8

$D[i][j]$

3	5	12	14	22
3	10	18	21	29
7	21	31		
9	24			
16	31			

$A[i][j]$

Figure 15: 어떤 2차원 배열과 그 배열의 submatrix의 합이 누적된 추가 배열

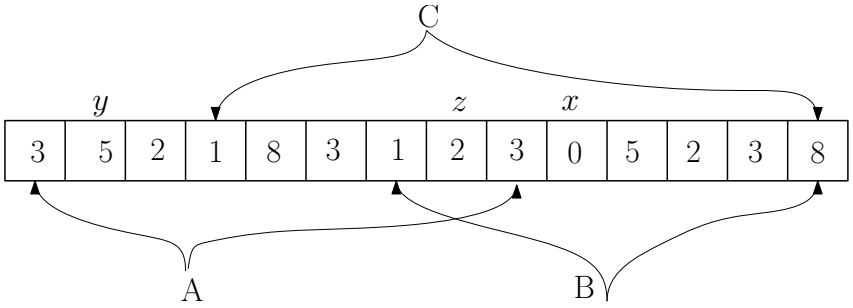


Figure 16: 자료가 있을 때 특정 원소는 데이터의 일부에 대한 대표정보를 가지고 있도록 설계한다.

10.5.2 BIT의 구조와 알고리즘

만일 전체를 대표하는 값을 $O(n)$ 개를 선정하고 그 중 하나의 값이 update될 때 $O(\log n)$ 개의 값을 update할 수 있고, 처음부터 i번째까지의 값의 합을 구하는 작업을 역시 $O(\log n)$ 만에 할 수 있다면 상당히 유용한 자료구조가 될 것인데 Binary Indexed Tree는 이것을 가능하게 하는 구조이다. 먼저 Skip List를 이용해서 이 작업을 하는 과정을 살펴보자.

1부터 $n = 2^k$ 까지의 자연수를 다음과 그림과 같이 계층적인 그룹으로 나누고자 한다. 이러한 계층적 구조를 Binary Index Tree (BIT) 라고 한다. 예를 들어, 아래 그림은 1부터 16까지의 자연수를 계층적 그룹으로 나눈 예이며, 가장 위 그룹은 1부터 16까지의 모든 정수를 포함하는 그룹이고, 가장 아래 그룹은 각각 정수 1, 3, 5, 7, 9 , 13, 15 의 한 개 정수 만을 가진 그룹들이다.

만일 특정 index까지의 합을 구하고자 한다면 우리는 $\log n$ 개의 대표 index를 access 함으로서 구할 수 있다. 각 대표 index의 partial sum을 구하는 것도 그 index 값이 W 라면 $O(\log W)$ 개의 subindex를 이용해서 구할 수 있기 때문에 이 binary indexed tree를 구성하는 시간은 $O(n)$ 만에 구할 수 있다. leaf node의 갯수가 n 개 이며 그 위

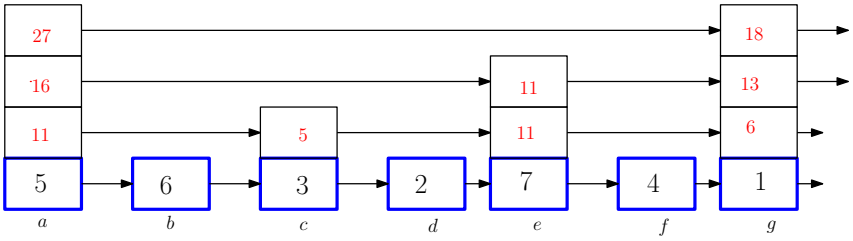


Figure 17: Skip List를 이용해서 특정 구간을 합으로 구하고 update하는 방법을 생각해보자. 만일 c가 삭제된다고 할 때 어떻게 수정하면 되는지 생각해보자.

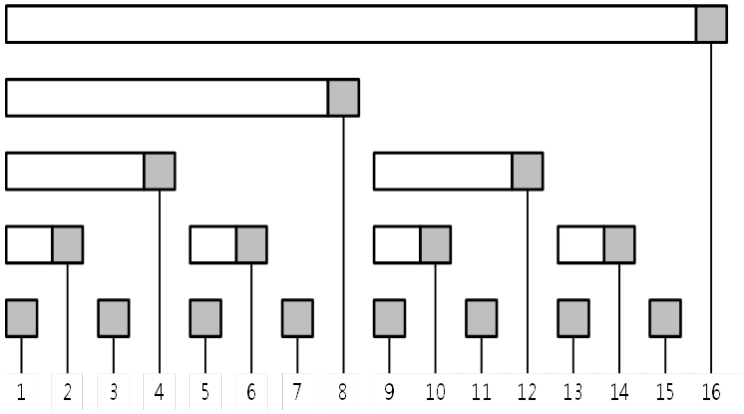


Figure 18: Binary Indexed Tree. shading된 cell은 해당 연속 subarray에 들어있는 값의 합을 기록하고 있다.

internal node의 갯수 역시 $O(n)$개 이다. 이 시간은 일반적인 heap을 구성하는 heapify 시간과 동일하며 아래와 같이 계산된다. 즉 $\log n$ 개의 partial sum 이 필요한 원소의 수는 1개이며, $\log n - 1$개가 필요한 원소는 root 아래인 노드 즉 2개이며 $\log n - i$개가 필요한 원소는 2^{i-1}개 이므로 아래와 같이 계산된다.

$$O() = \sum_{i=1}^{\lceil \log n \rceil} 2^{i-1} \cdot \lceil \log(n-i) \rceil$$

10.6 Priority Search Tree

앞에서 설명한 탐색 문제는 특정하게 지정한 공간에 존재하는 모든 데이터를 찾아내는 동작을 지원한다. 그런데 만일 그 작업에서 전체 데이터가 아니라 가장 높은 우선순위 데이터만을 보고하라고 한다면 이것은 기존의 우선순위 큐의 동작과 range searching을 동시에 지원해야만 한다.

물론 앞서의 range searching으로도 가능하지만 이것은 단 하나의 highest priority

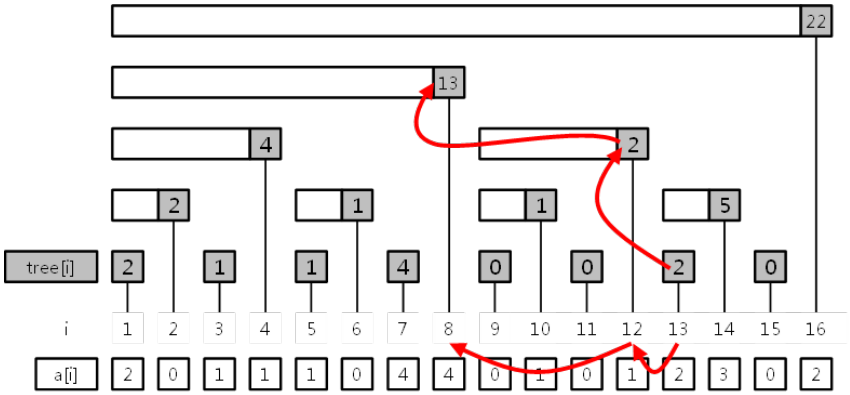


Figure 19: Binary Indexed Tree. shading된 cell은 해당 연속 subarray에 들어있는 값의 합을 기록하고 있다.

원소만을 원하는 작업에 비하여 과한 동작을 하는 셈이다. insert를 허용하는 조건에서 sorting와 min heap의 성능이 비교하여 생각하면 된다. 이 작업을 가능하게 하는 자료구조가 우선순위 탐색 트리(Priority Search Tree)이다. 이 트리 PST는 McCreight에 의해서 제안된 것으로 binary search tree와 priority queue의 hybrid 형식이라고 할 수 있다. 기본적인 아이디어는 구간을 x축, priority를 y축으로 놓고 이것을 2차원 공간에서 특정 점, 즉 구간에 있는 점들 중에서 가장 높은(낮은) 점을 찾는 기하문제로 접근하는 것이다.

우선순위 탐색 트리(Priority Search Tree, PST)에서 지원하는 동작은 다음과 같다. 특정 구간 $[l, r]$에 존재하는 데이터 중에서 가장 highest priority data를 report하라는 식이다. 이 문제는 기존의 range searching tree를 이용해서 답을 할 수 있지만 이것을 이용하는 것은 비효율적이다. 왜냐하면 그 안에 전체 k개의 데이터가 존재할 때 이 모두를 검사하여 그 중에서 가장 높은 순위의 데이터를 찾아내는 것보다 적절한 priority queue를 유지하여 그 중 하나를 매우 빠른 시간에 찾아주는 작업이 가능하기 때문이다.

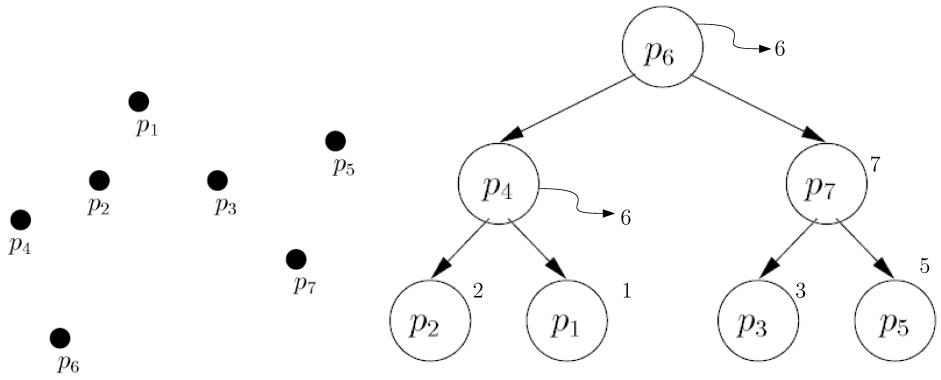
PST에 존재하는 모든 원소는 두 개의 totally ordered set, $[k_i, p_i]$, 실수의 값을 가진다. 하나는 key값이며 다른 하나는 우선순위 값이다. 이 문제에서 질문은 Query구간 $Q[l, r]\times$에 존재하는 데이터 중에서 가장 높은 우선순위 데이터 '1개'를 보고하라는 식이다. PST는 이 작업을 $O(k + \log n)$에서 k가 삭제된 $O(\log n)$에 해결하는 자료구조이다.

10.6.1 Priority Search Tree의 구성

PST는 binary tree이며 모든 노드 v는 v.key와 v.aux를 가지고 있다. v.key는 해당 노드의 key값이며 a.aux는 해당 노드의 index, 해당 노드의 priority를 가지고 있다. 즉 $S = \{P_i = (x_i.p, y_i.p)\}$ 로 구성된 이차원 상에서의 점이다. 편의상 y축의 해당 data의 priority라고 할 수 있다. 우리는 어떤 x-축 구간에 포함된 자료 중에서 가장 우선순위가 '높은' 점, 기하학적으로 lowest point를 찾으려고 한다.

일단 v.key의 median을 찾아서 balanced search tree를 구성한다. 이것은 앞서의 range tree, interval, segment tree의 구성방법과도 동일하다. root r은 S의 모든 점을

나타내는 점으로 그 값은 median을 key로 한다. 즉 r.key= median 이다. 그리고 r.aux 는 S의 점 중에서 가장 y축이 낮은, 즉 가장 높은 highest priority point를 가지고 있는다. 즉 $r.y = \{p_i.y, p_i \in S\}$



(a) 7개의 점 (Key,Priority) (b) 대응하는 PST

Figure 20: 입력 점과 그에 대응하는 Priority Search Tree의 한 예

이 작업은 $O(n)$에 처리할 수 있는데 balanced tree를 만들기 위하여 median을 찾는 과정에 가장 높은 우선순위 점을 찾아서 연결할 수 있기 때문이다. 그 단계는 모두 $O(\log n)$에 가능하게 되고 이 작업은 recursive 처리되므로 recursion 식 $P(n) = 2P(n/2) + n$에 의해서 전체 처리 시간은 $O(n \log n)$이 된다.