Lecture 11: 범위탐색 자료구조 Part II
- Interval Tree, Segment Tree -

11.1 Range Tree 예제 풀이

지난 시간에 일부 설명한 Range Tree에 대하여 몇 가지 보충 문제를 풀어보자.

문제 11.1.1 *1*차원 점 집합 a_i가 있고 구간이 주어질 때 이것을 *binary search*로 하는 방법을 제시하고 시간 복잡도를 보이시오. □

문제 11.1.2 위 일차원 *sorted array*로 *Range query*를 처리하는 방식과 *Range Tree*를 이용하는 것의 시간 차이는 없다. 두 자료구조의 차이는 어디에 있는지 설명하시오. □

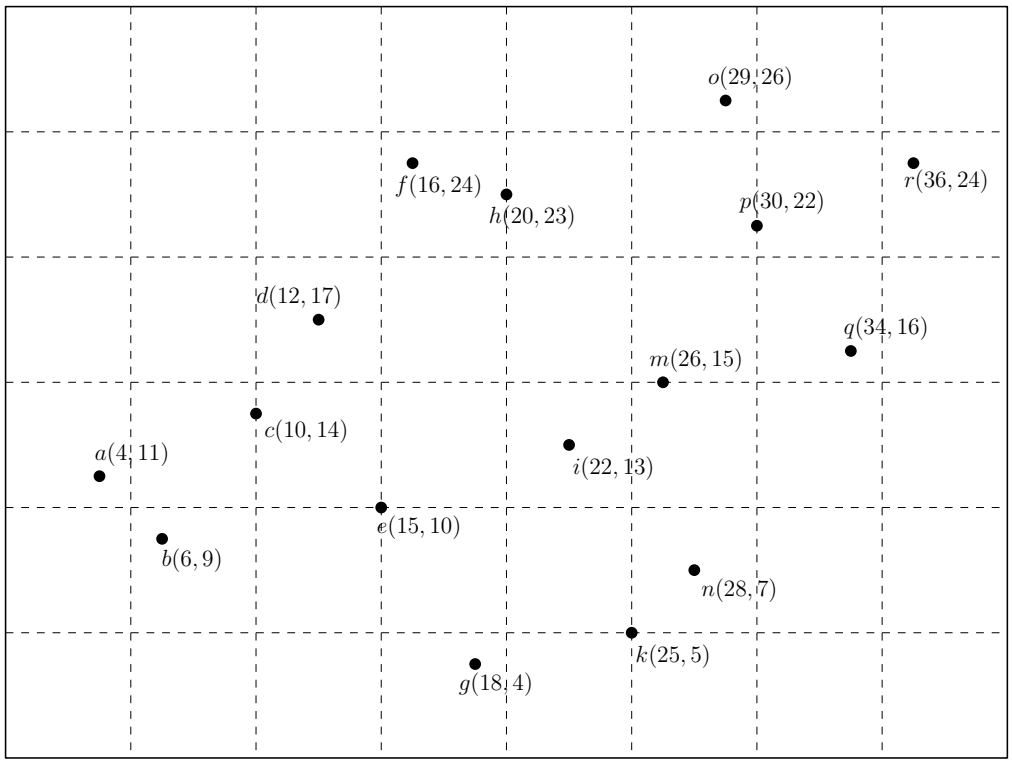
문제 11.1.3 *16*개의 *2*차원 점 집합 S가 다음과 같다. S = { *(1,16), (2,8), (3,1), (4,10), (5,13), (6,14), (7,2), (8,11), (9,9), (10,7), (11,3), (12,15), (13,12), (14,4), (15,6), (16,5)*} 이 점을 x축으로 정렬한 배열 $xsort[]$ 에 저장하고 y축으로 정렬하여 $ysort[]$에 저장해서 *query rectangle*에 포함된 점을 모두 보고하시오. 만일 점의 수가 n개라고 한다면 이 작업의 복잡도를 n의 함수로 제시하시오. *(*두개의 배열을 같은 수끼리 서로 *link*로 상호 표시한 상황이라고 가정해도 좋다.*)* □

문제 11.1.4 위 문제-*11.1*에서 어떤 *2*차원 점이 새롭게 삽입되거나 기존의 점이 *update* 될 때 위 *2*개의 일차원 배열 방법으로는 어떤 점이 문제가 되는지를 설명하시오. □

문제 11.1.5 *16*개의 *2*차원 점 집합 S가 다음과 같다. S = *{ (1,16), (2,8), (3,1), (4,10), (5,13), (6,14), (7,2), (8,11), (9,9), (10,7), (11,3), (12,15), (13,12), (14,4), (15,6), (16,5)}* S를 사용하여 *2*차원 *Range tree*를 그리시오. 이 트리를 이용해서 $[3,7] \times [8,13]$의 구간에 존재하는 점을 *report*하는 과정을 보이시오. 이 작업을 위의 *2*개의 일차원 배열을 이용하는 방법과 시간적으로 비교해 보시오. □

문제 11.1.6 위의 *2*차원 *Range Tree*에서 y-축을 *fractional cascading*을 이용할 수 있도록 *sorted array*로 구현해 보시오. 이제 우리는 y축으로 또 다른 *range tree*를 구성하지 않고 y축 값으로 정렬된 단순히 *sorted array*를 가지고 있다. □

문제 11.1.7 다음 점 집합을 이용해서 *Range Tree*를 구성하고 *fractional cascading* 과정을 보이시오.



11.2 구간 탐색 문제의 예

이번 장에서는 균형 트리를 이용하여 특정 순서의 key를 찾는 문제나 특정 원소에 대한 탐색의 빈도가 가장 최근에 탐색된 key(most recent)에 집중될 경우 이런 동작을 위하여 특별히 고안된 Splay 트리에 대하여 살펴본다. 이 자료구조의 기본적인 가정은 static 구조이다. 즉 어떤 자료에 대하여 우리는 충분한 시간을 가지고 preprocessing을 하고 이에 대하여 다양한 작업을 하는 것으로 가정한다.

먼저 유명 서양음악 작곡가들의 생존기간에 관한 사례를 살펴보자. 특정 연도 y-Year가 주어질 때 그 시간에 생존한 음악가를 찾아서 출력하는 프로그램을 작성하시오.

Interval	Composer	Birth	Death
A	Stravinsky	1888	1971
B	Schoenberg	1874	1951
C	Grieg	1843	1907
D	Schubert	1779	1828
E	Mozart	1756	1791
F	H. Schuetz	1585	1672

Table 1: 작곡가들의 생존연대표

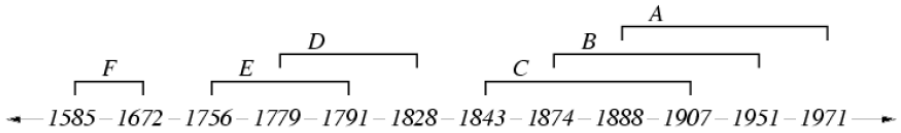


Figure 1: 6명의 음악가의 생애가 interval로 표시되어 있다.

문제 11.2.1 *(a) t-year*에 생존한 작곡가를 모두 출력하시오.

(b) 어떤 기간 *[b,e]*기간 동안 생존한 음악가를 모두 출력하시오.

(c) 특정 음악가 A와 생존 구간이 겹치는 음악가를 모두 출력하시오.

(d) 가장 많은 작곡가가 동시대에 생존한 기간을 계산해서 출력하시오.

(e) 작곡가가 하나도 없는 기간을 모두 출력하시오. *(empty space reporting)*
□

	Interval Tree	Segment Tree	Range Tree
Element	Intervals I_i	Intervals I_i	points
Query	$I_q = [l, r]$	point x_q	$I_q = [l, r]$
Storage	$O(n)$	$O(n \log n)$	$O(n)$
Construction	$O(n \log n)$	$O(n \log n)$	$O(n)$
Point Query	$O(\log n + k)$	$O(\log n + k)$	None
Interval Query	$O(\log n + k)$	$O(\log n + k)$	$O(\log n + k)$
Point Update	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Table 2: 다양한 구간처리용 자료구조의 비교. 단 표준형을 기준으로 평가함.

문제 11.2.2 매 *5*초마다 거래된 주식량이 s_i가 *10*년간 저장되어 있다. 우리는 특정 시간 동안 가장 많이 거래된 시점과 거래량을 계산하려하고 이 작업을 매우 빈번하게 처리하려고 한다. 이 작업을 $O(1)$에 처리해주는 자료구조를 제시하고, 그 때의 *space complexity*를 밝히시오. 만일 $O(N)$의 공간만 주어진다면 어떻게 이 질의를 빠르게 처리할 지 그 방법을 제시해 보시오. 예를 들어 보자.

$$3, 6, 1, 1, 2, 9, 21, 5, 9, 1, 12, 15, 3, 5, 6, 10, 22, 5, 0, 19$$

□

문제 11.2.3 위 문제에서 s_i 데이터에 오류가 있어서 특정 시점의 값을 수정하려고 한다. 즉 $s_i = s_i + \delta$. 이 작업을 제시한 자료구조에서 어떻게 반영할지를 생각해 보시오. □

구간을 다루는 가장 대표적인 자료구조는 Interval Tree와 Segment Tree이다. 이 둘을 간단하게 비교해보자.

11.3 구간 트리(Interval Tree)

구간트리 주로 시간정보(temporal information)를 가진 데이터를 처리할 때 보편적으로 활용되는자료 구조이다. 그 뿐만 아니라 시간정보와 같이 기하공간의 특징을 가진 자료를 다루는 계산기하문제에서 활용되는 매우 중요한 자료구조이다. 유의해야 할 사항은 사람마다 interval tree을 다르게 정의하고 있다. 따라서 이것을 잘 확인해야 한다.

Interval Tree(version 1)은 Binary Search Tree의 일종으로 elementary interval로 만들어진 complete tree이다. 그리고 각 노드는 해당되는 구간의 정보를 최적으로 저장하고 있다. 각 노드는 interval의 한 끝 점(point)을 가지고 있다.

이 방식은 이후에 설명한 segment tree에서 다시 다루어질 것이다. 다음은 2번째 버전의 구간트리이며 많은 사람들이 이것을 interval tree라고 불린다. 1차원 직선상에서 폐구간(close interval) $[a, b]$ 은 두 개의 실수 $a, b(a \leq b)$에 대하여 두 실수 사이 (a, b를 포함하여)의 모든 실수 집합을 나타낸다. 두 구간 $[a, b], [c, d]$ 가 서로 겹치는 경우는 $(a \leq d)AND(c \leq b)$ 인 경우이다. Interval tree 는 다음과 같은 연산을 지원하는 자료구조이다. version-1의 구간트리는 recursive하게 구성될 수 있다. 입력은 n개의 interval $I_i = [l_i, r_i]$라고 가정하자.

Step 1. 각 점에 대하여 median point를 찾는다. 그 점을 $v = x_i$라고 하자.

Step 2. 트리 T의 root에 해당하는 노드 r_T에는 v점을 포함하고 있는 모든 구간을 넣는다.

Step 3. v에는 추가 정보인 두 개의 배열 v.Left[], v.Right[]을 준비한다.

Step 4. 이 두 배열에 r에 포함된 구간의 왼쪽 끝 점을 v.Left[]에 정렬하여 넣고

Step 5. r_T에 포함된 구간의 오른쪽 끝 점을 v.Right[]에 정렬하여 넣는다.

Step 6. 전체 $\{I_i\}$ 에서 r_T에 포함되지 못한 구간 중 r_T의 왼쪽에 있는 구간은 x에 disjoint하다. 이 구간집합에 대하여 recursive하게 새로운 Interval tree를 만들고 이것을

Step 7. r_T의 left subtree로 매단다. 이 작업을 오른쪽에서 disjoint interval에도 적용하여 그것을 right subtree로 매단다.

문제 11.3.1 위에서 제시한 *interval tree*를 구성할 때 그 때의 *time complexity*를 구하시오. □

root node에 포함되는 interval을 구할 때 걸리는 시간은 median point를 구하는 시간과 모든 interval을 검사하는 시간이 $O(n)$이다. 이후 나머지 N/2개의 두 점집합에 대하여 recursive하게 구성해야 하므로 그 복잡도는 다음과 같으므로 전체 복잡도는 $O(N \log N)$이다.

$$T(N) = 2 \cdot T(N/2) + N$$

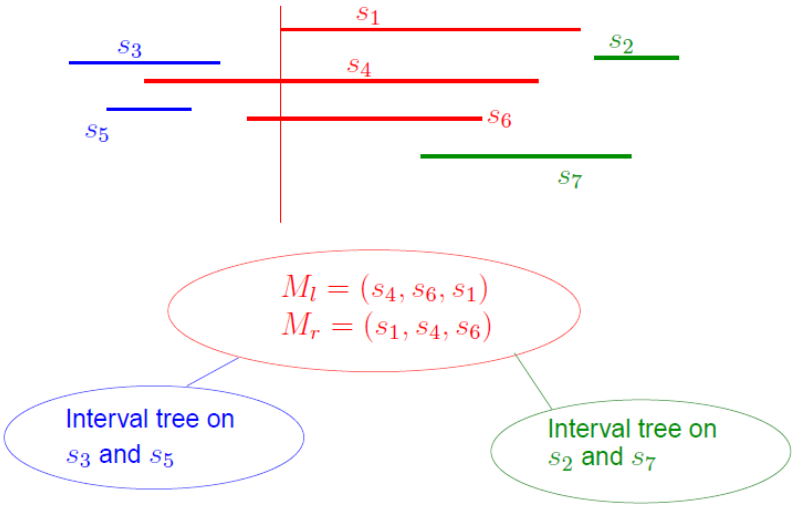


Figure 2: 구간을 분리하는 방법. 중앙 line l_x를 그어 그것과 겹치는 것을 한 노드에 담고 그것과 겹치지 않는 것 중에서 왼쪽에 있는 구간들은 모두 왼쪽 Interval Tree가 되고, 오른쪽은 오른쪽 Interval Tree가 된다. root는 median을 선택하기 때문에 전체 tree는 balanced된다.

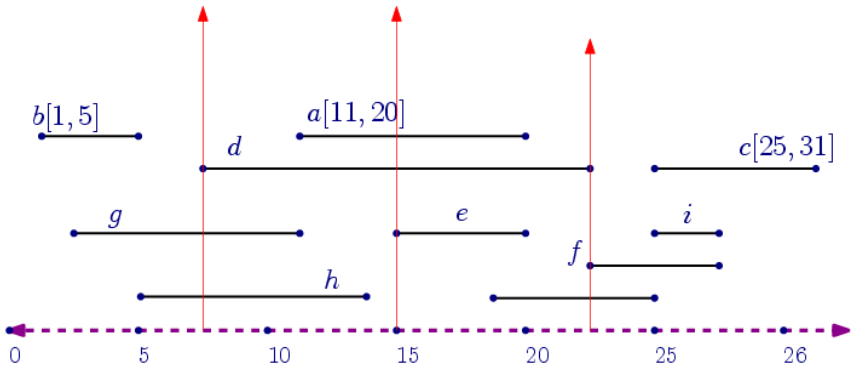


Figure 3: 10개의 interval로 Interval Tree 구성하기. median point를 찾아서 시작한다.

문제 11.3.2 다음 그림에 제시된 *sample* 구간에 대하여 *version-1* 방식으로 *Interval Tree* T_a를 구성하시오. □

11.3.1 Interval Tree의 동작

Interval Tree를 이용하는 가장 중요한 목적 중의 하나는 stabbing query를 처리하는 것이다. Stabbing[1] Query는 주어진 x_q에 대하여 이것을 포함하는 interval을 보고하는 것이다. 주의해야 할 것은 max stabbing 값을 출력하는 것은 아니라는 것이다. static version에서 max stabbing number는 고정되어 있기 때문에 preprocessing 시간에 처리될 수 있으므로 쉽게 처리될 수 있다.

문제 11.3.3 n개의 구간으로 구성된 *interval tree version-1*이 있다. *Query Interval* `Q=[l,r]`이 주어질 때 이 구간과 겹치는 모든 구간을 *report*하는 알고리즘을 제시하고 실제의 예를 트리-*11.3.2*를 활용하여 동작과정을 보이시오. 그리고 이 때의 *time complexity*를 *output sensitive* 복잡도로 제시하시오. □

문제 11.3.4 일차원 *interval*이 n개 주어져 있다. 이 *interval*의 *max stabbing number*를 구하는 알고리즘을 구하시오. 만일 이 *interval*의 *end points*들이 *sorting* 되어있는 상황에서 이 문제를 다시 풀어보시오. □

Hint) 어떤 자료구조를 사용하지도 않고 간단하게 scanning만으로 max stabbing number를 구할 수 있다.

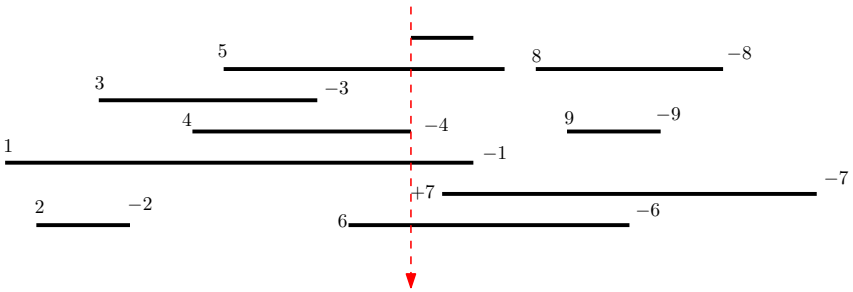


Figure 4: max stabbing number 구하기. 왼쪽부터 scan 하면서 새로운 구간의 왼쪽 점 $+x$이 입력되면 +1을 하고 오른쪽 끝 점 $-x$가 나가면 -1을 더한다.

문제 11.3.5 만일 그 *max stabbing interval*을 모두 *enumerate*해야 한다면 어떻게 해야 할 것인지 그 알고리즘을 제시하시오. □

[1] 어떤 도구로 찌르는 동작을 말한다.

문제 11.3.6 위 *2* 문제를 *2*차원으로 확장해서 풀어보시오. 죽 공간에 *recti-linear box* 가 있다. 색종이를 생각하면 된다. 이 중에서 가장 많은 종이들이 겹친 갯수를 출력하는 알고리즘을 제시하고 그 때의 시간 복잡도를 제시하시오. 하나의 *max stabbing number* k, 하나만을 출력하면 된다. □

이 point stabbing query가 확장된 interval query도 생각해볼 수 있다.

문제 11.3.7 n개의 구간으로 구성된 *interval tree version-1*이 있다. *Query Interval* `Q=[l,r]`이 주어질 때 이 구간과 겹치는 모든 구간을 *report*하는 알고리즘을 제시하고 실제의 예를 트리-*11.3.2*를 활용하여 동작과정을 보이시오. 그리고 이 때의 *time complexity*를 *output sensitive* 복잡도로 제시하시오. □

구간트리는 동적으로 관리될 수 있다. 즉 새로운 구간이 추가되거나 기존의 구간이 삭제될 수도 있다. version-1 구간트리의 특성을 보면 특정 구간은 반드시 하나의 노드에 unique하게 배정된다. 또한 하나의 노드에는 하나 이상의 interval에 포함된다. 이 사실을 이용하면 insertion도 쉽게 할 수 있다.

입력 구간이 $Q = [\ l_Q, r_Q\ ]$이라면 root node v의 point $v.value$부터 검사를 한다. 만일 이 점이 Q 구간 안에 포함되면 이 구간을 v에 추가해서 넣는다. 만일 그렇지 않다면 이 구간은 v의 왼쪽 또는 오른쪽에 들어가야 하므로 이 과정을 recursive하게 적용하면 된다. 만일 이렇게 추가하게 되면 tree의 높이가 깨어질 수 있다. 예를 들어 한쪽으로만 insert만 일어나면 tree가 skew될 수 있다. 이 경우 tree rotation을 통해서 (Red-Black Tree와 같이) 높이를 조정한다. 이 rotating 작업을 해도 여기에 좌우받는 부가정보는 없기 때문에 별 문제없이 처리된다. 전체적으로 red-Black tree로 구성한다면 높이를 $O(\log n)$을 유지할 수 있다.

특정 구간을 구간트리에서 삭제하는 과정은 좀 복잡하다. 왜냐하면 특정 점이 median point일 때 이것을 제거하면 새로운 median이 나타나기 때문이다. 만일 어떤 구간을 트리에서 삭제하면 해당 구간을 포함한 노드를 삭제될 수 있고 이 경우 복답한 경우를 다루어야 한다. 이 경우는 단순 이진탐색트리에서 노드를 삭제하는 것 보다는 복잡한다.

11.3.2 다차원 공간에서 Stabbing Query

이것을 2차원 Stabbing Query로 확장할 수 있다. 2차원 Stabbing Query는 주어진 점 (x, y)에 대하여 이것을 포함하는 rectilinear 사각형을 보고하는 것이다. 앞서의

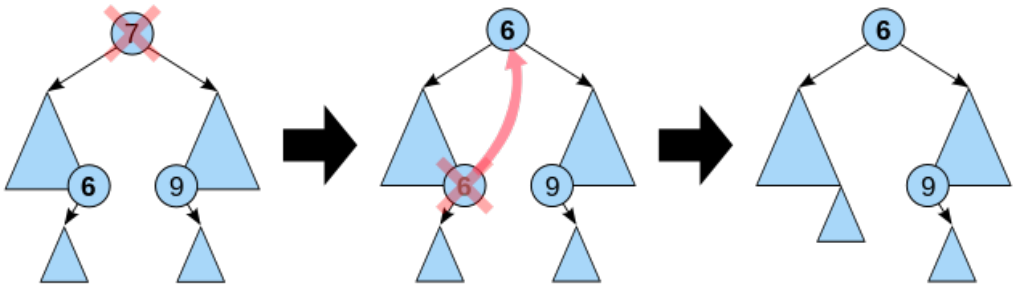


Figure 5: Interval Tree에서 이미 삽입된 특정 구간을 제거하기 (rightmost node in the left subtree).

일차원 처리방법을 확장하면 Query time은 $O(\log^2 n + k)$: $\log n$에는 한 쪽 차원 트리를 내려가는데 걸리는 시간이고 각 노드에 대하여 $O(\log n + k_v)$ 작업을 해야하므로 이 둘을 곱해야 한다. 만일 factional cascading을 하면 $O(nlogn)$ 공간이 필요하다. 즉 각 interval은 하나의 노드에만 포함되므로 전체 공간은 $O(nlogn)$이 소요된다. 만일 Priority search trees를 사용한다면 전체 공간은 $O(n)$으로 축소될 수 있다.

다차원의 경우에는 $(d-1)$-차원의 문제로 recursive하게 적용하면 되기 때문에 각각 $\log n$번의 작업이 필요하다. 따라서 d차원의 경우 다음과 같다.

(a) query time: $O(\log^d n + k)$, k는 output 크기

(b) space: $O(n \log^d n)$

(c) preprocessing time: $O(n \log^d n)$

11.3.3 구간트리 version 2 : 삽입과 삭제

구간을 삽입하고 삭제하는 작업에서 유의해야할 것은 새로운 점을 추가하는 것이 아니라 이미 결정된 point set으로 새로 이루어지는 interval을 삽입하는 과정을 말한다. 삭제하는 작업의 경우에는 이미 존재하는 interval을 찾아서 삭제하는 작업을 말한다. 즉 삭제할 구간 $[a, b]$을 주는 것이 아니라 이미 삽입된 특정 구간 $I_x = [a, b]$을 삭제해야 한다.

이 버전으로는 주어진 점 p을 포함하는 모든 interval을 구할 수 있으며 동시에 새로운 interval의 추가와 기존 interval의 삭제가 $\log n$에 가능한 특징을 가지고 있다. version 1과 비교한다면 insertion seletion이 가능한 구조이다.

11.3.4 구간트리 version 3 : Total Ordering

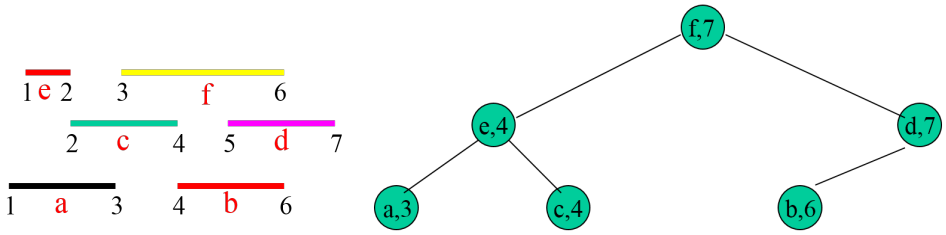
이 버전은 모든 포함 구간은 아니지만 하나의 구간을 찾을 수 있는 간결한 구조이다. 우리는 모든 서로 다른 n개의 구간에 대하여 이것을 순서대로 완전히 정렬하려고 한다. 이것은 Red-black tree로 구성하며, 각 노드는 정확하게 하나의 구간 v.int를 저장하고 있으며 어떤 point v.max는 v.max = max { v를 root로 하는 모든 subtree의 interval

중에서 오른쪽 좌표 점 }. 이것은 결국 binary search tree를 intervals에 대하여 구성하는 것이며 추가 삭제, 삽입될 때 적절히 rotation을 통하여 균형을 잡는다.

정의 11.3.1 두 개의 구간 $X = [l_x, r_x]$, $Y = [l_y, r_y]$가 있을 때 이 두 구간의 순서는 $X < Y$ 으로 결정된다. *if and only if* $l_x < l_y$ 또는 만일 $l_x = l_y$ 이면 $r_y > r_x$ ■

즉 두 구간의 왼쪽 끝이 먼저 나오면 그 순서도 빠르다. 만일 왼쪽 시작점이 같으면 오른쪽 점 r이 큰 쪽이 더 큰 구간, 즉 뒷 순서가 된다. 이렇게 구성하면 모든 노드는 하나의 interval을 가지고 있게 되므로 tree node의 수는 interval의 갯수와 동일하다. 단 주의할 점을 RB-tree에서 rotation을 할 경우 v.max가 변화한다는 것이다. $B \leftarrow A$ 를 $A \rightarrow B$로 rotate할 때 각 노드의 max 값은 다음과 같이 update되어야 한다. 또 이 경우 max field가 변하는 노드는 rotate되는 두 노드 A,B 밖에 없다. Max value는 오로지 A와 B에 대해서만 바뀌므로 다음을 알 수 있다.

A.max = max{A.interval.right, BR.max, AR.max}. B.max = max{B.interval.right, B'L.max, A.max}.



(a) 6개의 interval (b) version 3으로 구성한 구간 트리

Figure 6: 각 노드는 두 가지 값을 가지고 있다. (해당 interval ID, v.max value)

문제 11.3.8 다음에 그림에 제시된 *10*개의 구간을 이용하여 *version 3*에 기초한 *interval tree(balanced)*를 구성하시오.

문제 11.3.9 이 *version 3 interval tree*에서 주어진 구간 $Q = [l, r]$에 대하여 이것을 포함하는 **어떤 하나의** *interval*을 출력하는 프로그램을 작성하시오.

Hint) `Q = [l,r]`와 겹치는 **하나의 구간**만을 찾으려고 한다. 어떤 것이라도 좋다. tree의 root는 v이다. 만일 v.interval와 Q 가 겹치면 done. Otherwise, if v.leftChild.max $\geq l$, 그러면 v.leftChild 쪽 tree에서 작업을 계속한다. 만일 그렇지 않다면 v.rightChild 쪽 subtree에서 작업을 계속한다. 이 탐색 시간은 $O(\log n)$.

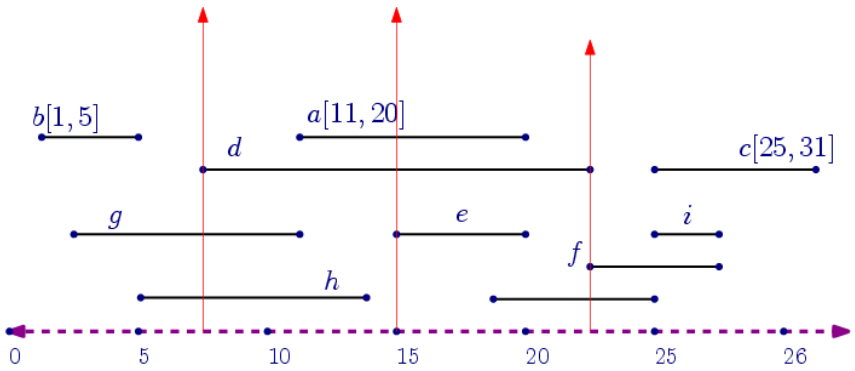


Figure 7: 10개의 interval에 대하여 전순서 매기기(total ordering).

문제 11.3.10 이 *version 3 interval tree*에서 특정 구간 $[a, b]$ 이 존재하는지 검사하는 알고리즘을 제시하시오.

문제 11.3.11 이 *version 3 interval tree*에서 새로운 구간 *(node)*를 추가하거나, 삭제하는 알고리즘을 제시하고 그 때의 시간 복잡도를 제시하시오.

위 모든 작업에서의 시간복잡도를 보면, rotation 등으로 높이를 유지하는 작업의 시간은 $O(1)$이므로 삽입 삭제에 대한 시간은 트리의 높이와 비례하므로 $O(\log n)$ 이다. 이 version에서는 겹치는 구간 중 하나만 report하므로 전체 모든 구간을 보고하기 위해서는 한번 탐색된 것은 삭제해야 한다. 따라서 만일 정답이 k개라면 k번의 탐색과 삭제(높이 조정)가 필요하므로 그 시간은 $O(k \log n)$이다. 이것은 static version의 $O(k + \log n)$ 보다는 높지만 dynamic insert/delete를 제공해준다는 면에서 의미를 가진다.

11.4 선분트리 (Segment Tree)

선분트리는 Interval Tree와 함께 계산기하학에서 활용되는 가장 기본적인 자료구조로서다음의 문제에 답을 할 수 있다. 선분트리를 구성하는데 사용되는 기본 원소(leaf node)는 아래의 elementary interval이다.

정의 11.4.1 입력 구간 $[a_i, b_i]$이 주어질 때 *elementary interval* $[x, y]$는 a_i, b_i를 모두 정렬하여 만들어진 새로운 수열 r_i, $1 \le i \le 2n$에서 $[r_k, r_{k+1}]$을 말한다.

문제 11.4.1 n개의 *interval* $I_i = [l_i, r_i]$이 주어질 때 *elementary interval*의 최대 갯수, 최소 갯수를 계산하시오.

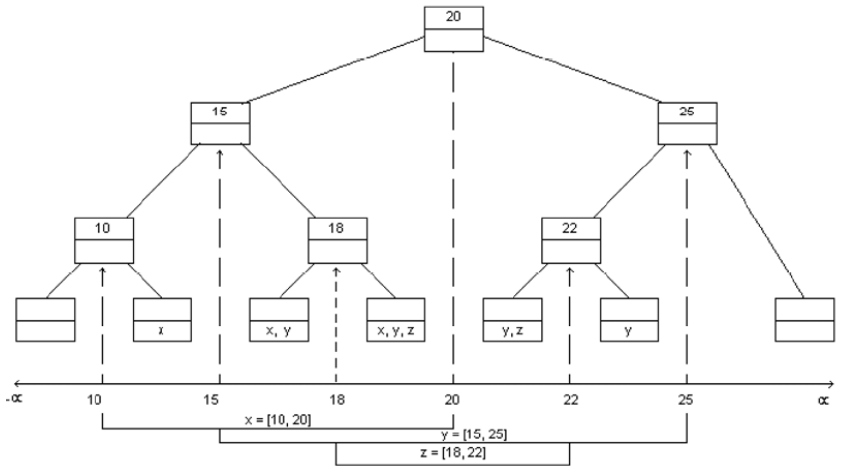


Figure 8: 단위 구간 3개를 바탕으로 만들어진 선분 트리의 예. 이렇게 구성하면 모든 elementary interval 정보가 표시되므로 각 노드에는 그 단위 구간을 포함하는 선분의 id가 linked list로 추가된다.

문제 11.4.2 n개의 선분이 주어지고 해당 *leaf node*에 그것을 포함하는 모든 선분의 *id (*예를 들면 *{ x,y,z })*를 표시한다고 하자. 이렇게 하면 모두 n^2개의 *mark*가 필요한데, 그림-**??**에서 표시된 n^2개의 정보를 최소로 하는 방법을 *minimal covering node*방식으로 제시하시오. *(range tree covering*과 같다. *)*

선분 트리(segment)의 유형은 update의 여부에 따라서 다음 두 버전이 존재한다.

(a) Static segment trees: 가장 일반적인 트리로서 주어진 크기 N인 배열을 전처리하여 트리로 만든다. 구간에 관한 질의를 대부분 $O(logN)$ 처리할 수 있다.

(b) 각 점의 update가 가능한 선분트리. 선분의 한 점을 $O(logN)$에 고칠 수 있으며 전체적으로 같은 성질을 유지한다. 구간 질의에 대하여 $O(logN)$ 시간에 답을 구할 수 있다.

(c) 가장 일반적인 선분 자체에 대하여 update가 가능한 트리이다. 하나의 선분을 삭제하거나 추가하거나 하는 작업을 최악의 경우에는 $O(N)$ 시간에 해결할 수 있

다. 그러나 특별한 optimization을 하거나 lazy operation을 동원하면 구간 질의에 대하여 $O(logN)$ 시간에 질의를 처리할 수 있다.

Segment Tree의 일반적인 구조는 다음과 같다. 앞으로 어떤 선분을 $[i,j], i<j$ 으로 표시한다. 단 i,j는 모두 정수이다.

(1) 이진트리로 구성된다.

(2) leaf 노드는 단위 구간을 나타내고 internal node는 복수의 elementary interval을 나타낸다.

(3) 각 노드는 구간 $[l,r]$을 나타낸다.

(4) 특정 구간을 cover하는 hit node들은 모두 그 선분의 id를 linked list L로 관리하고 있다.

(5) 항상 $l<r$이며 이 값은 정수이다.

(6) Root 노드는 전체 구간 [1,n]을 나타낸다.

(7) $r=l+1 \Rightarrow v$ is a leaf node (unit interval).

(8) $r > l+1$ 는 다음 아래의 두가지 경우로 나뉜다. case1) Left child range is $[l,(l+r)/2]$. case2) Right child range is $[(l+r)/2,r]$.

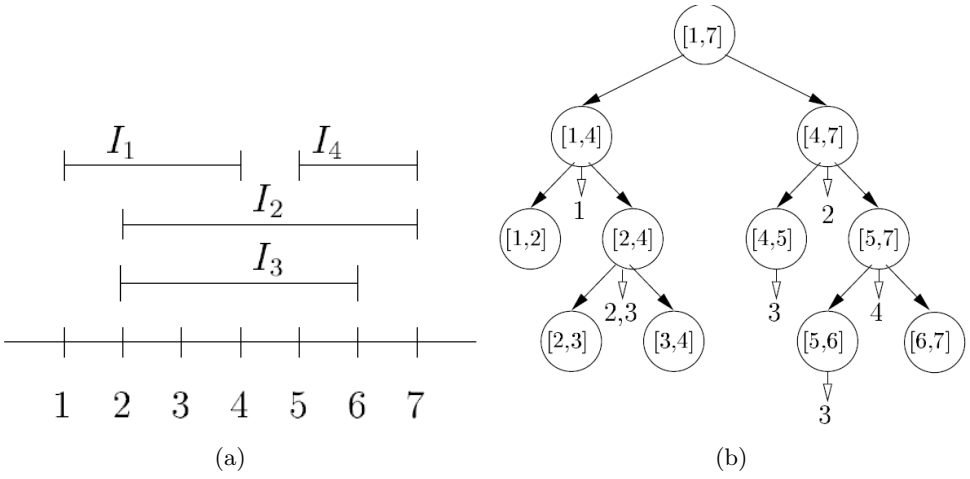


Figure 9: (a) 4개의 구간 I_i으로부터 구성된 선분 트리 (b). 노드 아래 표시된 숫자는 그 구간을 완전히 포함하는 선분의 id를 나타낸다.

문제 11.4.3 *n*개의 *elementary interval*을 가진 *segment tree*의 공간복잡도가 $O(n \log n)$ 임을 보이시오.

Solution) 일단 전체 트리 구조의 크기는 n이다. 그리고 전체 공간의 크기는 각 노드에 부가의 정보, 즉 cover node 정보까지를 고려해서 계산해야 한다. 어떤 입력 구간을 cover하는 hit node는 각 level에 최대 2개만 가능하므로 전체적으로 트리의 깊이 곱하기 2개의 정보가 추가된다. 따라서 하나의 interval에 대해서 $2\log n$) 개의 정보가 추가되므로 전체적으로 $O(n \log n)$ 개의 정보가 segment tree에 포함되어야 한다.

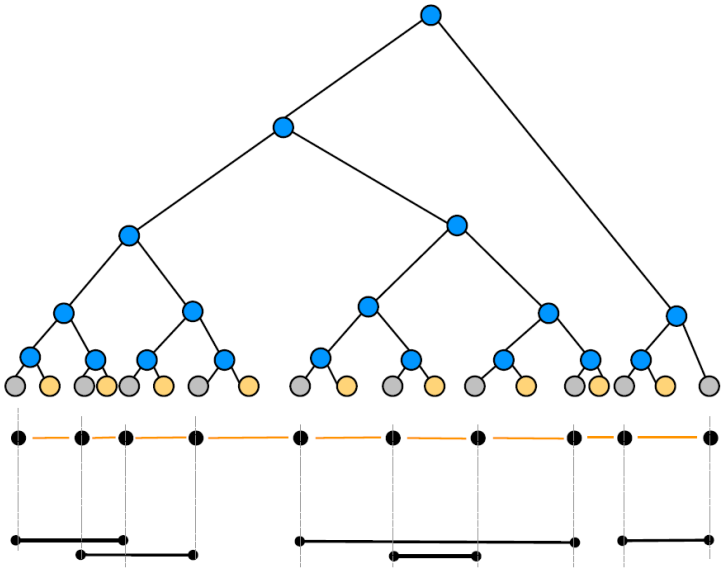


Figure 10: Segment tree의 새로운 표현. gray node는 각 segment의 양끝 점을 나타내고 오렌지 색 노드는 open interval [a,y)를 표현한다. 이렇게 표현하면 보다 다양한 질의를 처리할 수 있다.

11.4.1 선분 트리에서의 삽입과 삭제

우리는 기존의 segment의 끝 점으로 이루어진 새로운 구간을 기존의 tree에 삽입할 수 있다. 단 이 경우 반드시 standard form으로 더해야 한다. 즉 어떤 노드의 두개 모두가 marking되어있다면 그 상위노드는 mark되어야 한다. (노드에 포함되어야 한다.)

문제 11.4.4 *segment tree*에 어떤 구간 s를 *insert*할 경우 이것은 한 *level*에서 두 개 이상의 노드에 포함되지 않음을 보이시오. 즉 한 *level*에서 최대 *at most 2*개의 노드에 저장됨을 증명하시오. □

Proof) 우리는 contradiction으로 증명한다. if s stored at more than 2 nodes at level i. let u be the leftmost such node, u_0 be the rightmost let v be another node at level i containing s....

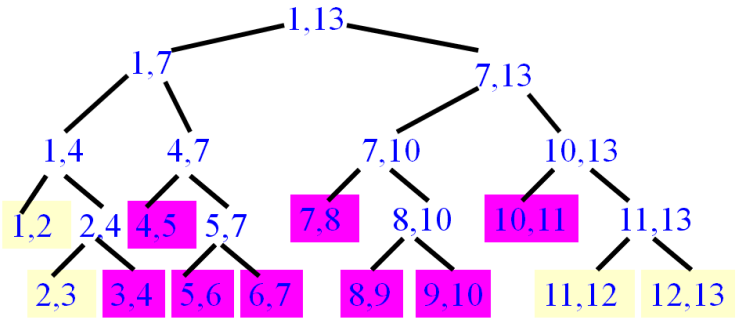


Figure 11: Segment tree에 새로운 구간 l_x을 추가하는 방법. 즉 추가된 이후에 바뀐 점은 이 구간을 완전히 포함하는 hit node에 l_x를 새로 marking해야 한다.

삽입과정의 동작의 복잡도를 분석해보자. 삽입할 때의 sliding 작업을 볼 때 공간은 $O(n \log n)$이 필요하며, 삽입(insertion)은 $O(\log n)$ 시간에 가능하다. 최대 4개의 노드가 각 level에서 만나기 때문이다. 따라서 실제 필요한 공간은 $\Theta(n \log n)$. 그리고 Query time은 $O(k + \log n)$의 시간이 필요하다. 전처리 시간을 살펴보자. 일단 전체를 정렬(sorting)을 해야하므로 $\Theta(n \log n)$의 시간이 empty segment tree에서부터 필요하다. 그리고 $O(n)$번의 insertion을 해야하므로 전체 시간은 $\Theta(n \log n)$이 걸린다.

11.5 Lazy Propagation

선분 트리를 이용하면 특정한 구간의 합을 구할 수 있다. 즉 선분 트리의 root를 그 하위 모든 원소의 총합을 가지고 있으면 특정 구간 $[i, j]$구간의 숫자 합을 특정 구간의 interval을 표시하는 노드의 합을 구하는 방식으로 쉽게 구할 수 있다.

이 자료구조에는 약간의 update를 가능하게 하는데 특정 원소의 값이 변할 때 해당 원소에서 root로 올라가면서 고치면 된다. 문제는 하나의 원소가 아니라 특정 구간 $[i, j]$에 속한 모든 원소 즉 $a_i, a_{i+1}, a_{i+2}, \cdots a_j$의 모든 값이 특정한 값 d를 더해줄 경우이다. 이 경우 모든 값에 d 더하고 이것을 root까지 가면서 고친다면 그 시간 복잡도는 $O(n \log n)$이 될 것이다. 따라서 이렇게 하지 않고 구간 $[i, j]$의 대표구간에 해당 하는 원소의 갯수 곱하기 d만큼의 값을 더해주고 이것을 root까지 propagation 시키면 일은 처리된다. 그런데 문제는 나중에 다른 구간의 합을 구할 때 문제가 된다. 이런 작업을 할 때 자주 사용하는 기법이 Lazy Propagation이다. 일단 간단한 문제를 예를 들어 살펴보자.

11.5.1 구간 최소치

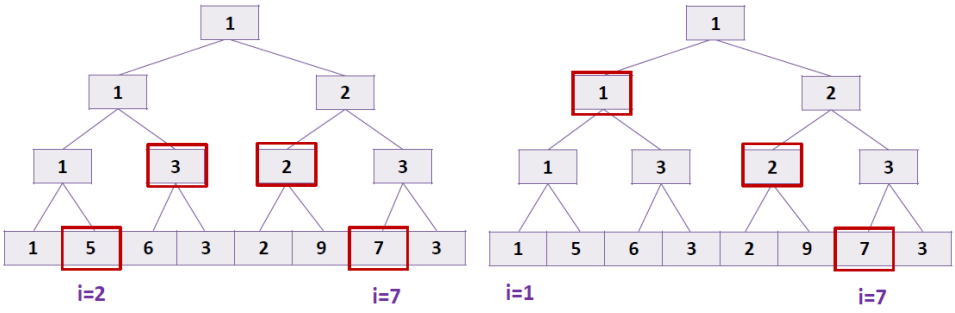
문제 11.5.1 n개의 원소 a_i가 있다. 우리는 그 원소 중에서 어떤 *index interval* $[l, r]$에 대해서 가장 작은 값 $q = \min\{a_k\}, l \le k \le r$ 을 구하는 자로구조를 제시하시오. □

이 query는 Range Minimum Query(RMQ)라고 부른다. 위 문제를 다양한 자료구조로 제안해보고 그 때의 space complexity와 time complexity를 각각 제시하고 비교해보자. 만일 $O(n^2)$개의 exhaustive static 공간을 준비한다면 $O(1)$에 할 수 있지만 하나를

고치면 $O(n)$ 개의 자료구조 원소를 수정해야 한다. 이것은 너무 큰 낭비하고 할 수 있다.

문제 11.5.2 만일 이 중간에 $a_i = a_i + d_0$로 *update*하는 것을 허용한다고 하자. 이 경우 *RMQ*를 위한 자료구조를 제시하시오. □

Hint) 각 원소를 $\sqrt{(n)}$개의 subarray로 쪼개서 뭔가의 작업을 한다면 한번 point update 할 때 얼마나 많은 변화가 필요한지 생각해보자. (양쪽의 균형을 적절히 잡아야 한다.) 또는 앞서 배운 구간 정보를 위한 트리를 이용해서 할 수도 있다.



(a) min(2,7) (b) min(1,7)

Figure 12: 특정 구간 [l,r]에서 최소값 구하기

결국 문제는 주어진 데이터의 공간을 어떻게 decompose하는가의 문제로 귀납된다. 그리고 update 시간과 Query 시간과의 균형을 맞추는 것이다. 이 방법 중에 Sparse Table 방법이 있다.

11.5.2 구간 Update하기

4개의 원소를 가진 배열의 모든 원소에 1을 더한다. 그 다음에는 마지막 두 원소(3번째, 4번째)에 +1을 더한다. 이 둘 작업후에 첫 3개 원소의 값을 계산하는 과정을 생각해보자. 이 작업을 매번 변화가 있을 때마다 배열의 원소를 update하지 않고 그것을 미루어 두었다고 처리하는 방법을 생각해보자.

우리는 Lazy Propagation Indicator(LPI)를 이용해서 구간이 부분적으로 Query될 때마다 LPI의 값을 적용하여 내부 원소를 바꿔 나가는 전략을 편다.

앞서 하나의 원소 변경, point update는 쉽게 처리할 수 있음을 보였다. 이번에는 range update문제를 살펴보자. 만일 구간 [l,r]에 대하여 특정 상수를 더한 경우 이 모두를 update해야 한다면 시간이 너무 걸리게 된다. 이 경우 hit node만을 update하면 된다. 즉 어떤 노드는 hit 노드가 되는데 그 조건은 다음과 같다.

정의 11.5.1 *segment tree*의 구간 *[l,r]*이 주어질 때 특정 노드의 *subtree*의 모든 *node*가 이 구간에 들어가면 이 노드는 *hit*노드가 된다. 그리고 *hit node*가 *child*인 노드 역시 *hit node*가 된다.

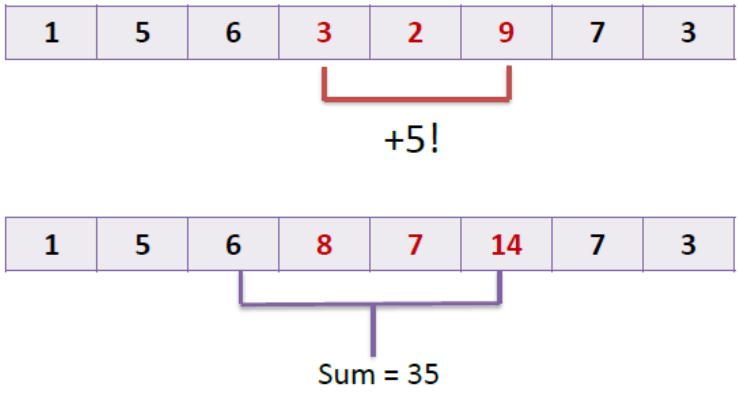


Figure 13: 특정 구간에 속한 원소에 모두 같은 값을 더하기

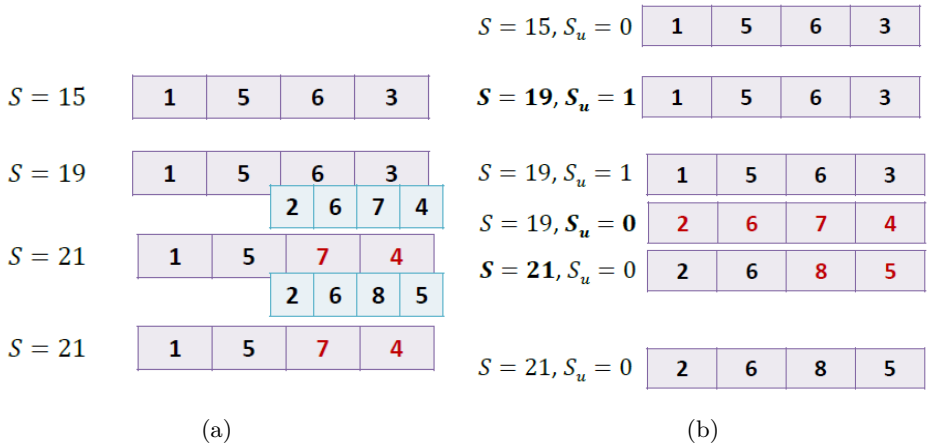


Figure 14: 4단계의 작업. LPI를 이용해서 나중에 계산하는 방법

궁극적으로 lazy propagation을 이용하면 다음의 동작을 할 수 있다.

(1) 구간 [l,r] 의 최대, 최소값 원소

(2) 구간 [l,r] 의 합.

(3) 특정 원소의 값을 update

(4) 특정 구간에 있는 모든 원소의 값을 update

(5) 특정 구간의 모든 원소값을 negate(부호반대)

이 모든 작업을 $O(Q \log n + n)$ 시간에 처리할 수 있다.

11.6 희소 테이블을 이용한 Range Minimum 계산하기

우리는 보다 빠른 시간에 특정 구간에서의 최소값을 구하는 새로운 자료구조를 만들어 본다. 전체 답의 갯수가 $O(n^2)$ 이므로 이 공간이면 항상 $O(1)$ 에 답을 할 수 있지만 우리는 그 보다 작은 $O(n \log n)$ 개의 공간으로도 이 상수 시간에 그 값을 계산할 수 있는 자료구조를 만들어 본다. 이런 자료구조를 Sparse Table이라 부른다.

어떤 크기가 n 인 일차원 배열이 있을 때 각 index i 에 대해서 우리는 연속된 2^k 개의 값의 minimum 값과 그 위치를 을 가지고 있다. 여기에서 $2^k \leq n$ 이다. 따라서 각 index i 에 대해서 모두 $\log n$ 개의 부가 정보를 가지고 있게 되고 이것을 이용해서 빠르게 최소값을 찾아낸다. 문제는 이 부가 정보를 이용해서 최소값을 빠르게 찾아내고 이후의 update도 쉽게 하고자 하는 것이다.

문제 11.6.1 *Sparse Table* 전체 자료구조의 크기를 계산하시오. □

문제 11.6.2 $min[k][i]$ 를 *index* i 에서 $i + 2^k$ 인 원소까지의 값 중에서 최소를 가지고 있다고 할 때 이것을 이전의 계산한 $min[k-1][\ ]$ 을 이용해서 로 계산하는 방법을 제시하시오. □

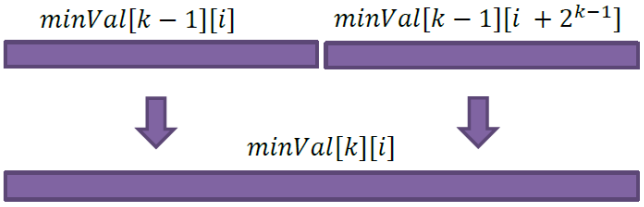
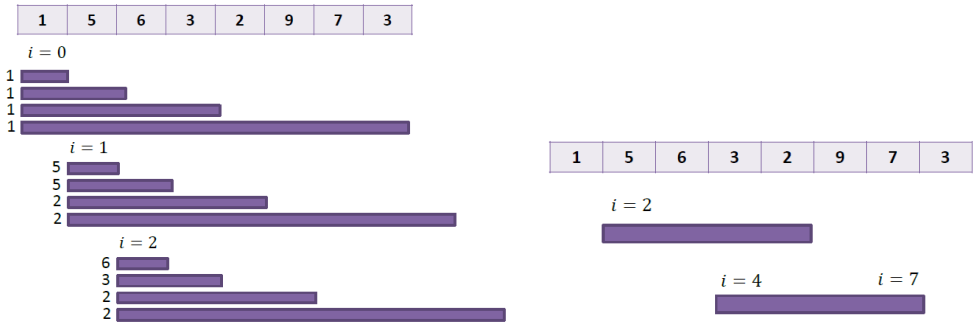


Figure 15: Sparse Table에서 `min[ ][ ]` 계산하기

우리는 길이 2^k 개의 range 구간을 이용해서 그것을 모두 카버하려고 한다. 즉 우리는 최소갯수의 range로 구간을 모두 커버할 수 있는 k 의 최대값을 구하면 된다. 따라서 이 문제는 $2^k \leq r - l + 1$ 이 식을 만족하는 k 의 최대값을 구하는 문제로 귀납된다.

이를 위해서 모든 가능한 Query 구간의 길이 [1,N] 에 대하여 미리 가장 큰 k 값을 미리 계산해둔다. 이 작업으로 최소값 질의는 $O(1)$ 에 놀랍게도 해결된다. 그리고 전처리 시간은 $O(\log n)$ 이며 필요한 공간역시 $O(\log n)$ 로 동일하다. 각 index는 각각 $O(\log n)$ 개의 부가 정보를 가지고 있다.



(a) (b)

Figure 16: Sparse Table. 질문 구간 [2, 7]은 길이가 length = 6이다. 우리는 여기에서 $k = 2$로 선택한다. 그것은 $2^k = 4$인 길이 구간의 최소갯수로 덮을 수 있기 때문이다.

최소값을 찾는데 $O(\log n)$이 걸리는 segment tree에 비해서 시간은 상수로 줄어드는 효과가 있지만 이 자료구조의 문제는 update가 불가능하다는 것이다. 불가능하다기보다 다시 $O(n \log n)$ 시간에 완전히 rebuild를 해야 한다는 점이 단점이다.

11.7 구간 자료구조의 성능비교

우리는 앞에서 설명한 4가지 자료구조인 segment tree, interval tree, range tree, binary indexed tree 를 다양한 관점에서 종합하여 다시 비교하는 것으로 이 장을 마치기로 하자.

11.7.1 1차원 공간에서의 시간, 공간 복잡도

(a) Segment tree - $O(n \log n)$ preprocessing time, $O(k+\log n)$ query time, $O(n \log n)$ space

(b) Interval tree -$O(n \log n)$ preprocessing time, $O(k+\log n)$ query time,$O(n)$ space

(c) Range tree - new points can be added/deleted in $O(\log n)$ time

(d) Binary Indexed tree - the items-count per index can be increased in $O(\log n)$ time

11.7.2 동적 관리의 복잡도

모든 자료구조는 동적으로 관리될 수 있다. 새로운 원소를 추가하거나 특정 값을 찾아서 제거하는 작업의 시간복잡도를 살펴보자.

(a) Segment tree - interval can be added/deleted in $O(\log n)$ time (see here)

(b) Interval tree - interval can be added/deleted in $O(\log n)$ time

(c) Range tree - new points can be added/deleted in $O(\log n)$ time (see here)

(d) Binary Indexed tree - the items-count per index can be increased in $O(\log n)$ time

11.7.3 다차원에서의 성능

다차원(higher dimensions) $d > 1$-차원 경우에 대하여 전처리 과정과 탐색 시간에 대해서 살펴보자.

(a) Segment tree - $O(n(\log n)^d)$ preprocessing time, $O(k + (\log n)^d)$ query time, $O(n(\log n)^{d-1})$ space

(b) Interval tree - $O(n \log n)$ preprocessing time, $O(k+(\log n)^d)$ query time, $O(n \log n)$ space

(c) Range tree - $O(n(\log n)^d)$ preprocessing time, $O(k+(\log n)^d)$ query time, $O(n(\log n)^{d-1})$ space

(d) Binary Indexed tree - $O(n(\log n)^d)$ preprocessing time, $O((\log n)^d)$ query time, $O(n(\log n)^d)$ space