Range queries 2

Binary indexed tree

- Binary indexed tree หรือ Fenwick tree สามารถมองได้เป็น prefix sum array ที่เปลี่ยนแปลงได้

- มันรองรับการดำเนินการสองอย่างที่ใช้เวลา $O(\log n)$ บน array ได้แก่การการสอบถามผลรวมแบบช่วงและการ update ค่า

- ข้อได้เปรียบของ binary indexed tree คือมันอนุญาตให้เรา update ค่าใน array ได้อย่างมีประสิทธิภาพระหว่างที่มีการสอบถามผลรวม ซึ่งไม่สามารถทำได้หากใช้ prefix sum array เพราะว่าหลังจากการ update แต่ละครั้ง มันจะต้องสร้าง prefix sum array ใหม่ซึ่งใช้เวลา $O(n)$

โครงสร้าง

- แม้ว่าชื่อของโครงสร้างจะเป็น binary indexed tree แต่เราสร้างด้วย array
- เราจะสมมติว่า array เริ่มต้นที่ index ที่ 1 เพื่อที่จะทำให้สร้างได้ง่ายขึ้น
- กำหนดให้ $p(k)$ แทนค่าของสองยกกำลังที่มากที่สุดที่หาร k ได้ เราจะเก็บ binary indexed tree เป็น array ชื่อ tree ที่

$$\mathrm{tree}[k] = \mathrm{sum}_q(k - p(k) + 1, k)$$

นั่นคือแต่ละช่อง k จะเก็บค่าผลรวมในช่วงความยาว p(k) ใน array ต้นฉบับที่สิ้นสุดที่ตำแหน่ง k

- ตัวอย่างเช่น พิจารณา array

1	2	3	4	5	6	7	8
1	8	7	1	2	9	2	6

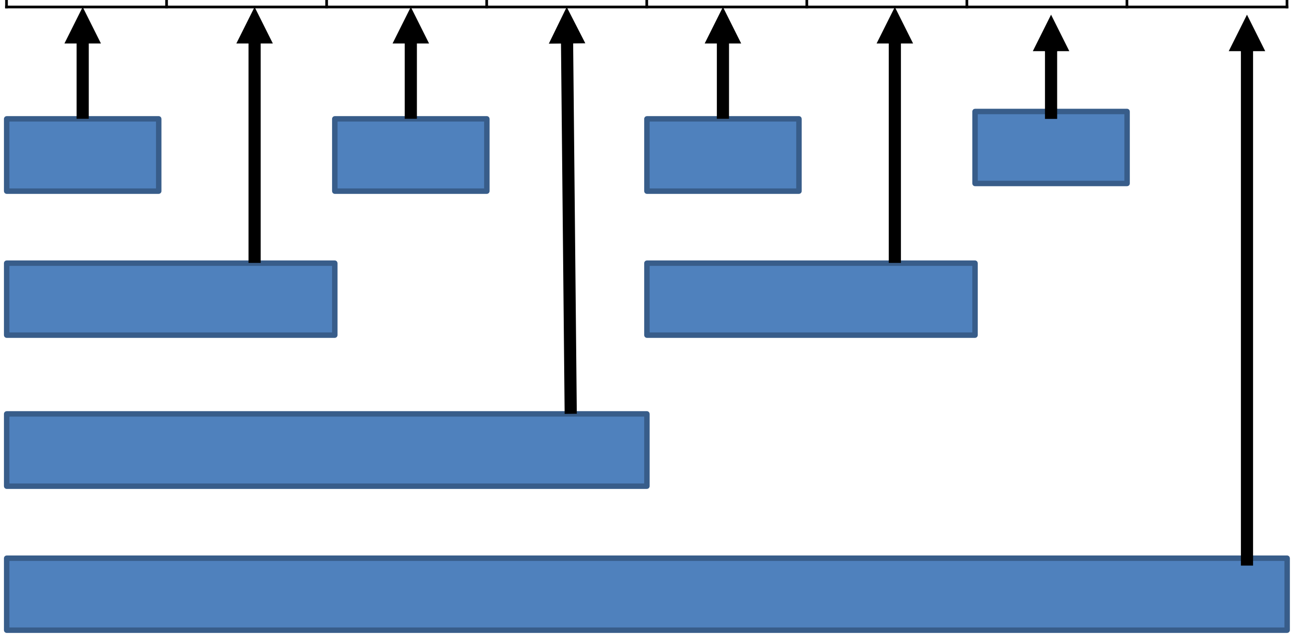
- binary indexed tree ของ array ข้างบนคือ

1	2	3	4	5	6	7	8
1	9	7	17	2	11	2	36

- ตัวอย่าง **tree[2] = sum_q(2–2+1, 2) = sum_q(1,2) = 9**
- **tree[3] = sum_q(3–1+1, 3) = sum_q(3,3) = 7**

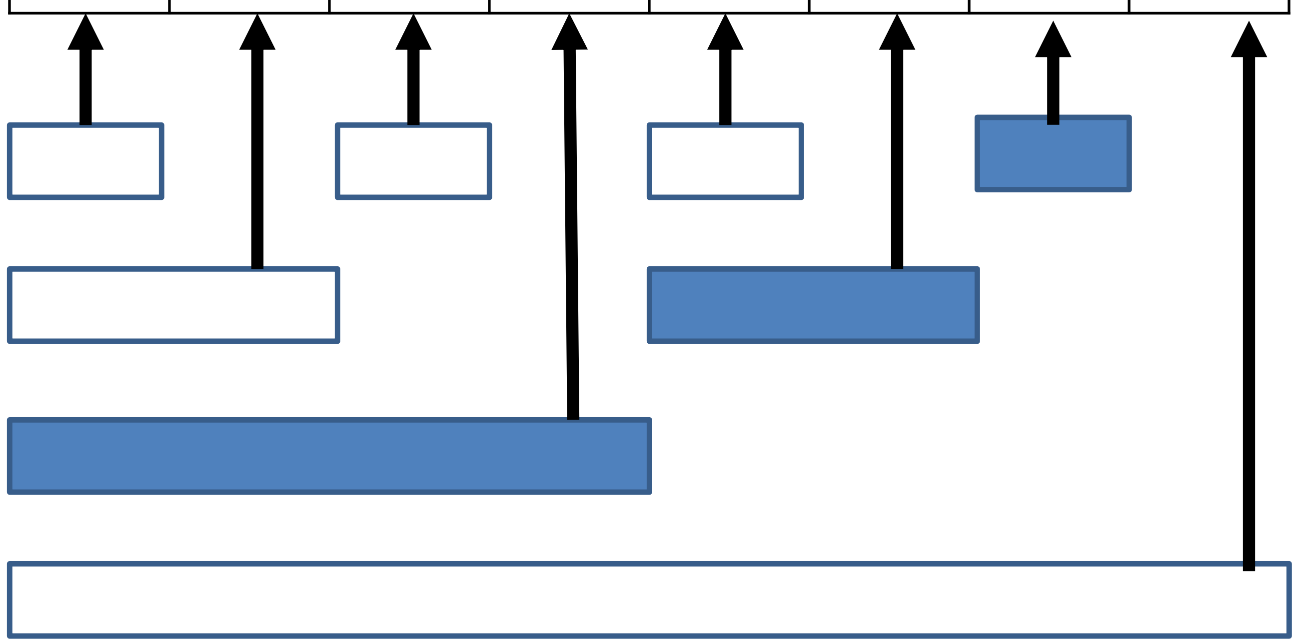
- รูปต่อไปแสดงถึงว่าค่าแต่ละค่าใน binary indexed tree สอดคล้องกับช่วงอย่างไร

1	2	3	4	5	6	7	8
1	9	7	17	2	11	2	36



- การใช้ binary indexed tree นั่น ค่าของ **sum_q(1,k)** สามารถถูกคำนวณได้ใน **O(logn)** เพราะว่าช่วง [1,k] สามารถถูกแบ่งได้เป็น **O(logn)** ช่วง ซึ่งผลรวมนั้นถูกเก็บใน tree แล้ว
- ตัวอย่างช่วง [1,7]

1	2	3	4	5	6	7	8
1	9	7	17	2	11	2	36



- ดังนั้นเราสามารถคำนวณค่า sum ได้ดังนี้

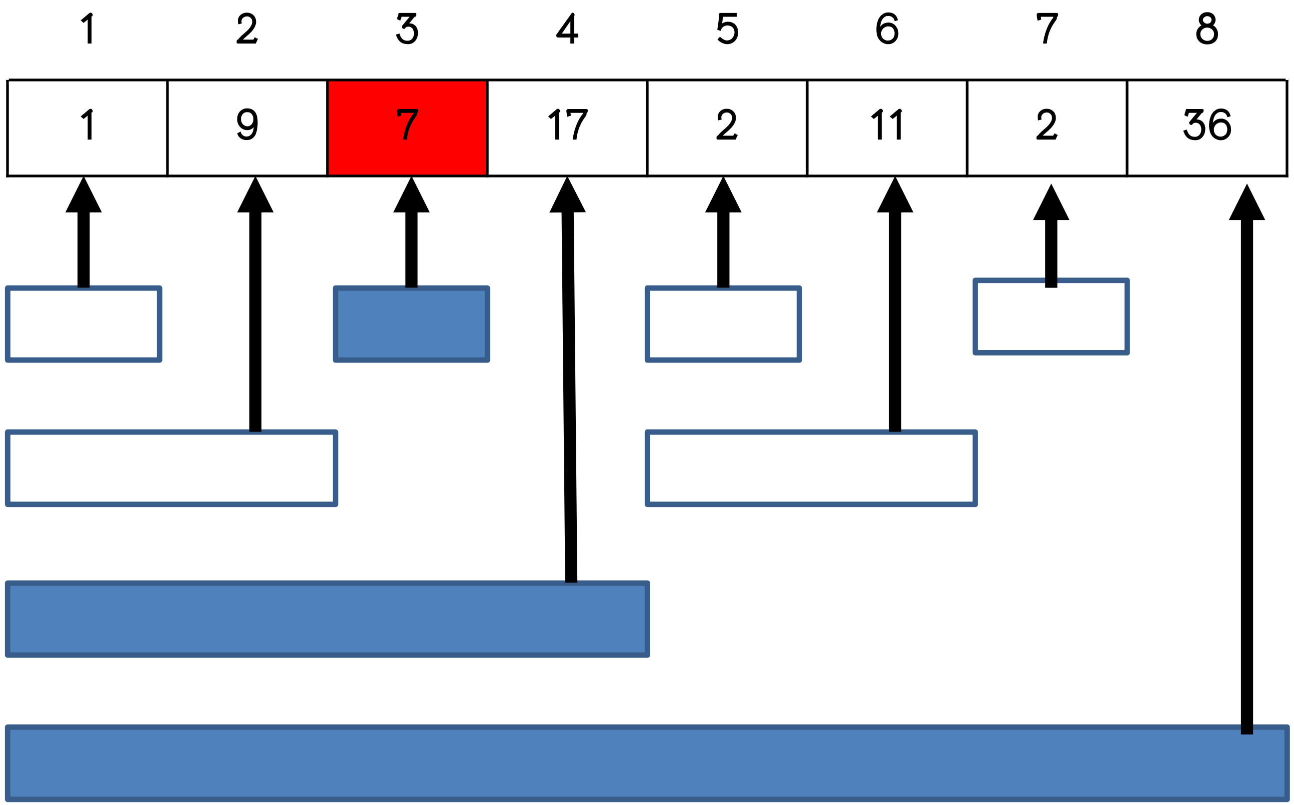
$$\mathbf{sum_q(1,7) = sum_q(1,4) + sum_q(5,6) + sum_q(7,7) = 17+11+2 = 30}$$

- ในการคำนวณค่า **$sum_q(a,b)$** เมื่อ a > 1 สามารถทำได้เช่นเดียวกับ prefix sum array

$$\mathbf{sum_q(a,b) = sum_q(1,b) - sum_q(1,a-1)}$$

- เนื่องจากเราสามารถคำนวณทั้ง **sum_q(1,b)** และ **sum_q(1,a−1)** ได้ในเวลา **O(logn)** ดังนั้นเวลารวมก็เป็น **O(logn)**

- หลังจากการ update ค่าใน array ต้นฉบับ พบว่าหลายๆ ค่าใน binary indexed tree ต้องการการ update ตัวอย่างเช่น ถ้าเราเปลี่ยนค่าในช่อง 3 ผลรวมของช่วงจะต้องเปลี่ยนหลายอัน



- เนื่องจาก array แต่ละค่านั้นจะอยู่ในช่วง **O(logn)** ช่วงใน binary indexed tree ดังนั้นเรา update **O(logn)** ช่วงนั้นของ tree

Implement

- การดำเนินการของ binary indexed tree สามารถ implement ให้มีประสิทธิภาพได้ โดยการใช้ bit operation

- key หลักที่ต้องการคือการคำนวณค่าของ **p(k)** โดยใช้สูตร

$$p(k) = k\&-k$$

Last set bit

การแยก bit แรกที่เป็น 1 ออกมา

ตัวอย่างเช่น x = 1110 (ในฐานสอง)

นี่เป็น last set bit

binary digit	1	1	1	0
Index	3	2	1	0

สมมติว่า x = a1b เป็นจำนวนที่ last set ที่เราต้องการแยก

โดย a เป็น binary sequence ที่ยาวเท่าไรก็ได้ที่มี 1 หรือ 0

b เป็น binary sequence ที่มีแต่ 0

101010110101	1	000000
a	1	b

ทั้งนี้ –x = 2's complement ของ x

–x = (a1b)'+1

= a'0b'+1 = a'0(0...0)'+1 =a'0(1..1)+1 = a'1(0..0) =a'1b

ดังนั้น

a1b

& a'1b

=(0..0)1(0..0)

ทำให้เราได้ last set bits

- ฟังก์ชันที่คำนวณ $sum_q(1,k)$

```
int sum(int k){
    int s = 0;
    while (k >= 1){
        s += tree[k];
        k -= k&-k;
    }
    return s;
}
```

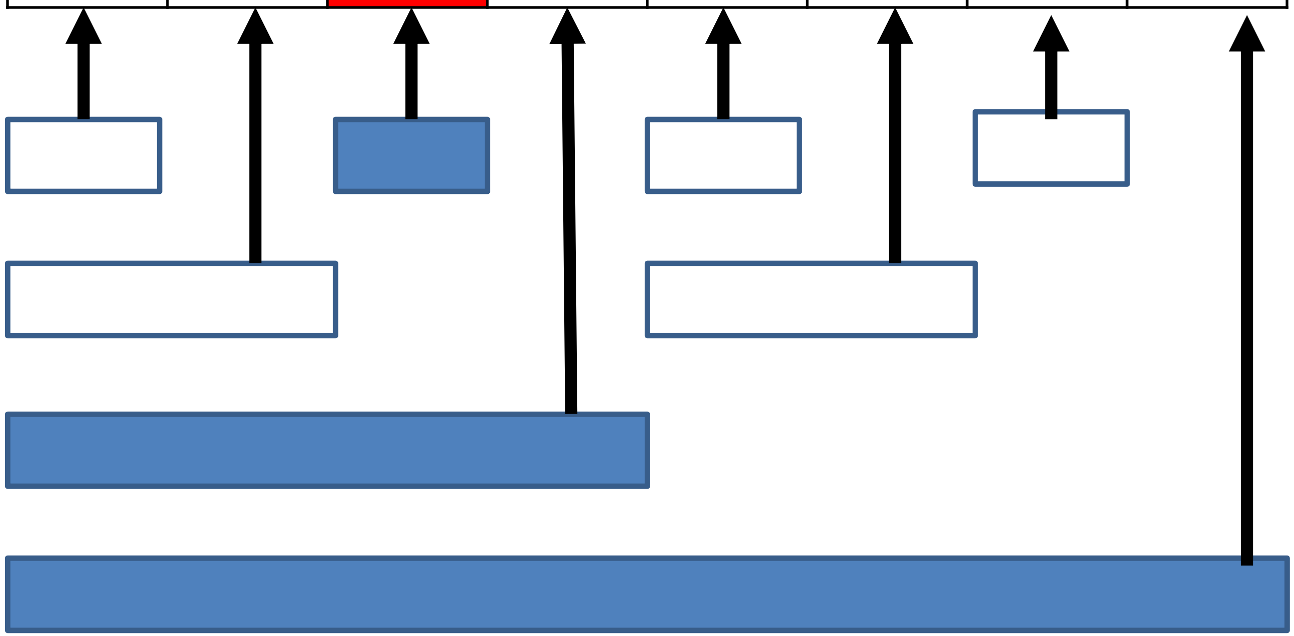
Sum 7
0111 tree[7]
0110 tree[6]
0100 tree[4]

Sum 8
1000 tree[8]

Sum 5
0101 tree[5]
0100 tree[4]

1	2	3	4	5	6	7	8
0001	0010	0011	0100	0101	0110	0111	1000

1	2	3	4	5	6	7	8
1	9	7	17	2	11	2	36



1	2	3	4	5	6	7	8
0001	0010	0011	0100	0101	0110	0111	1000

- ฟังก์ชันต่อไปเป็นการเพิ่มค่า array ช่องที่ k ด้วยค่า x

```
void add(int k, int x){
    while (k <= n){
        tree[k] += x;
        k+= k&-k;
    }
}
```

เวลาในการทำงานของทั้งสองฟังก์ชันคือ O(logn)

Segment tree

- Segment tree เป็นโครงสร้างข้อมูลที่รองรับการดำเนินการ 2 อย่างคือ การคำนวณการสอบถามเป็นช่วงและการ update ค่าใน array

- Segment tree สามารถรองรับ sum queries, minimum queries และ maximum queries และการสอบถามอื่นๆ ใช้เวลาในการทำงาน O(logn)

- เมื่อเทียบกับ Binary indexed tree ข้อได้เปรียบของ Segment tree คือ มันเป็นโครงสร้างข้อมูลที่ general กว่า ขณะที่ Binary indexed tree นั้น รองรับเพียง sum queries แต่ Segment tree รองรับการสอบถามอย่างอื่นด้วย แต่ต้องใช้หน่วยความจำมากกว่าและสร้างยากกว่า

โครงสร้าง

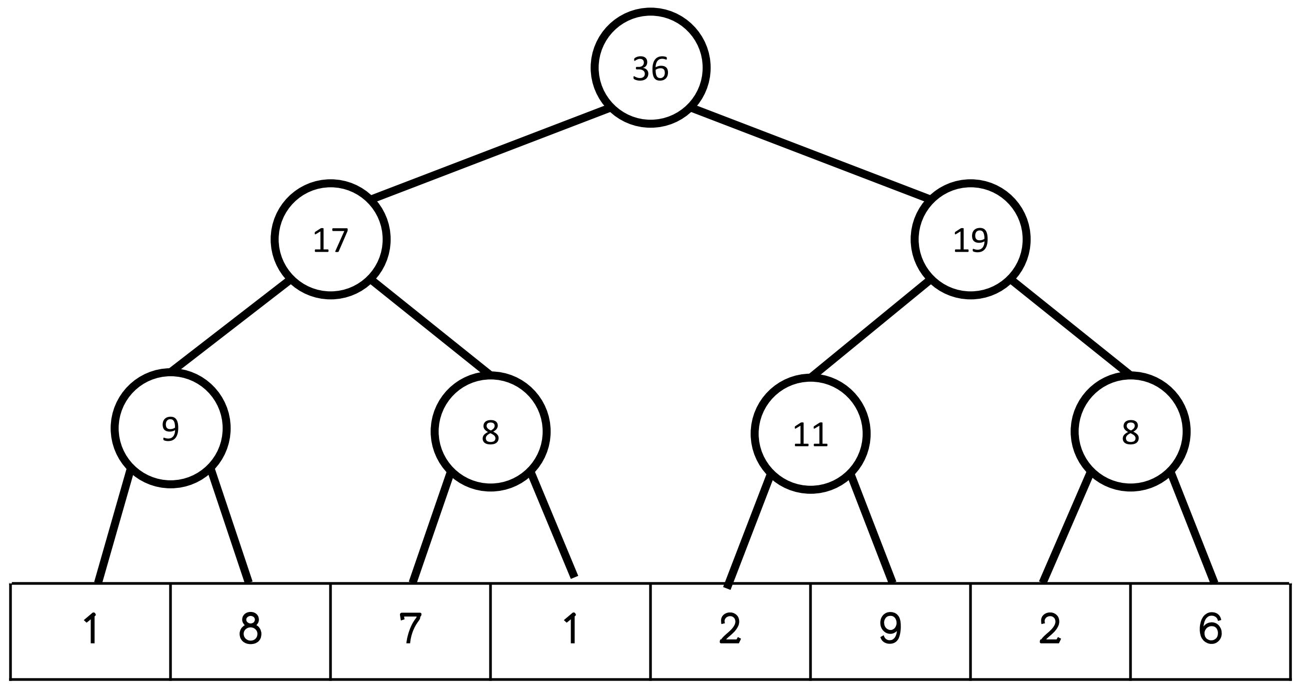
- Segment tree คือ binary tree ที่โหนดในชั้นล่างสุดของ tree เป็นสมาชิกใน array ส่วนโหนดในชั้นอื่นๆ จะเก็บข้อมูลที่จำเป็นสำหรับการประมวลผลแบบช่วง

- เราจะสมมติว่าขนาดของ array มีค่าเป็นยกกำลังของสองและเริ่มต้นใช้ช่องแรกที่ index 0 ถ้าหากขนาดของ array ไม่เท่ากับยกกำลังของสองก็เพียงขยายขนาด array

- เราจะเริ่มต้นด้วย Segment tree ที่รองรับ sum queries ให้พิจารณา array ต่อไปนี้

0	1	2	3	4	5	6	7
1	8	7	1	2	9	2	6

- จะได้ Segment tree ดังนี้



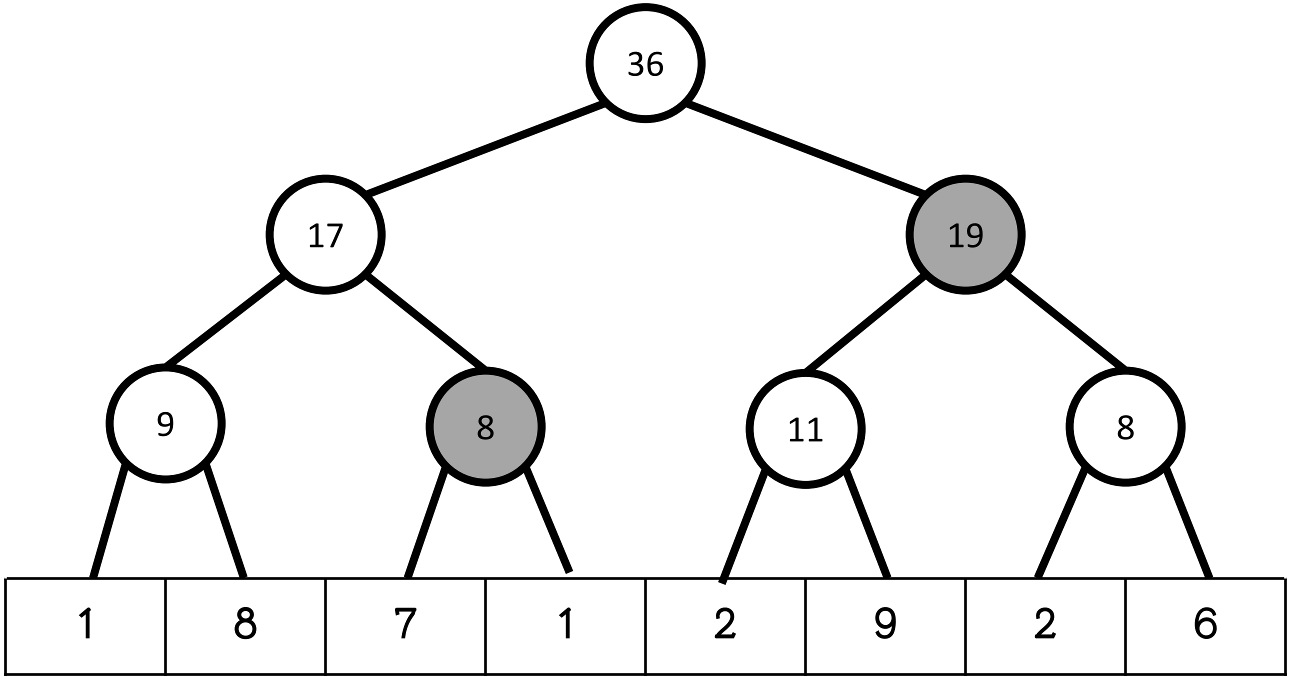
- แต่ละ internal node สอดคล้องกับช่วงของ array ซึ่งมีขนาดเป็นยกกำลังของสอง ในตัวอย่าง tree ค่าของ internal node คือผลรวมของ array ที่สอดคล้อง และมันถูกคำนวณได้จากผลรวมของค่าจากโหนดลูกทางซ้ายและโหนดลูกลูกทางขวา

- การทำเช่นนี้ทำให้ได้ว่า ช่วง [a,b] ใดๆ สามารถแบ่งได้เป็น O(logn) ช่วง ซึ่งค่าถูกเก็บในโหนดใน tree

- พิจารณช่วง [2,7] จะได้ว่า $\mathbf{sum_q(2,7)}$ =7+1+2+9+2+6=27

0	1	2	3	4	5	6	7
1	8	7	1	2	9	2	6

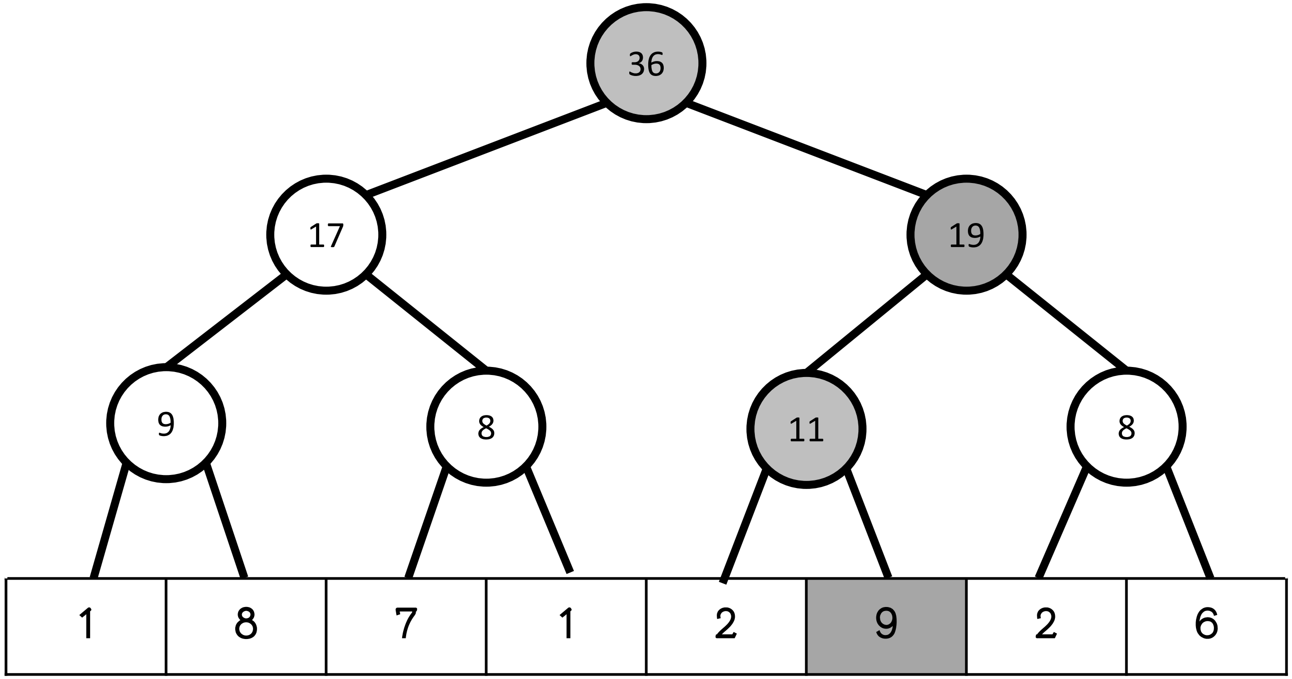
- โหนดใน tree สองโหนดสอดคล้องกับช่วงดังกล่าวซึ่งได้ค่า 8+19=27



- เมื่อผลรวมถูกคำนวณโดยใช้โหนดที่อยู่ในชั้นสูงที่สุดเท่าที่จะเป็นไปได้ใน tree พบว่าไม่เกินสองโหนดในแต่ละชั้นของ tree ที่จะถูกใช้ ดังนั้นจำนวนของโหนดทั้งหมดคือ O(logn)

- หลังจาก array ถูก update เราจะต้อง update ทุกโหนดที่ค่าของมันขึ้นกับค่าที่ update ด้วย ซึ่งทำได้โดย traverse ไปตาม path จากค่าที่ update ใน array ไปยังโหนดบนสุดและ update โหนดตาม path นี้

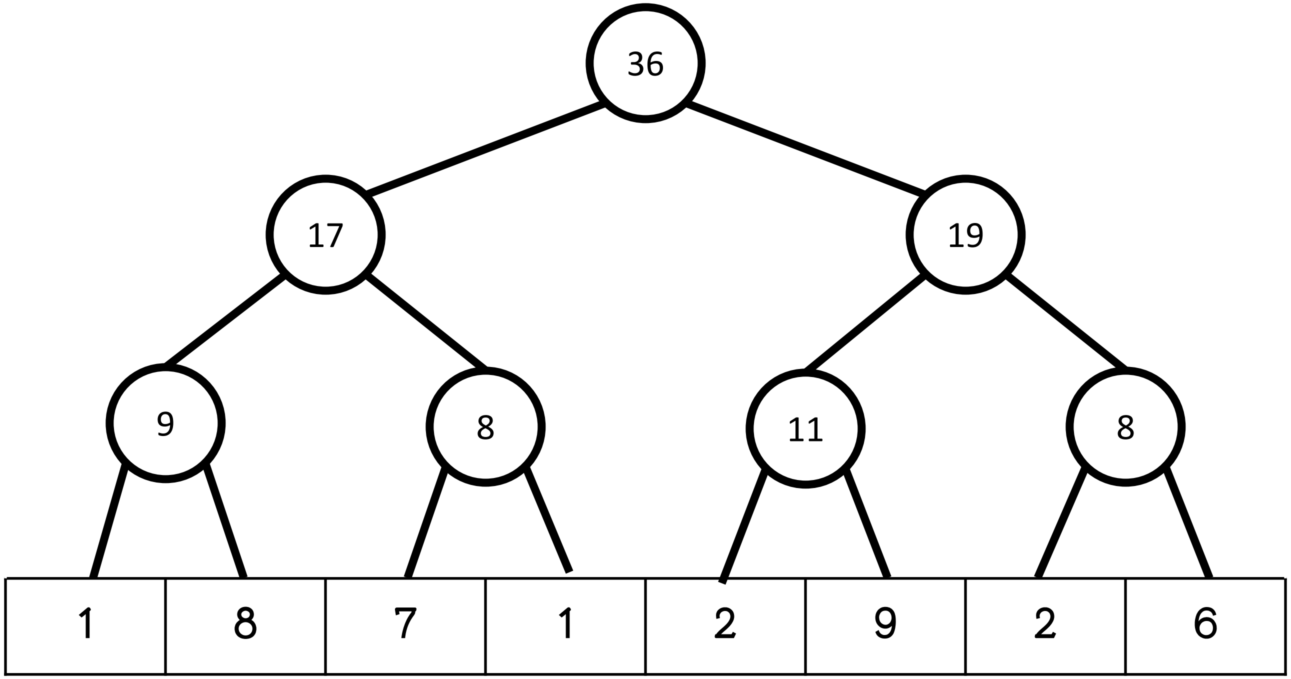
- รูปต่อไปแสดง tree หากมีการ update ค่า 9



- path จากโหนดล่างสุดไปโหนดบนสุดจะประกอบด้วย O(logn) โหนด ดังนั้นการ update แต่ละครั้งเปลี่ยน O(logn) โหนดใน tree

Implementation

- เราจะเก็บ Segment tree เป็น array ขนาด 2n เมื่อ n เป็นขนาดของ array ต้นฉบับและมีค่าเป็นยกกำลังของสอง
- โหนดใน tree จะถูกเก็บจากบนลงล่าง นั่นคือ tree[1] เป็นโหนดบนสุด tree[2] และ tree[3] เป็นลูกของมัน ไปเรื่อยๆ
- สุดท้ายค่าจาก tree[n] ถึง tree[2n−1] จะเป็นค่าของ array ต้นฉบับซึ่งเป็นชั้นล่างสุดของ tree



- Segment tree จะถูกเก็บดังนี้

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
36	17	19	9	8	11	8	1	8	7	1	2	9	2	6

- จากการเก็บแบบนี้เราพบว่า parent ของ tree[k] คือ tree[$\lfloor k/2 \rfloor$] และลูกของมันคือ tree[2k] และ tree[2k+1] สังเกตว่าเลขคู่เป็นลูกทางซ้าย
- ต่อไปเป็น $sum_q(a,b)$

```
int sum(int a, int b){
        a = a+n;
        b = b+n;
        int s=0;
        while(a<=b){
                if(a%2==1)
                        s=s+tree[a++];
                if(b%2==0)
                        s=s+tree[b--];
                a=a/2;
                b=b/2;
        }
        return s;
}
```

- ฟังก์ชันเก็บช่วงที่เริ่มต้น [a+n, b+n] หลังจากนั้นแต่ละ step ช่วงจะถูกย้ายขึ้นไป 1 level ใน tree และก่อนนั้น ค่าของโหนดที่ไม่ได้เป็นของช่วงที่สูงขึ้นจะถูกรวมเข้ากับ sum

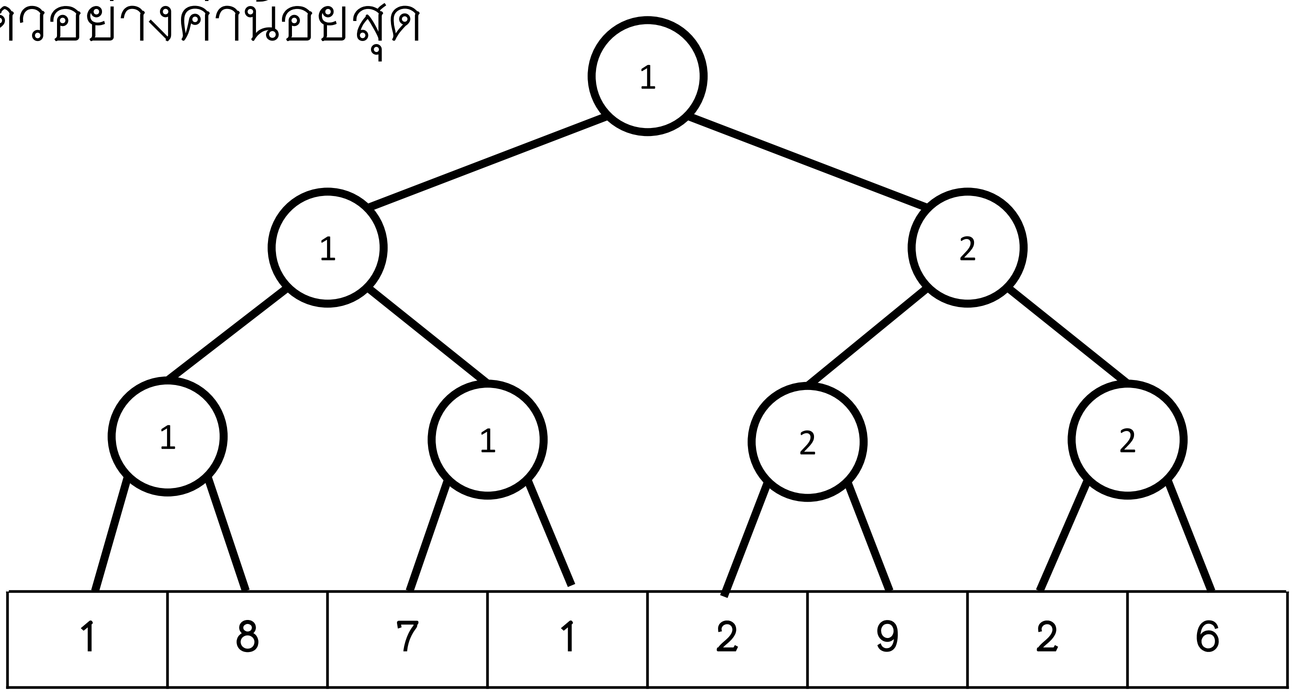
```
void add(int k, int x){
      k=k+n;
      tree[k] = tree[k]+x;
      for(k=k/2;k>=1;k=k/2){
            tree[k] = tree[2*k]+tree[2*k+1];
      }
}
```

- เริ่มต้นฟังก์ชันจะ update ค่าที่ level ล่างสุดของ tree หลังจากนั้นฟังก์ชันจะ update ค่าของทุกๆ internal nodes จนกระทั่งถึงโหนดบนสุด

- ทั้งสองฟังก์ชันทำงานใน O(logn) เพราะว่า Segment tree ของ n ตัวประกอบด้วย O(logn) ชั้น และฟังก์ชันในแต่ละรอบจะย้ายไปทำงานยัง level ที่สูงขึ้น

Other queries

- Segment tree สามารถรองกับการสอบถามแบบช่วงทุกรูปแบบที่มันสามารถแบ่งช่วงออกเป็น**สอง**ส่วน คำนวณคำตอบแยกกันจากนั้นค่อยนำมารวมกัน
- ตัวอย่างเช่นค่าน้อยสุด ค่ามากสุด หรม และ bit operations ต่างๆ ต่อเป็นเป็นตัวอย่างค่าน้อยสุด



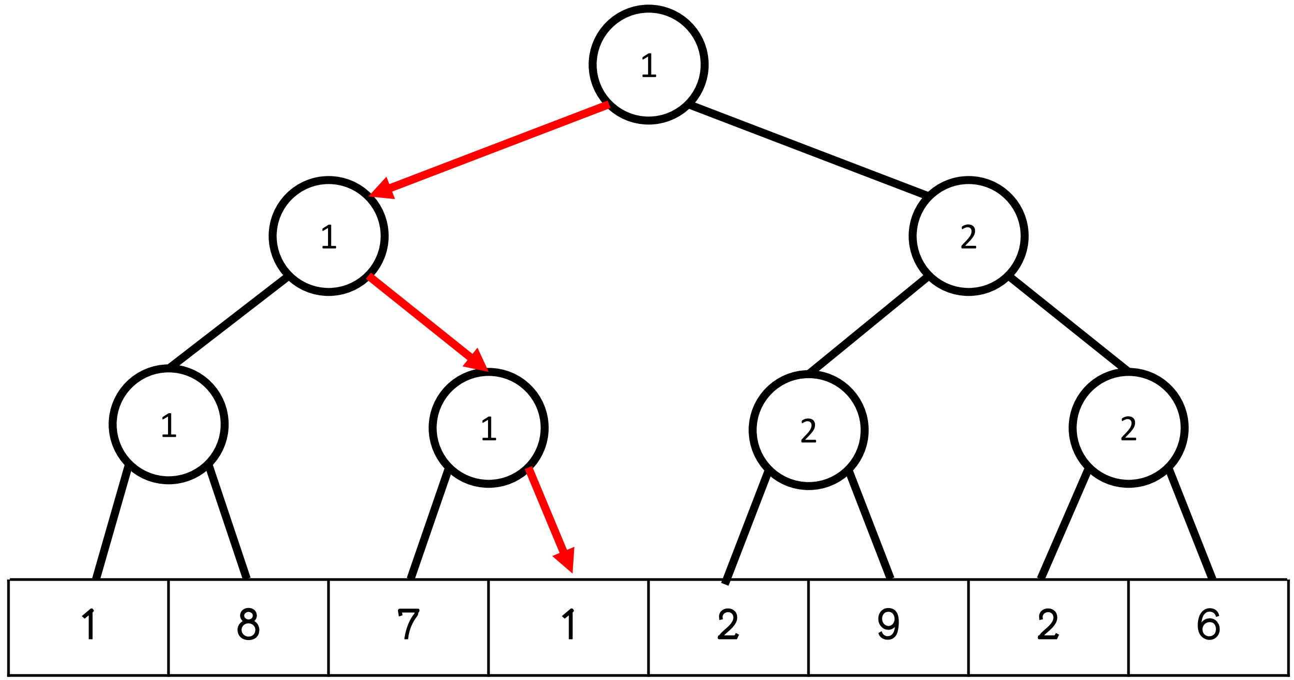
- ในรูปก่อนหน้า ทุกโหนดใน tree จะเก็บค่าที่น้อยที่สุดที่สอดคล้องกับช่วง

- โหนดบนสุดของ tree จะเก็บค่าน้อยที่สุดของทั้ง array

- การดำเนินการสามารถทำได้คล้ายกับตัวอย่าง sum ก่อนหน้า เพียงแค่คำนวณค่าน้อยกว่าแทน

- โครงสร้างของ segment tree อนุญาตให้เราใช้ binary search สำหรับค้นหาตำแหน่งของโหนดใน array ตัวอย่าง ถ้า tree รองรับ minimum queries เราสามารถหาตำแหน่งของโหนดที่มีค่าน้อยสุดได้ใน O(logn)

- ตัวอย่าง ค่า 1 เป็นค่าที่น้อยที่สุดเราก็ท่องไปใน tree ลงไปจากโหนดบนสุด



Additional Technique

- Range updates
- ก่อนหน้านี้เราได้ลองสร้าง data structure ที่รองรับการสอบถามแบบช่วงและ update ค่าได้ทีละ 1 ค่า ต่อไปเราจะมาพิจารณากรณีที่ต่างออกไปคือ เมื่อเราต้องการ update เป็นช่วงและสืบค้นข้อมูลค่าเดียว
- สมมติว่าเราพิจารณาการเพิ่มทุกค่าในช่วง [a,b] ด้วยค่า x
- เอทำอย่างไรดี

- จริงๆ แล้วเราสามารถใช้โครงสร้างข้อมูลที่ทำไปก่อนหน้านั้นได้ ในกรณีนี้
- เราจะสร้าง difference array ซึ่งเก็บค่าผลต่างระหว่างตัวที่ติดกันใน array ต้นฉบับ (ตัวเราลบตัวก่อนหน้า)
- ดังนั้น array ต้นฉบับ ก็คือ prefix sum array ของ difference array นั่นเอง !!!
- ตัวอย่างเช่น

1	2	3	4	5	6	7	8
1	8	7	1	2	9	2	6

- difference array คือ

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

- จากตัวอย่าง ค่า 9 ในตำแหน่งที่ 6 ใน array ต้นฉบับสอดคล้องกับผลรวมของ **1+7−1−6+1+7 =9** ใน difference array

- ข้อดีของ difference array คือเราสามารถ update ช่วงใน array ต้นฉบับได้ โดยการเปลี่ยนเพียงแค่ 2 ค่าใน difference array
- ตัวอย่างเช่น ถ้าเราต้องการเพิ่มค่าใน array ต้นฉบับในตำแหน่ง 1 ถึง 4 ด้วยค่า 5 เราก็เพียงเพิ่ม ใน difference array ในช่อง 1 ด้วยค่า 5 และลดลงที่ช่อง 5 ด้วยค่า 5 จะได้

1	2	3	4	5	6	7	8
6	7	−1	−6	−4	7	−7	4

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

- ในการเพิ่มค่าในช่วง [a,b] ด้วย x เราเพิ่มค่าที่ตำแหน่ง a ด้วยค่า x และลดลงที่ตำแหน่ง b+1 ด้วยค่า x
- ดังนั้นเราเพียงต้องการ update ค่า และประมวลผล sum queries เราสามารถใช้ binary indexed tree หรือ segment tree ได้

- แบบฝึกหัด [LightOJ – 1082]