Day 24 : Segment tree & Binary Index Tree

Q. Given an array perform two operations on it such that (no duplicates in the array)

1) A[i] = x
2) Tell the ~~minimum~~ max number Y, Y ≤ X   ST

1 5 8 13 4 6 9 10

BST → inorder → sorted

(i) O(1), O(n) → normal processing on the array.

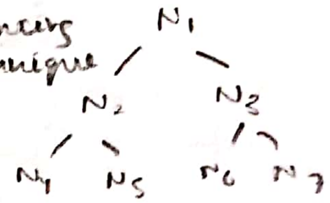
(ii) balanced BST → a) insertion → depends on implementation of balancing technique

b) deletion → O(log n)

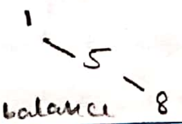
c) access or the query → O(log n)

otherwise the tree can be skew, O(n)

(red black tree)



(i) for each node during insertion, check balanced, and accordingly balance



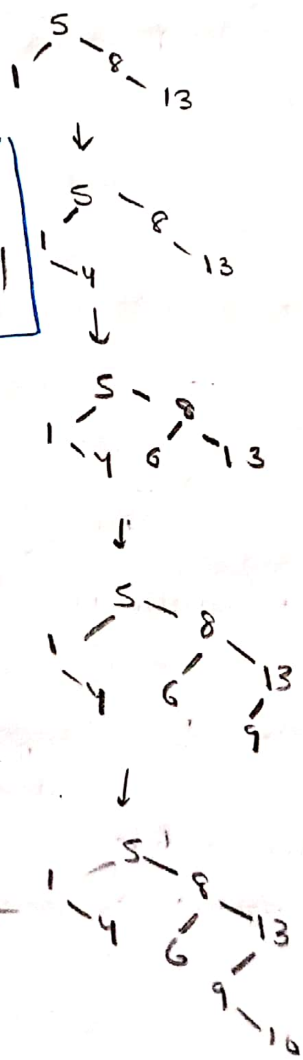
inorder

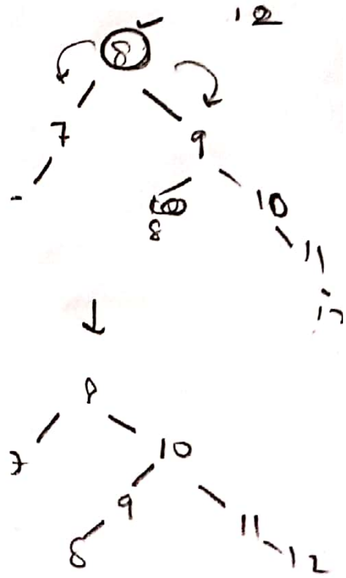
LNR

A[i] = x : (i) deletion of A[i]
(ii) and then insertion of x

greatest number less than x

9



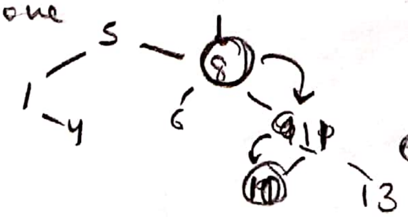


1 4 5 6 8 9 10 13

9

* index would be stored at each node, to know which one to delete

as T



we can traverse the tree in the most logical order, and keep maintaining a value, and then as we reach the end

STL

both have unordered counterparts { ↳ set &
↳ map.
(ordered)

vs AVL tree
↑
internally use red black tree
↓
self balancing BST
↓(operation 1)

for insertion, we first delete the element, and then insert the new element. A[i] x

s.erase(s.find(A[i])); ← using set
s.insert(x);

operation 2: max number smaller than or equal to x

↓
using lowerbound or upper_bound

returns an iterator
↓
so find a valid iterator using lower bound or upper bound and subtract 1 from it. to get the max element less than x.

lowerbound: ↓ ~~largest x or number number smaller than x~~
↓
x or number just greater than x

upper_bound: ↓ strictly greater than x
↓
if there is no such element, then iterator to last element is returned.

both included?
↑
operation 3: given(x, y), find number of elements in [x, y]

i) sort, O(nlogn) → O(n)
ii) upperbound() - lowerbound()
↓
difference of 2 iterators O(n)
← { ↓
distance → O(n)

not always defined

iterator : LegacyRandomAccessIterator
↓ O(1).

1 2 3 4 5 6
[3, 5] → 3

6 1 8 3 2 4 5
↓
1 2 3 4 5 6 8
↑ ↑

(RBT doesn't support -)

~~seg~~ 2 possible solutions : i) segment tree

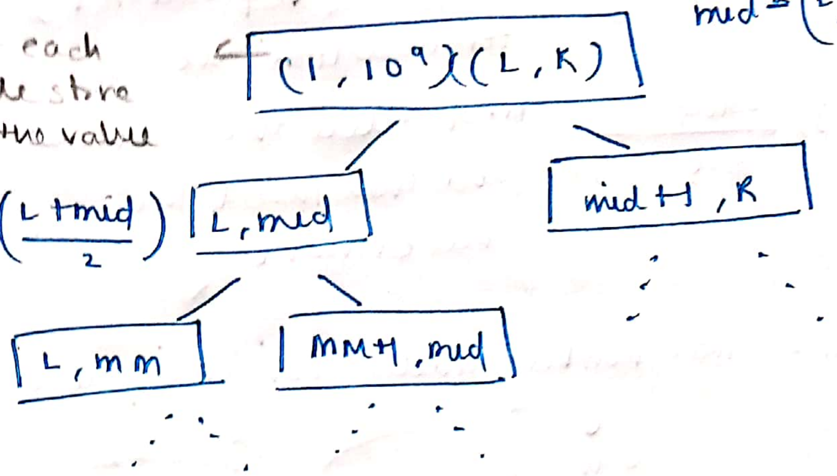
ii) binary index tree (fenwick tree)

i) segment tree → (1, 10^9) range

at each node store the value

mid = $\left(\frac{L+R}{2}\right)$

mm = $\left(\frac{L+mid}{2}\right)$



total levels = $\log_2 N$.

↓

1 5 8 3 2 15 23 35 40

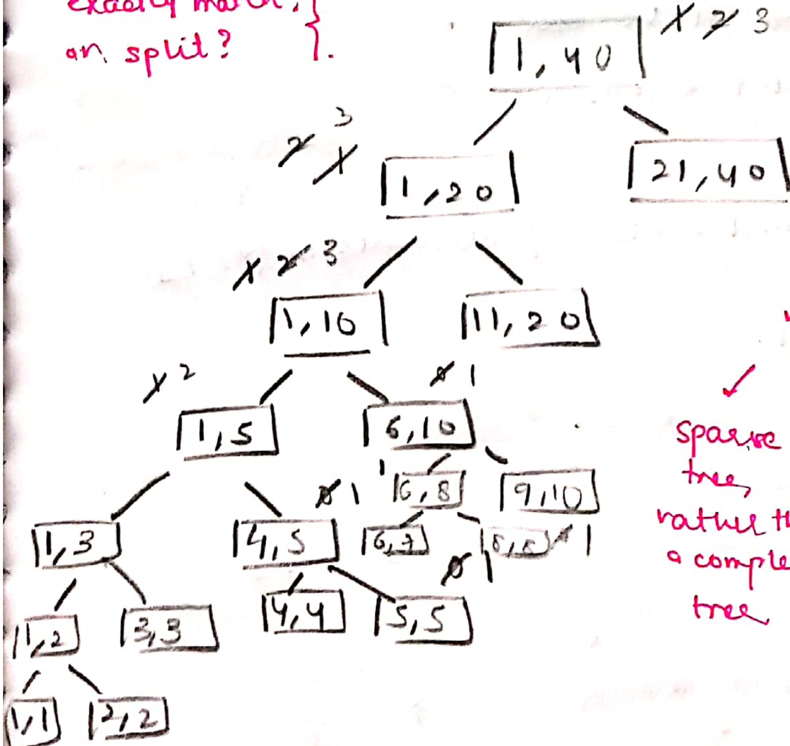
i) updates X to Y

ii) No of elements in [X, Y]

Sorted array ?
↓
O(n).

unsorted array ?
↓
set → O(n)
↓
(log n + rotation)
↓
O(n(log n + rotation))
↓
set to segment tree, for value based range ←
↓
access or query answering
↓
O(log n)

exactly match?
or split?



* do we construct the complete tree at the beginning itself ?

no, ← create new nodes as per requirement.

sparse tree, rather than a complete tree

given a query, we check if it lies in left side or right side, and accordingly move till the ranges match
↓
if the first and last lie on opp side, then accordingly

← LHS of left side segment matches.

(i) update X to Y : a) deletion on(X, X)
b) increment on(Y, Y)

↓

(i) deletion of node in tree

(ii) insertion of node in tree

on deletion, check for presence of key.

↓

since deletion is not an operation specified, so no need.

insertion : $O(\log n)$

deletion : $O(\log n)$ → as soon as we get 0 at some node, we can stop moving ahead.

query : $O(\log n)$ → we keep skipping one subtree at each point, so we skip one range at each point.

↓

($2 \log n$) when we move on both sides

↓

$O(\log n)$

implementation →

* in interview : ST implementation is not asked generally.

* initial range should be such that, it keeps into account all future values to be implemented.

↓

(min, max query not supported in BIT)

↓

(use depending on requirement)

↓

* BIT won't work always, but segment tree generally works.

* segment tree to store min element in the given range (building the tree bottom to up)

↓

recursively, work on child first, till leaf node is not reached, and then move on.

↓

can be done iteratively, if we keep track of parent.

↓

through array or node structure can be used to keep parent pointer.

[x, y] → count of elements

↳ max element in the range.

* original question, y ≤ x → max in range [min, x]

↓

maintaining multiple entries to answer different types of queries. ← using max segment tree

* tradeoff between tree construction and time complexity, efficiency.

↓

i) 2 different arrays

ii) element in the array can be a pair, or struct or node element itself.

↓

(the limits of complete tree construction using array) [A]

↓

hence node based implementation seems better

↓

(too many elements in case of big range, in case of complete tree)

* implementation based ~~[illegible]~~ decisions

ii) Binary index tree / fenwick tree

↓

operations : a) update

b) findPrefixSum (0, i)

1 1 0

⇓

1

1	2	3	4	5	6
1	3	5	8	10	15

⇓

BIT

6	25
5	10
4	17
3	5
2	4
1	1
0	0

BIT[i] ⟶ i is odd ⟶ A[i]

↳ i is power of 2 ⟶

sum [A[1] . . . . A[i]]

⟶ J = i after removing the last set bit, then

BIT [i] = Sum[A[i], A[i−1]. . . . A[J-1])

↓

(J not inclusive)

application : prefix anything (sum, max, min) .

↓ bit unsetting → gives the last set bit

(gives last set bit) ← J = i & (−i)

↓

2's complement

(x − = J) or x −= (x & (−x))

10100

01011

1

01100

00100

1 : 35 : 2 c

* update : 0 1 2 3 4 5 6
1 2 5 8 15 20 29

if we change 9 to 3, then ~~all the~~ the sums that involve this element will also have to be changed by Δ

↓

The main question here stands, that which are these elements that need to be updated, & which ones can be left as it is.

binary representation of the index to be ~~represented~~ updated.

↓

10(1)0 → 10

$x - (x \& (-x))$

= 1010

1100 → 1000 → 10000 → keep updating

J → from i to j where j is the no. formed by removing the last set bit

64
32
16 → A[1] . A[16]
15 → A[15]
14 → A[14] + A[13]
13 → A[13]

1110 → 1100

→ 12 → A[12] + A[11] + A[10] + A[9]

1100 11 → A[11]

↓ 1000

10 → A[10] + A[9]
9 → A[9]
8 → A[1] + ... A[8]

1010 → $\boxed{1000}$

16 → 10000
18 → 10010
20 → 10100
22 → 10110
24 → 11000

16 → 24

32 → 100000
34 → 100010
36 → 100100
38 → 100110
40 → 101000
42 → 101010
44 → 101100
46 → 101110
48 → 110000

18 → 10010 $\xrightarrow{-RSB}$ 10000 → 16

Bit[18] = A[18] + A[17]

24 → 11000 $\xrightarrow{-RSB}$ 10000 → 16

26 → 110(1)0 $\xrightarrow{-RSB}$ 11000 → 24

30 → 11110 $\xrightarrow{-RSB}$ 11100 → 28

* ~~Amin~~ min to min, they will map to their ~~highest~~ highest bit msB.

Scanned by CamScanner

12 → 11000

Bit[12], Bit[16], Bit[32]

18 → 10010

Bit[18]

Bit[18] = A[18] + A[17]

22 ← 10100

Bit[22] = A[22] + A[21] + A[20] + A[19] + A[18] + A[17]

24 ← 11000

Bit[24] = A[24] + ... + A[17]

22 → 10100

Bit[22], Bit[24], Bit[32]

26 → 11010 → A[26] + A[25]

22 → 10100

```
void update(int x, int delta){
    while (x<=n){            range approaching
        BIT[x] += delta;
        x = x + (x & (-x));
    }
}
```

x 10100 (22)

01011
\+ 1
-x 01100

10100
01100
(x & -x) 00100

x 10100
+(x & -x) 00100
x' = 11000 (24)

x' → 11000
-x' → 00111
1
01000

x' & -x' → 01000

x' 11000
+(x' & -x') 01000
100000 (32)

x : 18 → 10010

x' : 01101

$$\begin{array}{r} 01101 \\ +\quad 1 \\ \hline 01110 \end{array}$$

x & x' : 00010

x : 10010
+ x & x' : 00010

10100 → 20

x : 10100

x' : 01011

$$\begin{array}{r} 01011 \\ +\quad 1 \\ \hline 01100 \end{array}$$

x & x' : 00100

x : 10100
x & x' : 00100

11000 → 24
↓
32
↓
64

---

1 0 1 1 0 0 1 1 0
↓
1 0 1 1 0 1 0 0 0 > i
↓
1 0 1 1 0 0 0 0 0 < i


```
int query ( int x ) {
    int sum = 0;
    while (x > 0) {
        sum += BIT[x];
        x = x - (x & -x)
    }
    return sum;
}
```

→ as per the required function
↓
here the prefix sum was required

Application of BIT : i) given an array which keeps getting updated, we can find sum in ranges using

prefix sum (0 , i) - prefix sum (0, j-1)

~~ii) max (0 , i) → max in the specified range~~