(11) APPLICATIONS OF BBST

We can perform all the operations of ~~same~~ segment trees using AVL. Vice versa is NOT true.

—x—

(*) Range Query

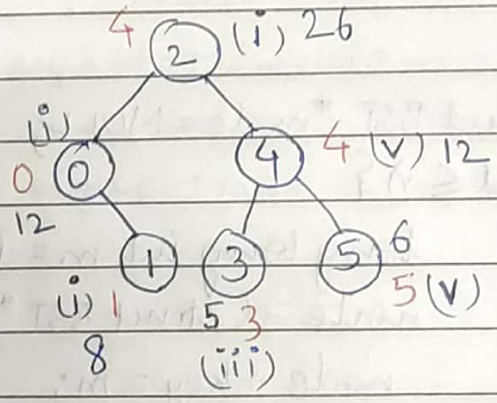
0	1	2	3	4	5
4	8	2	5	1	6

RangeMaxQuery(A, 0, 4) = 1;
RangeMinQuery(A, 0, 3) = 2;

Update(A, i, X) = set A[i] = X;

The tree will be formed on the basis of the index

For above ex the tree will be



The 3 parameters in the structure of the node:-

min
↓
minimum of the subtree rooted at the node

max(roman)
↓
maximum of the subtree rooted at the node

sum
↓
Sum of all the no. rooted at the node.

```
struct BST{
    long long int key, value, sum, max, min;
    struct BST *left, *right;
};
```

—x—

We've already seen (in BST) that given a sorted sequence we can create a BBST in linear time.

Here we're making the tree with indices as the keys. Since the indices are already sorted ⇒ we can create a BBST in $\Theta(n)$.

—X—

→ struct BST *CreateBBST(long long int A[], long long int l, long long int r)

(height of the tree = $\Theta(\log n)$)

```
{
    struct BST *node = NULL;
    if (l <= r){
        long long int m = (l+r)/2;
        node = (struct BST *)malloc(sizeof(struct BST));
        node->key = m;
        node->value = A[m];
        node->left = CreateBBST(A, l, m-1);
        node->right = CreateBBST(A, m+1, r);
        NodeUpdate(A, node);
    }
    return node;
}
```

$\Theta(n)$

we do this in the bottom-up fashion

→ constant time operation
$\Theta(1)$

```
→ void NodeUpdate (long long int A[],
                   struct BST *node)
{
   if (node){
      node → max = node → min = node → key;
      node → sum = node → value;
      if (node → left) {
         if (A[node → left → min] < A[node → min])
            node → min = node → left → min;
         if (A[node → left → max] > A[node → max])
            node → max = node → left → max;
         node → sum += node → left → sum;
                  }

      if (node → right) {
         if (A[node → right → min] < A[node → min])
            node → min = node → right → min;
         if (A[node → right → max] > A[node → max])
            node → max = node → right → max;
         node → sum += node → right → sum;
                  }
      }
}
```

$O(\log n)$

→ void Update(long long int A[], struct BST *node,
long long int *i, long long int X)

```
{
  if(node){
    if(node->key > i) Update(A, node->left, i, X);
    else if(node->key < i)
      Update(A, node->right, i, X);
    else{
      A[i] = X;
      node->value = X;
    }
    NodeUpdate(A, node);
  }
}
```

returns the sum of no. present at indices $\leq i$

→ long long int PreSum(struct BST *node, long long int i)

```
{
  long long int sum = 0;
  while(node){
    if(node->key > i) node = node->left;
    else if(node->key < i){
      if(node->left) sum += node->left->sum;
      sum += node->value; node = node->right;}
    else{
      if(node->left) sum += node->left->sum;
      sum += node->value;
      return sum;}}
  return 0;}
```

$O(\log n)$

returns x such that ~~min value of~~ A[x] is minimum for $l \le x \le r$

```
long long int RangeMin(long long int A[],
                       struct BST *node,
                       long long int l,
                       long long int r,
                       long long int s,
                       long long int e)
{
  if(node){
    if(s>=l && e<=r) return node->min;
    else if(e<l || r<s) return -1;
    else {
      long long int min, ml, m=(s+e)/2;
      if(node->key>=l && node->key<=r)
        min=node->key;
      else
        min=-1;
      ml=RangeMin(A,node->right,l,r,m+1,e);
      if(min<0) min=ml;
      if(min>-1 && ml>-1 && A[ml]<A[min])
        min=ml;
      ml=RangeMin(A,node->left,l,r,s,m-1);
      if(min<0) min=ml;
      if(min>-1 && ml>-1 && A[ml]<A[min])
        min=ml;
      return min;
    }}
  return -1;
}
```

O(logn)

It can done in O(1) if we use the parameter max & min of the structure.

→ O(logn)

(*) RangeMaxGap → max. difference among all the numbers in the given range.

→
```
long long int MaxGap(long long int A[],
                     struct BST *node,
                     long long int l,
                     long long int r,
                     long long int n)
{
    return A[RangeMax(A, node, l, r, 0, n-1)]
           - A[RangeMin(A, node, l, r, 0, n-1)];
}
```

—×—

→ O(nlogn)

(*) RangeMinGap(i, j) → find the minimum difference among all the no. b/w A[i] & A[j].

→ Sort the no. b/w A[i] & A[j] & compare the consecutive no.

This is the best we can do with the data structure that we already have.

—×—

However, to ans the RangeMinGap Query also in O(1) we can design another data structure with some more parameters.

Here node → key = value itself (instead of the index)

```
struct BST
{
    long long int key;
    int height;
    long long int count, c, min, max, mingap;
    struct BST *left, *right;
};
```

→ to handle duplicacy.

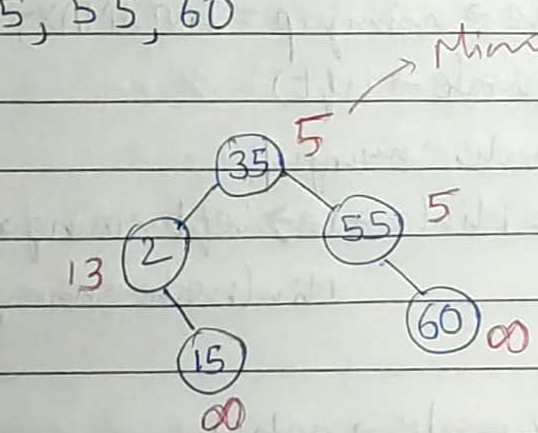
Def^n

→ Node → max = Max of { Node → value, Node → left → max, Node → right → max }

→ Node → min = Min of { Node → value, Node → left → min, Node → right → min }

→ node → mingap = Min of { (node → value) − (node → left → max), (node → right → min) − (node → value), node → left → mingap, node → right → mingap }

Ex → 2, 15, 35, 55, 60



⇒ The following update fun" is called after ~~you~~ you insert or delete.

- Each update is constant time operation. After insert and delete you may have to ~~do~~ call this fun" atmost logn time. So, this won't add any additional overhead to insert, delete.

→ O(1)

```
void Update(struct BST *node){
    node->max = node->min = node->key;
    if(node->left && node->left->max > node->max)
        node->max = node->left->max;
    if(node->right && node->right->max > node->max)
        node->max = node->right->max;

    if(node->left && node->left->min < node->min)
        node->min = node->left->min;
    if(node->right && node->right->min < node->min)
        node->min = node->right->min;

    node->mingap = INT_MAX;
    if(node->left)
    {node->mingap
      = Min(node->left->mingap,
            Min(node->mingap, node->key - node->left
                ->max));
    }
    if(node->right)
    { node->mingap
      = Min(node->right->mingap,
            Min(node->mingap, node->right->min - node
                ->key));
    }
```

```
    node->height = Height(node);
    node->count = Count(node);
}
```

—x—

(*) In this data structure, we can find the mingap & mangap of whole array in $\Theta(1)$.

—x—

(*) RangeMaxGap ⇒ (l, r) → gives the mangap for no. lying in [l, r]

$\Theta(\log n)$

if the given l is not there in the tree then find its successor.
" " " r " " " " " " predecessor.
& return (r-l).

→

```
long long int MaxGap(struct BST *node,
                     long long int l,
                     long long int r)
{
  if (l <= r)
  {
    if (!Search(node, l))
      l = Successor(node, l)->key;
    if (!Search(node, r))
      r = Predecessor(node, r)->key;

    return (r-l);
  }
  return -1;
}
```

→ $\theta(\log n)$

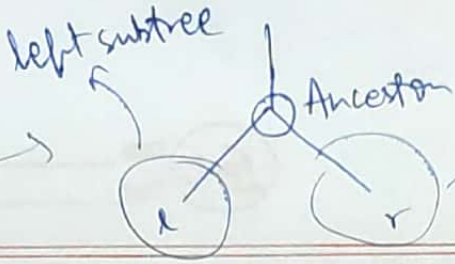
(*) Range Min Gap

→
```
long long int (struct BST *node, long long int l,
                long long int r)
{
    if (l <= r)
    {
        if (!Search(node, l))
            l = Successor(node, l) -> key;
        if (!Search(node, r))
            r = Predecessor(node, r) -> key;

        return MinGap1(node, l, r);
    }

    return INT_MAX;
}
```

— × —

In the following MinGap1 fun^{n} :—
we travel to the right
if l >= node -> key
b/c given the above condition true
we know that both l & r lie
in the right subtree

similarly if r <= node -> key
we move to the left.



II If neither of the above conditions are true then l will be on the left hand side while r " " " " right " "

III MinGap2 fun finds the mingap among the no. which are greater than l, on the left hand side.

Similarly MinGap3 fun finds the mingap among the no. which are smaller than r, on the right hand side.

IV After this we'll see that the ancestor node can make the mingap either with left hand side or right hand side

CODE FOR FUN WRITTEN IN NEXT PAGE.

→ long long int MinGap1 (struct BST *node,
long long int l,
long long int r)

{
if (l ≤ r && node)
{
I | if(node→key ≤ l)
return MinGap1(node→right, l, r);

else if (node→key ≥ r)
return MinGap1(node→left, l, r);

II | else
{
III | long long int min = INT_MAX;
min = Min(MinGap2(node→left), l),
MinGap3(node→right, r));

if(node→left && node→left→key ≥ l)
min = Min(min,
node→key - node→left
→max);

IV

if(node→right && node→right→key ≤ r)
min = Min(min,
node→right→min
- node→key);

return min;
}}
return INT_MAX;
}