Augmented Binary Search Trees

Height balanced BSTs (RBTs)

i) Search (T, x) ii) Find-Rank (T, x)
iii) Delete (T, x) iv) Insert (T, x)
v) Max (T) vi) Min (T)
vii) Predecessor (T, x) viii) Successor (T, x)
ix) Special Union (T, T') x) Split (T, x)

$T < T'$
$T^* = T \cup T'$

$T < \begin{matrix} T' \\ T'' \end{matrix}$ $T' < x < T''$

All operations in $O(\log n)$ time.

Predecessor

```
if (u->val < x) { lm = u;
                  u = u -> right; }
else              u = u -> left;
```

Pseudo Code for Special Union (T, T')

i) Let x be the node storing smallest element of T'.
ii) delete node x from T'.

Let black height of $T \leq$ black height of T'
i) Keep moving left in T' till we reach a node v s.t
   a) left (v) is black (both roots black)
   b) subtree T'' rooted at left (v) has equal black height as T.
ii) left $(x) \leftarrow T$ right $(x) \leftarrow T''$
   par (T), par $(T'') \leftarrow x$.
iii) par $(x) \leftarrow v$. left $(v) \leftarrow x$.

iv) color(x) ← red.
If v also red, remove color imbalance.

Find-Rank

- $O(k \log n)$: repeatedly find all predecessors of x.
- all predecessors can be computed in $O(k + \log n)$

we can do a split in the RBT, by x, and then traverse the left tree to list all predecessors ⇒ $O(k + \log n)$ time

Augumenting with size :

i) Elements present in the left subtrees of the path
ii) Some elements of the path (whose right edge was taken on the path)

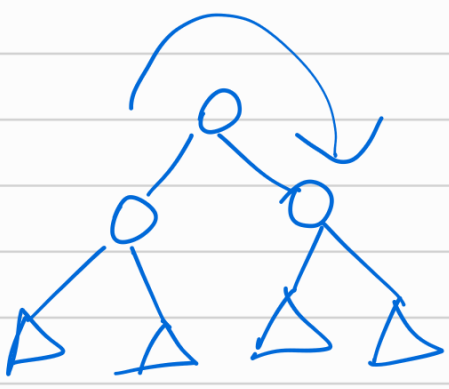
size (v) : the number of nodes stored in the subtree rooted at v.

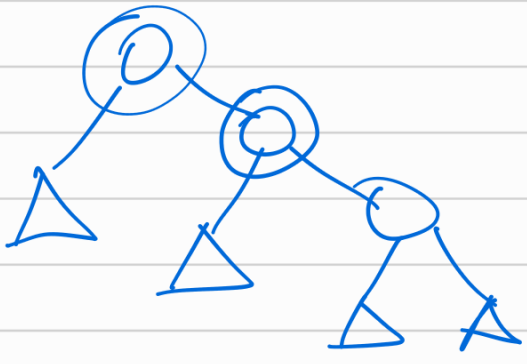
$$size(v) = size(v \to left) + size(v \to right) + 1$$

rank = 0
right link of some node v :

$$rank \leftarrow rank + size(left(v)) + 1$$

left link → nothing.





→ Orthogonal Range Search

Size = $O(n \log n)$ Query = $O(K + \log^2 n)$

Preprocessing time = $O(n \log n)$

↓ to list out the $O(K)$ points.

→ Dynamic Sequences

i^{th} → index

i) Insert (D, i, x) : insert element x at i^{th} place in the sequence.

ii) Delete (D, i) : delete i^{th} element from the seq.

iii) Report (D, i) : Report i^{th} element " " " .

---

iv) Min (D, i, j) : report min from e_i ... e_j (min)

v) Add (D, i, j, x) : add x to all elements e_i — e_j

vi) Flip (D, i, j) : Report i^{th} element. (incr)

xor – bit : flip = xor with 1.

⇒ keep doing xor in the path.

```
Report (T, i) {
    if (T->left == NULL)  s = 0;
    else  s = T->left->size;
    if (i = s+1)  return T->val;
    else if (i < s+1) return Report (T->left, i);
    else return Report (T->right, i-s-1); }
```

```
Insert (T, i, x) {
    if (T == NULL) {
        v = new node (x); v->val = x;
        v->left = v->right = NULL;
        v->size = 1;  return v; }
    if (T->left == NULL) s = 0;
    else s = T->left->size;
    if (i ≤ s+1)  T->left = Insert (T->lt, i, x)
```

size(T) = size(T) + 1 ←

```
    else  T→right = Insert(T→rt, i-s-1, x);
  return T; }
```
rebalance at the end.

```
Add(T, i, j, x){
  u ← Search(T, i);
  v ← Search(T, j);
  w ← LCA(u, v);   val(w) += x;
    if (u ≠ w) {
        val(u) += x;
      if (right(u) ≠ NULL)  incr(r(u)) += x
    while (par(u) ≠ w) {
     if (u = par(u) → left) {
      val(par(u)) += x;
        incr(par(u) → right) += x; }
     u = par(u); }

 if (v ≠ w) {
   val(v) += x;    incr(left(v)) += x;
 while (par(v) ≠ w){
    if (v = par(v) → right) {
     val(par(v)) += x;
    incr(par(v) → left) += x; }
      v = par(v); } }
```

```
Report(T, i)
    if (T→left = NULL) size=0;
      else size = (T→left)→size;
  if (i == size+1)  return T→val + incr(T);
 else if (i < s+1)  return incr(T) + Rep(T→l, i)
   else return incr(T) + Rep(T→r, i-s-1);
```

```
Insert (T, i, x)
  if (T == NULL){
     v ← new node( );  v→val = x;
  v→incr = 0;  v→lt = v→rt = NULL;
  v→size = 1;  return v; }
  size(T)++;
 if (left(T) == NULL)  s = 0;
  else  s = size(left(T));
 if (i ≤ s+1)
     T→left = Insert(T→left, i, x - incr(T))
else T→right = Insert(T→right, i-s-1, x-incr(T))
  return T; }
```

Interval Trees

An interval $[a,b] = \{x \in R \mid a \le x \le b\}$

For an interval $I = [a,b]$

high[I] = b low[I] = a

check if two intervals overlap ⇒ O(1) time.

if ($a \le c \le b$ or $a \le d \le b$ or $c \le a \le d$ or $c \le b \le d$) True;

Insert(S, I): insert interval I into S.
Delete(S, I): delete I from S.
Overlap(S, I): if I overlaps any interval from S.

- Create BST according to the starting point of each interval. Augment it with the maximum finish time in its subtree.

- Max-high(x) =

max (high(u), Max-high(u→left), Max-high(u→right))

- **Lemma**: If max high of any interval in $L \geq$ Low(I) and I overlaps with some interval in T, then I overlaps with an interval in subtree L.

```
Overlap (T, I) {
  found = false ;  u ← T;
  while (u ≠ Null and found == 0){
    if (I overlaps Interval(u)) found = 1;
    else if (left(u) ≠ Null and
                Max-high > Low(I)) u = left(u);
    else u = right(u);
```