Pseudo code

Noun explanation:

size: the total number of people under(left, right child) and its own ID size(people sharing same data)

ID: the set in every node to keep all ID with same data

```
LEFT_ROTATE(x)
    Same as ppt
    size(y)=size(x)
    size(x)=size(left(x))+size(right(x))+ID(x).size()
same as RIGHT_ROTATE(x)

Successor(Node x)=Tree_Minimum(right(x))

RB_Select(Node n, int rank)
    m=size(right(n))+1
    k= size(right(n))+ID(n).size()
    if i=m
        then return n
        else if i>m and i<=k
            then return nil
        else if i>k
            then return RB_Select(left(n),i-k)
        else return RB_Select(right(n),i-k)

RB_Search(int data, Node** p)//return the parent node together
    *p=nil
    x=root
    while x!=nil
        *p=x
        if data = data(x)
            then return x;
        if data<data(x)
            then x=left(x)
            else x=right(x)
    return nil

RB_Insert(int iID, int idata)
```

```
    y=RB_Search(data,&p)
    if y!=nil
        ID(y).insert(iID)
        while y!=nil
            size(y)++
            y=parent(y)
        return
        else
            RB_Insert(z) in the ppt
            fix_p=z
            while fix_p!=nil
                size(fix_p)=size(left(fix_p))+ size(right(fix_p))+ID(fix_p).size()
                fix_p=parent(fix_p)
        InsertFixUpRBT(z)

RB_Delete(int dID, int ddata)
    z=RB_Search(ddata,&p)
    if z=nil
        then return nil
        else if ID(z).size()>1
            ID(z).earse(dID)
            while z!=nil
                size(z)--
                z=parent(z)
            return
            else
                RB_DELETE(z) in the ppt
                if y!=z            //y is the actual delete node
                    data(z)=data(y)
                    delete ID(z)
                    ID(z)=ID(y)
                size(nil)=0
                fix_p=parent(x)    //x is the child node of the actual delete node y
                while fix_p!=nil
                    size(fix_p)=size(left(fix_p))+ size(right(fix_p))+ID(fix_p).size()
                    fix_p=parent(fix_p)
                if color(y)=BLACK
                    then DeleteFixUpRBT
```

```
                        delete y
int main(int argc, char** argv)
    switch:
        case 'I': RB_Insert
        case 'D': RB_Delete
        case 'R': RB_Select(root, rank), print first ID in the node
        case 'V': print size(root)-RB_Rank(n)+1
        case 'B':m1=min(n1,n2);m2=max(n1,n2)
                RB_Search(m1,&p); RB_Search(m2,&p);
                m1++,m2--until find not nil
        case 'A': m1=min(n1,n2);m2=max(n1,n2)
                nmin=RB_Select(root,m2);nmax=RB_Select(root,m1);
                m1++,m2--;until find not nil
                print first ID in nmin,nmax and the data
```

Time complexity

Case 'I': call RB_Insert O(1), RB_Search O(lgn)(try to find the node from root to leaf)

InsertFixUpRBT() O(lgn)(fix from the deleted leaf to root)

➔O(1)+O(lgn)+O(lgn)=**O(lgn)**

Case 'D':call RB_Delete, RB_Search, DeleteFixUpRBT, time complexity all same with 'I'

➔O(1)+O(lgn)+O(lgn)=**O(lgn)**

Case 'R': call RB_Select **O(lgn)**(try from the root to find the node with correct rank)

Case 'V': call RB_Rank→RB_Search+fix nodes size O(lgn)(from the node to root)

➔O(lgn)+O(lgn)=**O(lgn)**

Case 'B': need an additional variable 'k' to count the total time trying to find the node with the existing data by m1++,m2--, call RB_Search O(lgn) every time maybe affect by the sparsity of the data of the nodes

call RB_Rank O(lgn)

➔k*O(lgn)+O(lgn)=**O(klgn)**

Case 'A': same as 'B', need a k to count the time trying to find the node with the rank By m1++, m2--, call RB_Select O(lgn) every time, affect by the number of IDs in single node

➔k*O(lgn)=**O(klgn)**