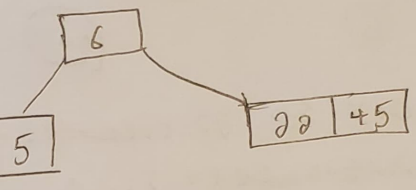
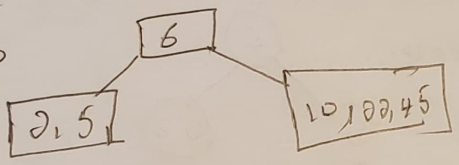
Ans - 5, 6, 22, 45, 2, 10, 18, 30, 50, 12, 1, 7, 55

→ 1. insert 5 [5] → insert 6 [5,6] → insert 22

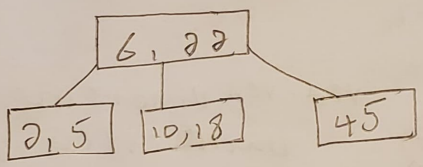
[5,6,22] → insert 45



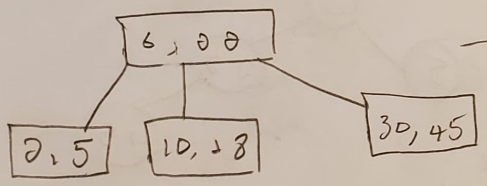
insert - 2
insert - 10



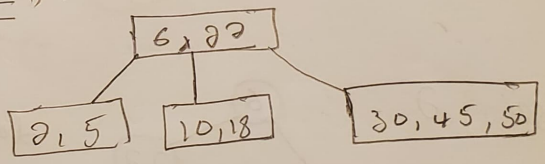
insert - 18



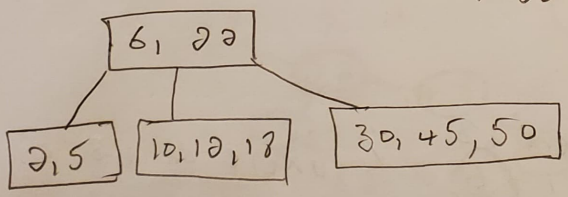
insert - 30



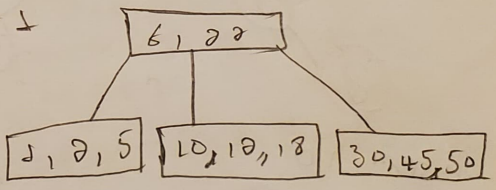
⟹ insert 50



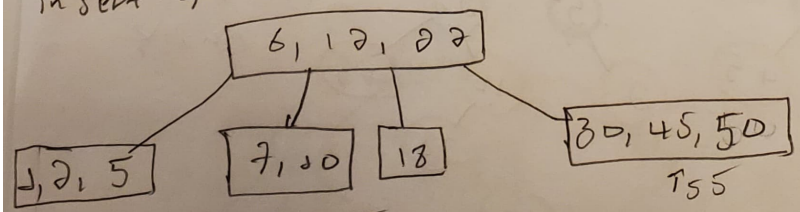
insert - 12



insert - 1

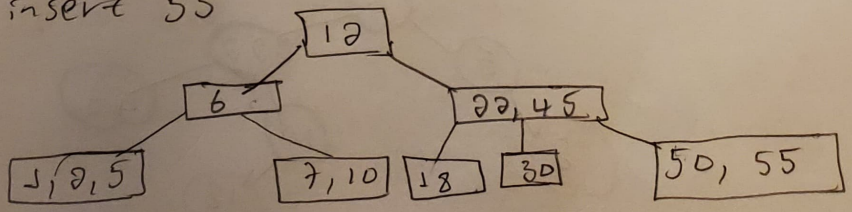


insert 7



↑55

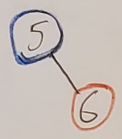
then insert 55



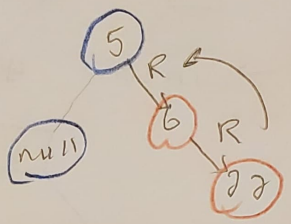
- 5, 6, 22, 45, 2, 10, 18, 30, 50, 12, 5, 7, ...

- insert - 5 → (5) this root change to black (5)

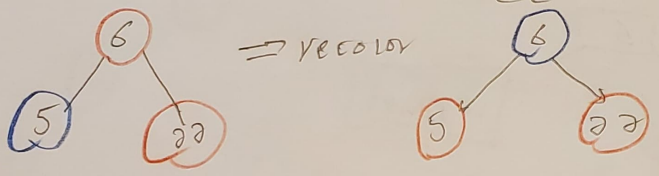
- insert 1



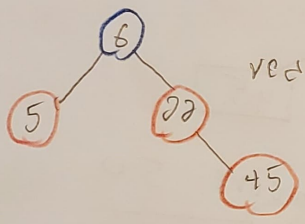
insert 22



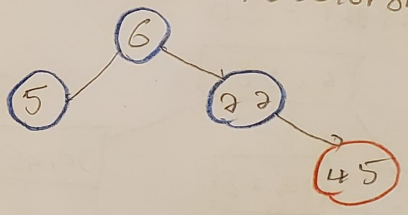
uncle of 22 - black so 22 - creates the problem make single Rotation to left side



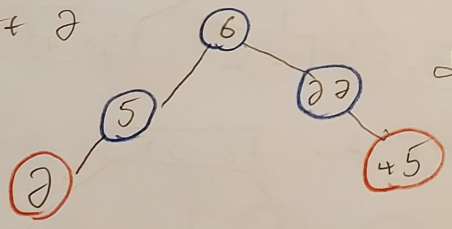
insert 45



red-red violate the rule and its uncle is red so recolor only

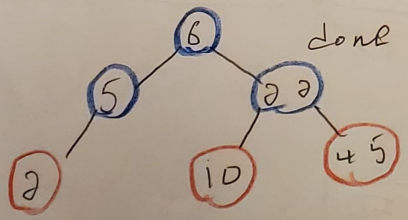


insert 2



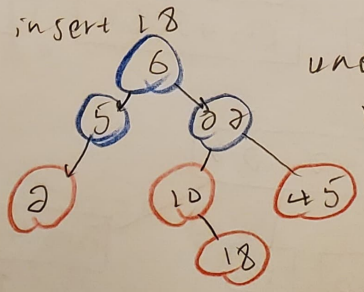
done
no violation here

insert 10

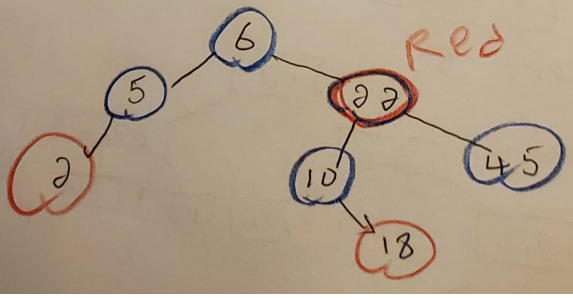


done

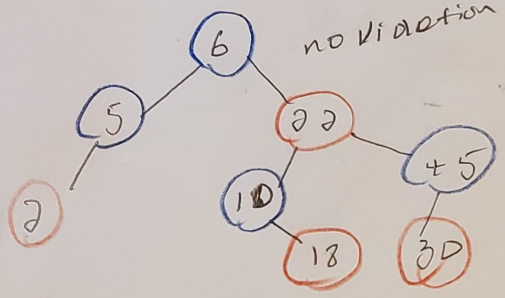
insert 18



uncle of 18 is red recolor only

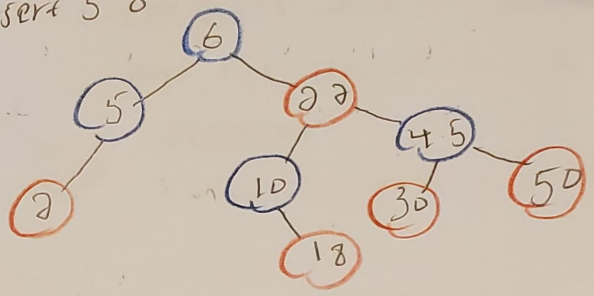


insert -30

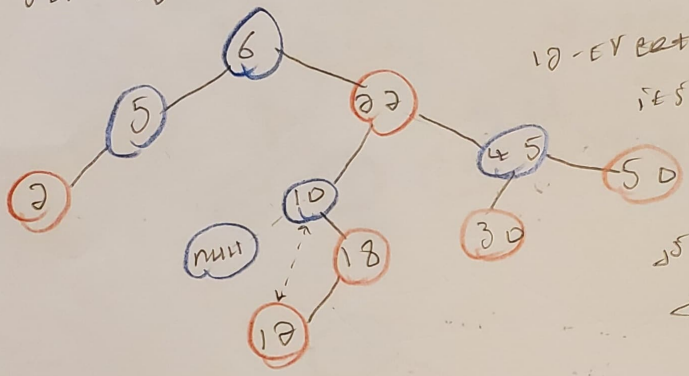


no violation

insert 50



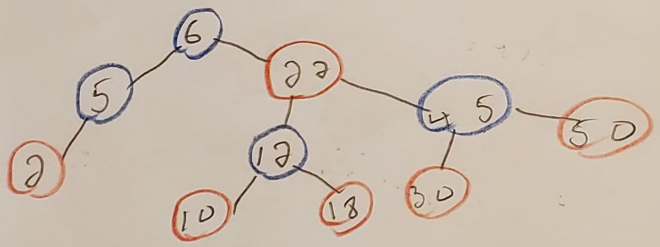
insert 12



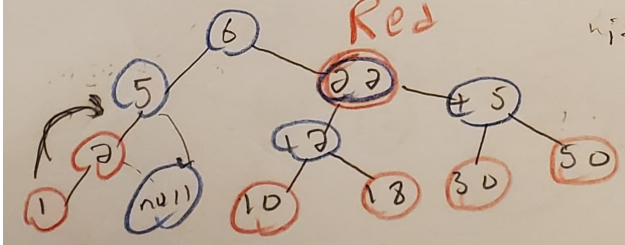
12 - creates the problem and its uncle is null so null is black

double rotation

1st rotate right then left

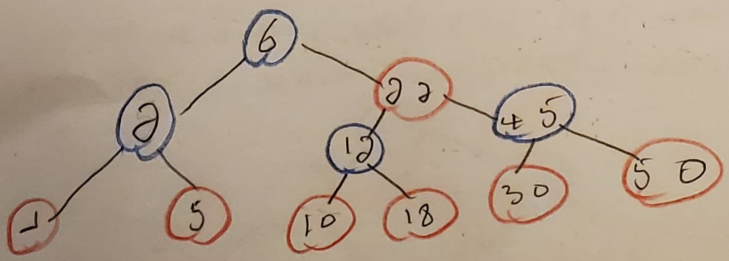


insert 1

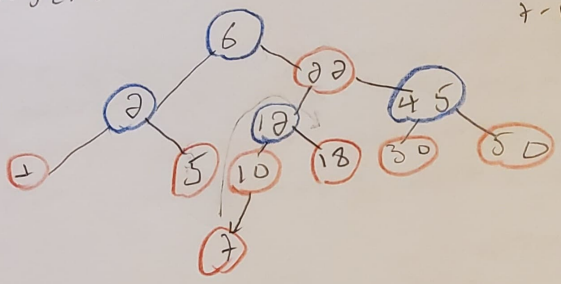


1 - creates the problem with its uncle is null and black

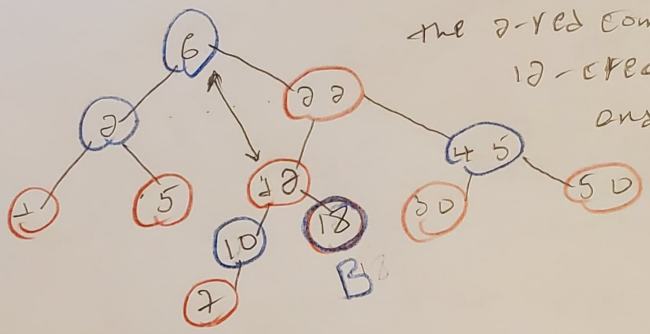
LL - single rotation



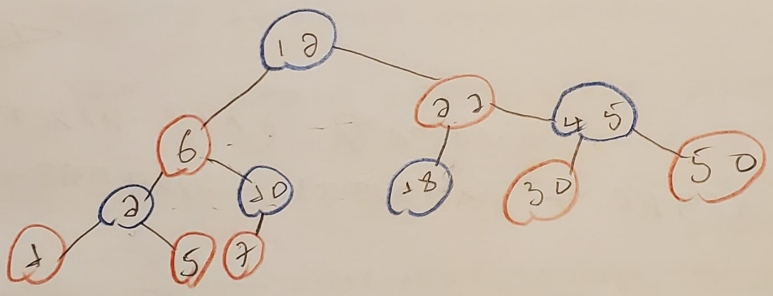
* insert - 7



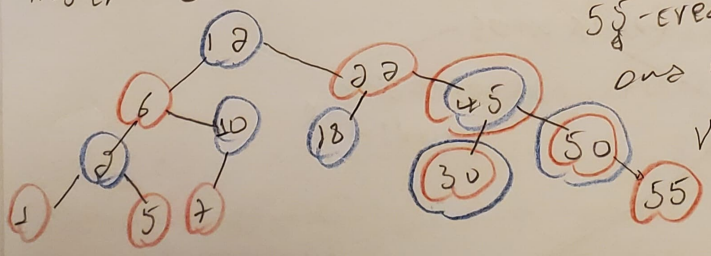
7-creates the problem
ans its uncle is red
recolor



the 2-red comes together if
12-create the problem
ans its uncle is black
Right-left
rotation

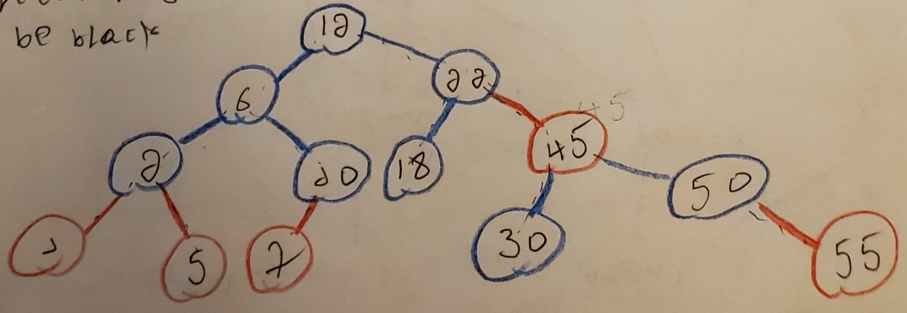


insert 55



55-creates the problem
ans its uncle is red
recolor 45-red
30-black
50-black

ans 22 & 45 are both red its uncle also red
then recolor 6-black, 22-black, 12-red but root
should be black



R-3.14 For each of the following statements about red-black trees, determine whether it is true or false. If you think if it is true, provide a justification. If you think it is false, give a counterexample.

a. a subtree of a red-black tree is itself a red-black tree.
b. the sibling of an external node is either external or it is red.
c. given a red-black tree *T*, there is an unique (2,4) tree *T'* associated with *T*. d. given a (2,4) tree *T*, there is an unique red-black tree *T'* associated with *T*.

Answer

a .is False it may not be because sometimes subtree of red black tree the root can be red but always in the red black tree the root must be black

b. true because all sibling of external node is external or red

c. true there is unique (2,4) tree for a given red black tree

d. false there is no unique red black tree for given of 2-4 tree for a given 2-4 tree we can have different red black tree that have different structure

4. Assume the elements in A and B cannot be sorted, i.e., there is no comparator. How would this restrict the way you would have to implement a solution to isPermutation(A,B), i.e., which of the above strategies could you use and which couldn't you use?

HashTable is best we do not need to sort the elements

Algorthim isPermutatinHT(A,B)

If(A.length!==B.length). 1

Return false; 1

DA:=new HashTable() 1

DB:=new HashTable() 1

insertArrayToHT(A,DA) n

insertArrayToHT(B,DB) n

iterA:=DA.items(); 1

while iterA.hasNext() then do n

a:=iterA.nextObject(); n

aValue:=DA.findValue(a.key()). n*1

bValue:=DB.findValue(a.key()). n*1

if (aValue!==bValue V bValue===null). n

return false; 1

return true. 1

Time complexity O(N)

```
function isPermutationHT(A, B) {
    let DA = new Map.HT_Dictionary();
    let DB = new Map.HT_Dictionary();
    insertArrayToHT(A,DA)
    insertArrayToHT(B,DB)
    if(DA.size()!==DB.size())
       return false;
```

```
  let iterA=DA.items();
  while(iterA.hasNext()){
        let  a=iterA.nextObject()
         let aValue=DA.findValue(a.key())
         let bValue=DB.findValue(a.key())
         if(aValue!==bValue)
             return false;
}
return true
}
```

Algorthim insertArrayToHT(arr,D)

for(n of arr) then do n

cnt:=D.findValue(n). n*1

if(cnt===null). n*1

D.insertItem(n,1). n*1;

else

cnt:=cnt+1. n*1;

D.insertItem(n,cnt). n*1;

Time complexity is O(N)

```
function insertArrayToHT(arr, D) {
    let key;
    for(let n of arr){
      key=D.findValue(n)//return the values of the given
id
      if(key===null)
```

```
            D.insertItem(n,1)
        else{
            key=key+1;
            D.insertItem(n,key)
        }
    }
}
```

Algorthim isPermutatinPQ(A,B)

PQA:=new PriorityQueue(). 1

PQB:=new PriorityQueue(). 1

insertArrayToPQ(A,PQA). n*logn

insertArrayToPQ(B,PQB). n*logn

If(PQA.size()!==PQB.size()). 1

return false; 1

while !PQA.isEmpty() the do n

a:=PQA.removeMin(). n*1

b:=PQB.removeMin(). n*1

if(a!==b) n

return false; 1

return true 1

Time complexity is O(N*logN)

```
function isPermutationPQ(A, B) {
    let PQA = new PQ.PriorityQueue();
    let PQB = new PQ.PriorityQueue();
    insertArrayToPQ(A,PQA)
    insertArrayToPQ(B,PQB)
    let a,b;
    if(PQA.size()!==PQB.size())
       return false;
    while(!PQA.isEmpty()){
       a=PQA.removeMin()
       b=PQB.removeMin()
       if(a!==b)
         return false;
    }
    return true;
}
```

Algorthim insertArrayToPQ(arr,PQ)

For(n of arr) then n

PQ.insertItem(n,n). n*logn

Time complexity is O(N*logN)

```
function insertArrayToPQ(arr,PQ){
    for(let n of arr){
        PQ.insertItem(n,n)
    }
```

Algorthim isPermutationBST(A,B)

DA:=new OrderedDctionary(). 1

DB:=new OrderedDctionary(). 1

insertArrayToBST(A,DA). nlogn

insertArrayToBST(B,DB). nlogn

iterA:=DA.items(); 1

iterB:=DB.items(); 1

while(iterA.hasNext()) then do n

a:=iterA.nextObject(). n

b:=iterB.nextObject(). n

if(a.key()!==b.key()) n

return false

if(a.value()!==b.value(). n

return false; 1

return true. 1

return true. 1

Time Complexity is O(NlogN)

```
function isPermutationBST(A, B) {
    let DA = new Tab.OrderedDictionary();
    let DB = new Tab.OrderedDictionary();
    insertArrayToBST(A,DA)
```

```
    insertArrayToBST(B,DB)
    let iterA=DA.items()
    let iterB=DB.items()
    while(iterA.hasNext()){
       let a=iterA.nextObject()
       let b=iterB.nextObject()
       if(a.value()!==b.value())
          return false;
    }
 return true
}
```

Algorthim insertArrayToBST(arr,BS)

For(n of arr) then do n

Cnt:=BS.findVAlue(n). n*logn

if(cnt==null) then n

BS.inserItem(). n*n

else

Cnt:=cnt+1; n

BS.inserItem(n,cnt). n*logn

Time complexity is O(NlogN)

```
function insertArrayToBST(arr,BS){
    for(let n of arr){
        key=BS.findValue(n)//return the values of the
given id
```

```
        if(key===null)
           BS.insertItem(n,1)
        else{
            key=key+1;
            BS.insertItem(n,key)
        }
    }
}
```

Algorthim isPermutationUsingSort(A,B)

If(A.length!==B.length)

return false;

A:=QuikSort(A). n* logn

B:= QuikSort(B). n*logn

i:=0. 1

while(i<A.length). n

if(A[i]!==B[i]. n

return false; 1

i:=i+1; n

return true 1

Time complexity is O(N*logN)

Algorthim height (T)

Return heightHelper(T,T.root())-1

Algorthim heightHelper(T,p)

If(T.isExternal(p)) return 1; n

leftH=1+ heightHelper(T,T.leftChiled(p)). n

rightH=1+ heightHelper(T,T.rightChiled(p). n

if(leftH>rightH). n

return leftH; 1

else return rightH. 1s

Time Complexity is O(N)

```
function height(T) {
    // your code goes here Hint: you need a helpe
    return heightHelper(T,T.root())-1;;
}
function heightHelper(T,p){
    if (T.isExternal(p))
            return 1;
    let leftH=1+heightHelper(T,T.leftChild(p))
    let rightH=1+heightHelper(T,T.rightChild(p))
    if(leftH>rightH)
        return leftH
    else return rightH
}
```

```
height(T) {
        return this.eulerTour(T, T.root());
    }
```

```
    eulerTour(T, p) {
        let leftH=0;
        let rightH=0;
         if (T.isExternal(p)) {
             this.visitExternal(T, p);
         } else {
              this.visitPreOrder(T, p);
             leftH = 1+this.eulerTour(T, T.leftChild(p));
             this.visitInOrder(T, p);
             rightH = 1+this.eulerTour(T,
T.rightChild(p));
             //this.visitPostOrder(T, p);
             if(leftH>rightH) return leftH
             return rightH
         }
         return Math.max(leftH,leftH)
     }
}
```