Balanced search tree- Red Black tree

Red Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black. It's a Binary Search tree structure with extra color field for each node satisfying Red-Black properties.

- 1. Every node is either red or black.
- 2. The root is black.
- 3. Every leaf (NIL) is black.
- 4. If a node is red, then both its children are black.
- 5. For each node, all simple paths from the node to descendant leaves contain the
- same number of black nodes. Red-Black tree with n internal keys nodes have a height of h<=2.log(n+1). Time is O log(n).

6) Sketching of the Red Black tree:

- Merge each red node into black node parent.
- The depth of leaves=black height of net.
- Every internal node has 2-4 children.
- Every leaf has same depth namely the black height of the root.
- Number of leaves= n+1.
- Its main property is the self-balancing because it ensures that any simple path from the root to leaf is not more than twice as long as any other such path.

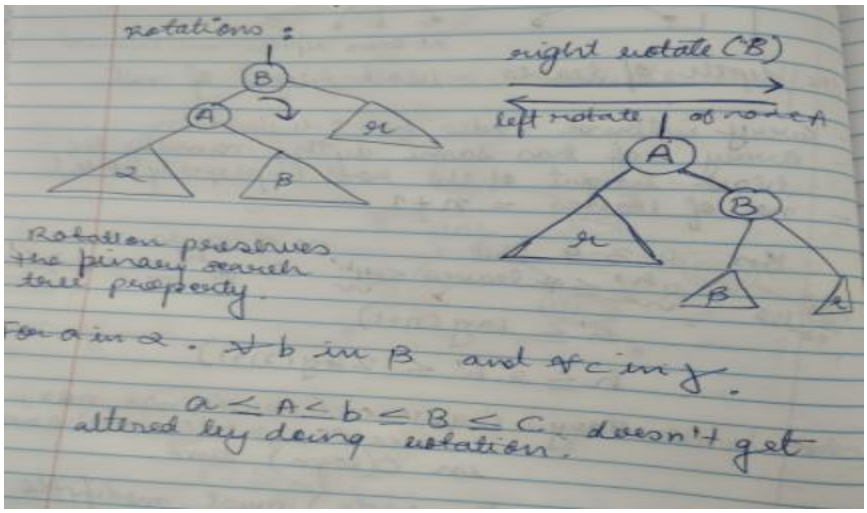
7)Operations under Red Black Tree :

In the Red-Black tree, we use two tools to do the balancing.

- Recoloring
- Rotation
    1. Left rotate : In left rotation the arrangement of node on the right is transformed into the arrangement of the left node.

    2. Right rotate : In right rotation the arrangement of notes on the left is transformed into the arrangement on the right node

    3. Left right and right left rotate : In left right rotation the arrangements are first shifted to the left and then to the right .In right left rotation the arrangements are first shifted to the right and then to the left.



7) Insertion under Red Black tree

- Tree-Insert(x) as a binary search tree
- Color it as red
- If Parent might be red then it can be a problem so move the violation of property 3 up the tree via recoloring until we can fix violation through rotation and recoloring . Property 3 says that every red node should have black parents .

Pseudo. code: Red-Black Insert (T, x)

Tree_Insert (T, x); colour[x] ← RED

while x ≠ root[T] and colour[x] = RED

do if p[x] = left[p[p[x]]] //A

then y ← right[p[p[x]]]

if colour[y] = RED

then <case 1> //recolour

else if x = right[p[x]]

then <case 2>

<case 3>

else //B

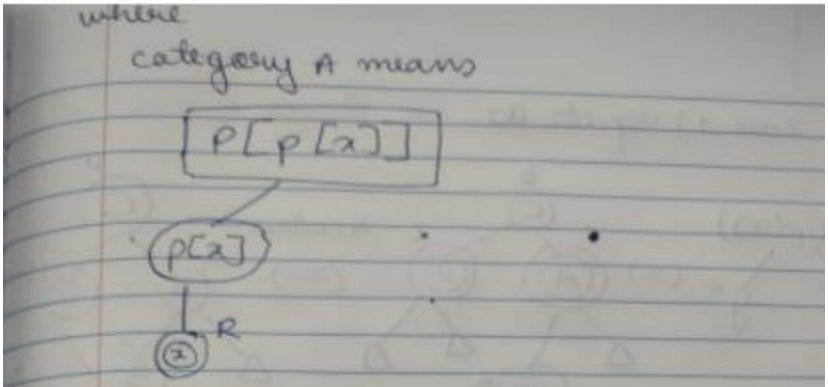
same as (A), but

reverse right ⇔ left

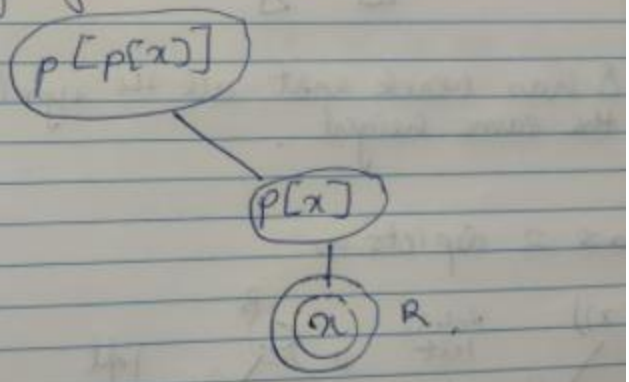
colour[root[T]] ← black

where

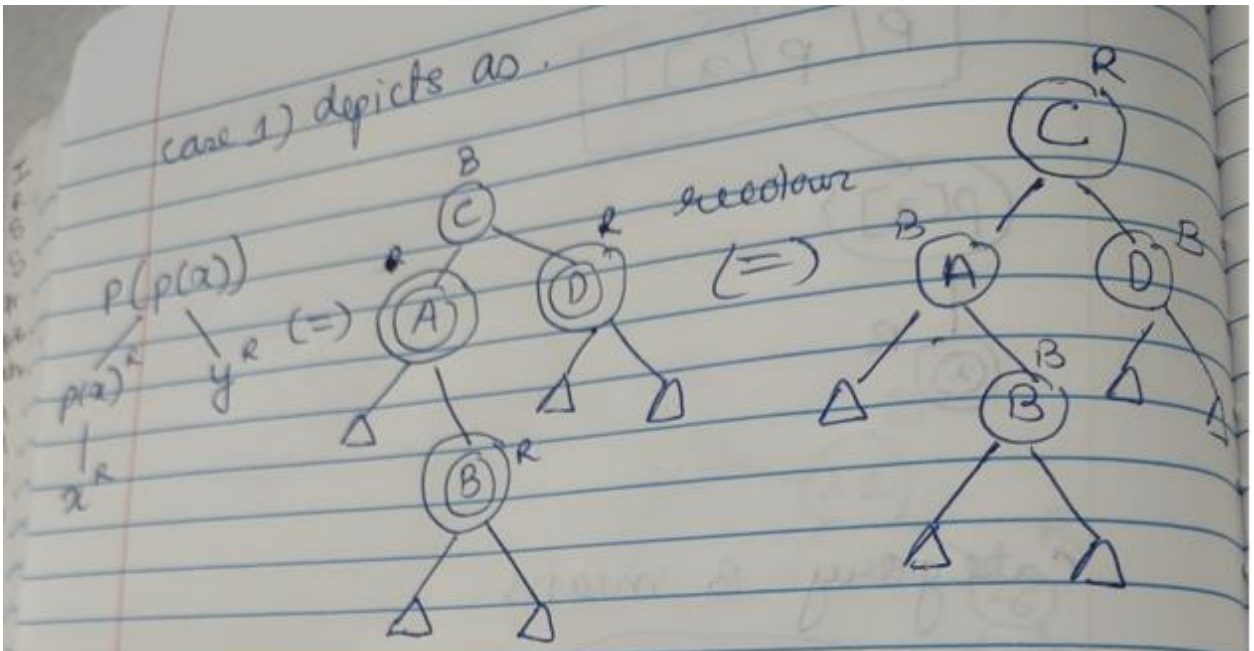
category A means



Category B means

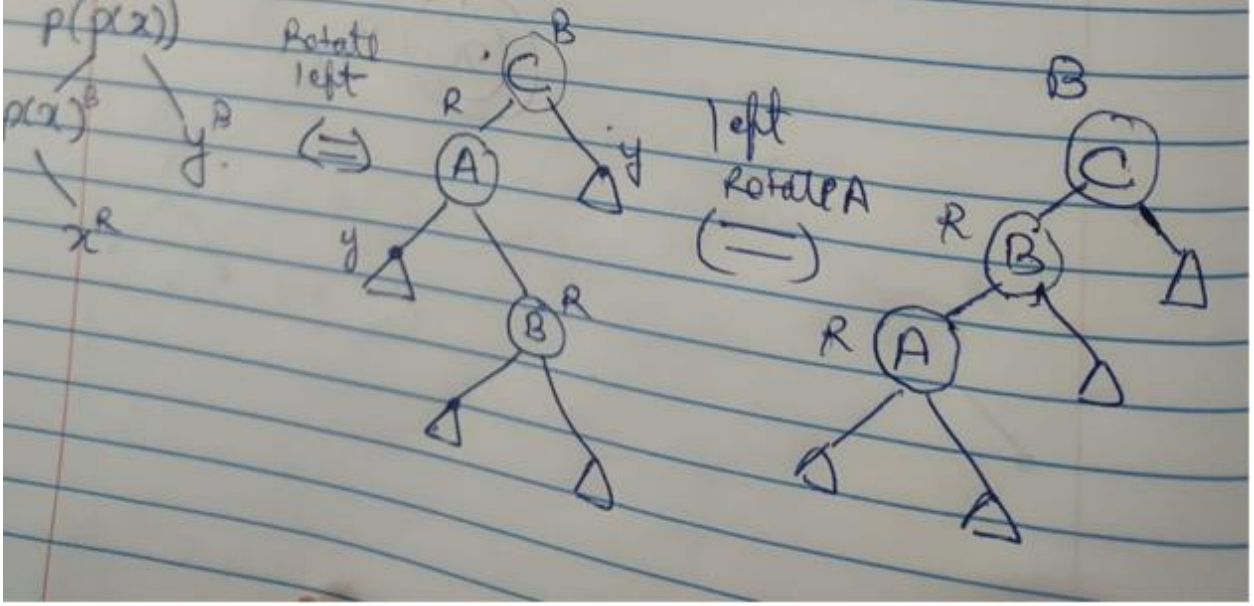


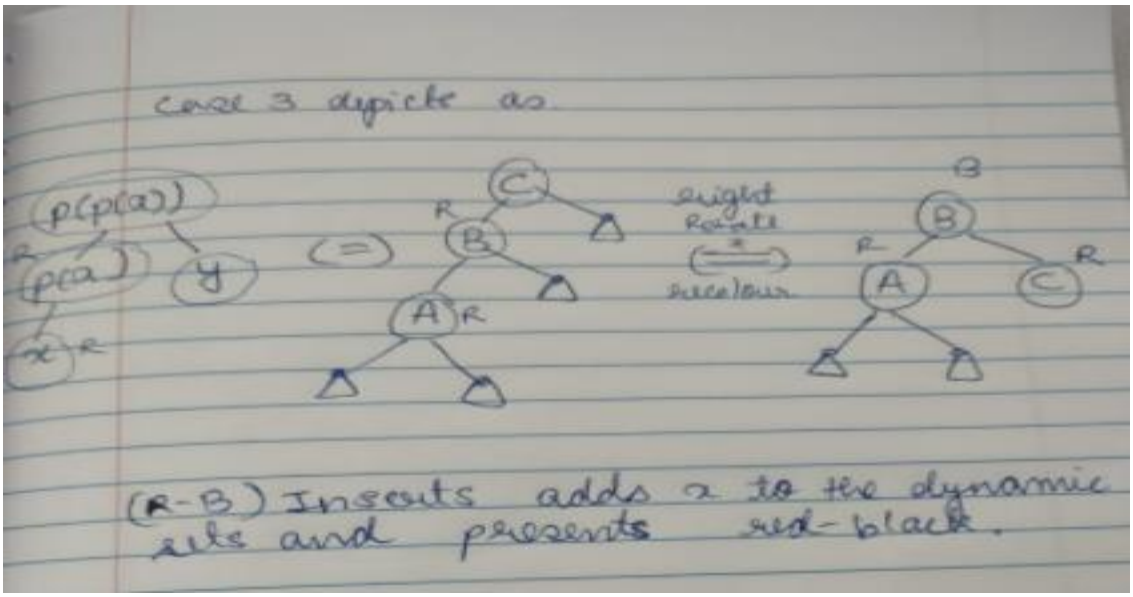
case 1) depicts as.



S has black root all the symbols have the same height.

case 2 depicts as





Code:

```
// Implementing Red-Black Tree in C++


#include <iostream>

using namespace std;

// initializing the node of the binary tree

// every node has left right and parent components

// rbt tree its stores information about color also

struct Node {

  int value;

  Node *parent;

  Node *left;

  Node *right;

  int color;

};
```

```
// creating reference for the structure
typedef Node *Ptr;
//defining a class that has functions of rbt node and insert, rotate
//functions
class createRedBlackTree {
   private:
  Ptr root;
  Ptr BNULL;
 // function to initialize the entities of  node of the structure
  void initnode(Ptr node, Ptr parent) {
    node->value = 0;
    node->parent = parent;
    node->left = nullptr;
    node->right = nullptr;
    node->color = 0;
  }

  // Balancing the tree after insertion
  // Here the value 1 denoted the red color
  //here if m is root change the color to black
  void fixvoilation(Ptr m) {
    Ptr v;
    while ( m!=root && m->parent->color == 1) {
        // checking if m's parent is not black or m is not root
      if (m->parent == m->parent->parent->right) {
```

```
    // if m's uncle is red
    // color of parent and uncle is black
    //color of grandparent is red
    //  change m=m's grandparent for new m
   v = m->parent->parent->left;
   if (v->color == 1) {
    v->color = 0;
    m->parent->color = 0;
    m->parent->parent->color = 1;
    m = m->parent->parent;
   } else {
    if (m == m->parent->left) {
      m = m->parent;
      Rotateright(m);
    }
    m->parent->color = 0;
    m->parent->parent->color = 1;
    Rotateleft(m->parent->parent);
   }
  } else {
   v = m->parent->parent->right;
 // if m's uncle is black then there can be rotations
   if (v->color == 1) {
    v->color = 0;
    m->parent->color = 0;
```

```
      m->parent->parent->color = 1;
      m = m->parent->parent;
    } else {
      if (m == m->parent->right) {
        m = m->parent;
        Rotateleft(m);
      }
      m->parent->color = 0;
      m->parent->parent->color = 1;
      Rotateright(m->parent->parent);
    }
  }
  if (m == root) {
    break;
  }
 }
 root->color = 0;
}
//function to print the tree
void printtree(Ptr root, string alpha, bool end) {
 if (root != BNULL) {
  cout << alpha;
  if (end) {
    cout << "Right ";
    alpha += "   ";
```

```
    } else {
      cout << "Left ";
      alpha += "|  ";
    }

    string sname = root->color ? "red" : "black";
    cout << root->value << "(" << sname << ")" << endl;
    printtree(root->left, alpha, false);
    printtree(root->right, alpha, true);
  }
}

 public:
 //default constructor of the class with default values
createRedBlackTree() {
  BNULL = new Node;
  BNULL->color = 0;
  BNULL->left = nullptr;
  BNULL->right = nullptr;
  root = BNULL;
}

//preorder of binary tree logic
 void preOrderbst(Ptr node) {
  if (node != BNULL) {
```

```
    cout << node->value << " ";
    preOrderbst(node->left);
    preOrderbst(node->right);
  }
}

//inorder function of binary tree logic
 void inOrderbst(Ptr node) {
  if (node != BNULL) {
    inOrderbst(node->left);
    cout << node->value << " ";
    inOrderbst(node->right);
  }
}
// Post order of binary tree logic
void postOrderbst(Ptr node) {
  if (node != BNULL) {
    postOrderbst(node->left);
    postOrderbst(node->right);
    cout << node->value << " ";
  }
}
void Rotateleft(Ptr y) {
  Ptr x = y->right;
  y->right = x->left;
```

```
  if (x->left != BNULL) {
    x->left->parent = y;
  }
  x->parent = y->parent;
  if (y->parent == nullptr) {
    this->root = x;
  } else if (y == y->parent->left) {
    y->parent->left = x;
  } else {
    y->parent->right = x;
  }
  x->left = y;
  y->parent = x;
}

void Rotateright(Ptr y) {
  Ptr x = y->left;
  y->left = x->right;
  if (x->right != BNULL) {
    x->right->parent = y;
  }
  x->parent = y->parent;
  if (y->parent == nullptr) {
    this->root = x;
  } else if (y == y->parent->right) {
```

```
    y->parent->right = x;
  } else {
    y->parent->left = x;
  }
  x->right = y;
  y->parent = x;
}

// Insertion of a node
// a Standard bst insertion is done by ensuring that every node is red in color
void insert(int newval) {
  Ptr node = new Node;
  node->parent = nullptr;
  node->value = newval;
  node->left = BNULL;
  node->right = BNULL;
  node->color = 1;

  Ptr a = nullptr;
  Ptr b = this->root;

  while (b != BNULL) {
    a = b;
    if (node->value < b->value) {
      b = b->left;
```

```
    } else {
      b = b->right;
    }
  }

  node->parent = a;
  if (a == nullptr) {
    root = node;
  } else if (node->value < a->value) {
    a->left = node;
  } else {
    a->right = node;
  }

  if (node->parent == nullptr) {
    node->color = 0;
    return;
  }

  if (node->parent->parent == nullptr) {
    return;
  }
  fixvoilation(node);
}
void printrbt() {
```

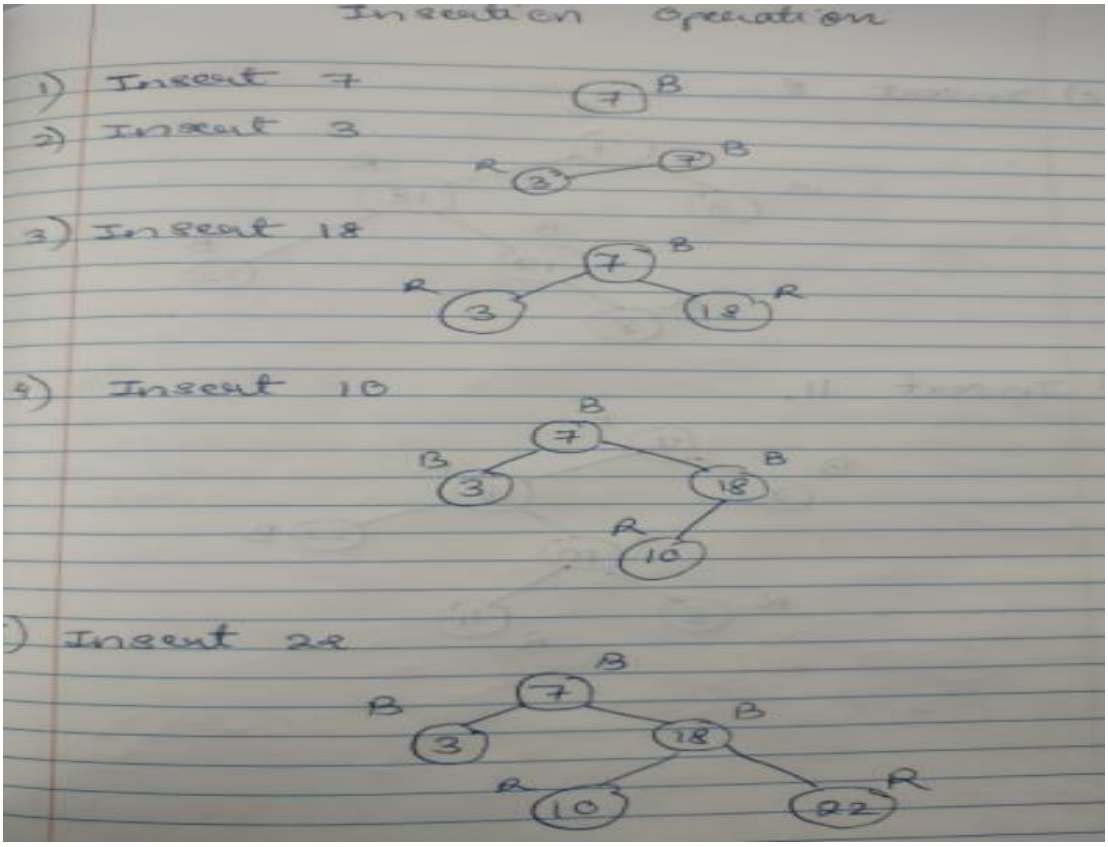
```
    if (root) {
      printtree(this->root, "", true);
    }
  }
};

int main() {
  createRedBlackTree rbt;
  rbt.insert(7);
  rbt.insert(3);
  rbt.insert(18);
  rbt.insert(10);
  rbt.insert(22);
  rbt.insert(8);
  rbt.insert(11);
  rbt.insert(26);
  rbt.insert(15);
  rbt.printrbt();
  cout << endl;
}
```

OUTPUT:

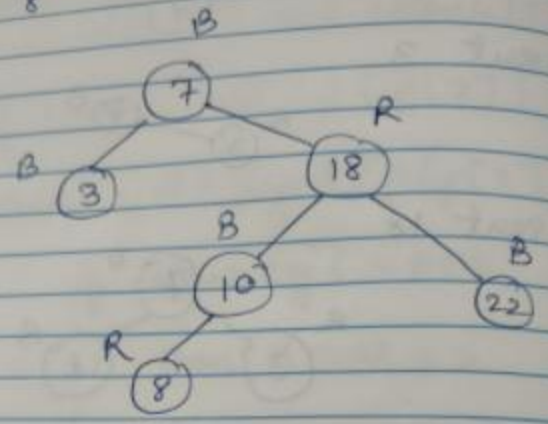
```
Right 10(black)
    Left 7(red)
    |   Left 3(black)
    |   Right 8(black)
    Right 18(red)
        Left 11(black)
        |   Right 15(red)
        Right 22(black)
            Right 26(red)
```

Operation of Insertion code step by step :

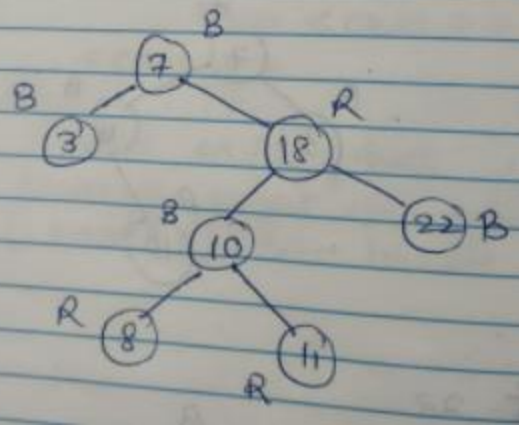
Consider the input array as: [7,3,18,10,22,8,11,26] and if we want to insert 15 below are the steps that depicts the recoloring and rotation of the table performed inside the code .



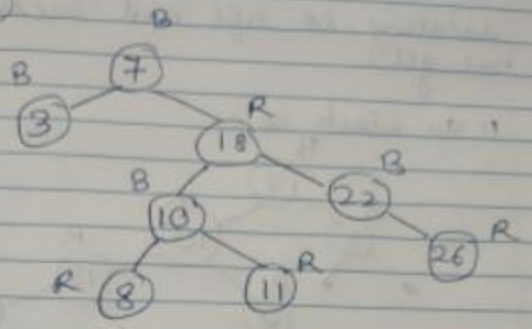
6) Insert 8



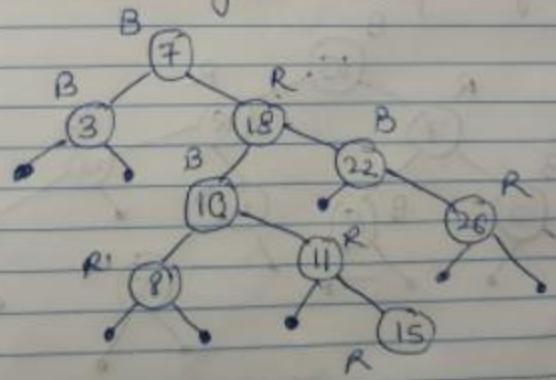
7) Insert 11.



8) Insert (26)



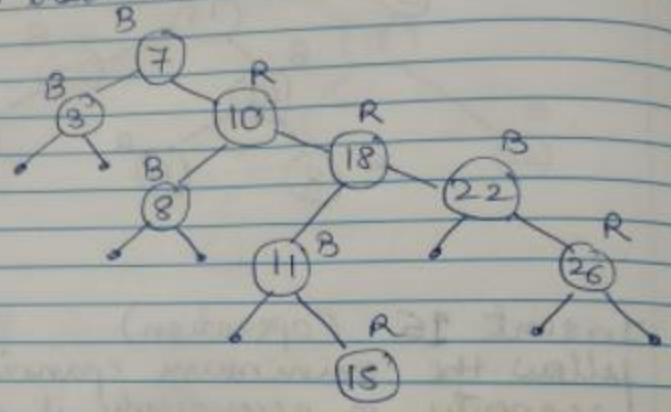
9) Insert 15 (operation)
follow the minimum spanning Tree property so accordingly it will go to the node as per the binary search tree algorithm.



15 is the position where it should be placed as per BST. Further by applying property 3 and 4 and on

rotating to left and recolouring it we get!

11 as black so:



Rotating (7) to left and recolouring we get final red Black tree as