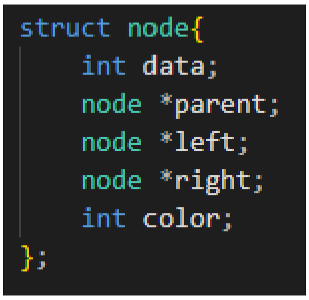
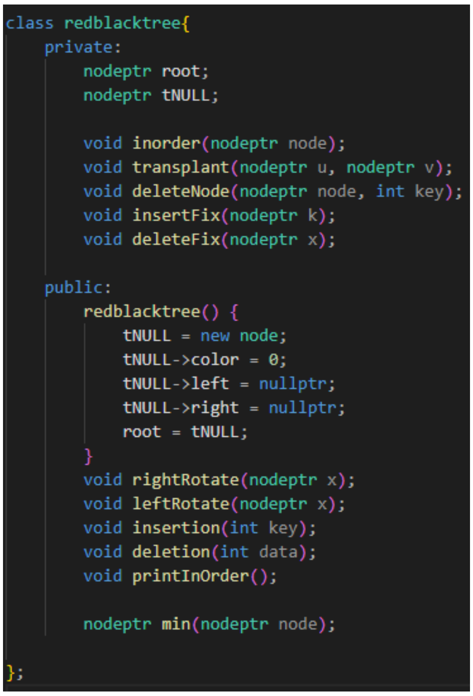
Using VScode / g++
Methods/Solution

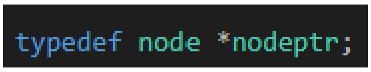
- previous overview(?)



```
struct node{
    int data;
    node *parent;
    node *left;
    node *right;
    int color;
};
```



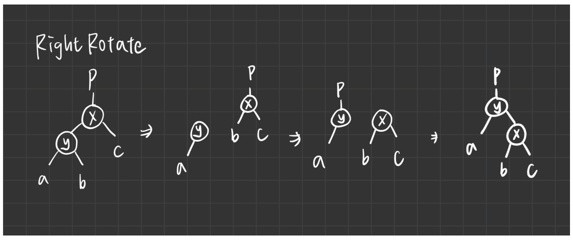
```
class redblacktree{
    private:
        nodeptr root;
        nodeptr tNULL;

        void inorder(nodeptr node);
        void transplant(nodeptr u, nodeptr v);
        void deleteNode(nodeptr node, int key);
        void insertFix(nodeptr k);
        void deleteFix(nodeptr x);

    public:
        redblacktree() {
            tNULL = new node;
            tNULL->color = 0;
            tNULL->left = nullptr;
            tNULL->right = nullptr;
            root = tNULL;
        }
        void rightRotate(nodeptr x);
        void leftRotate(nodeptr x);
        void insertion(int key);
        void deletion(int data);
        void printInOrder();

        nodeptr min(nodeptr node);

};
```

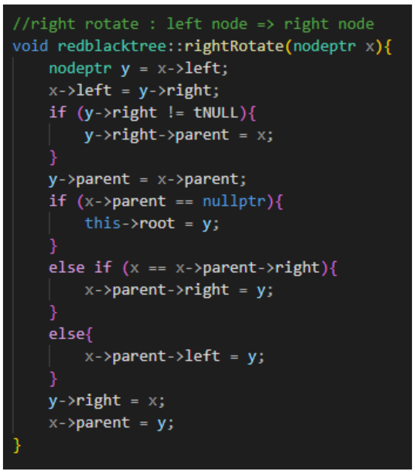


```
typedef node *nodeptr;
```

- some functions I used

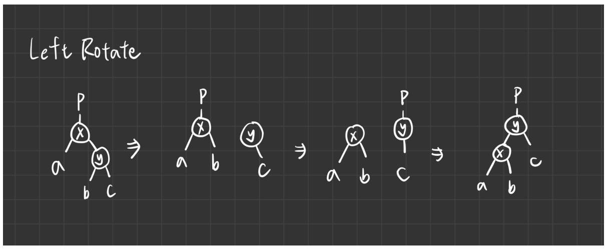
right rotate

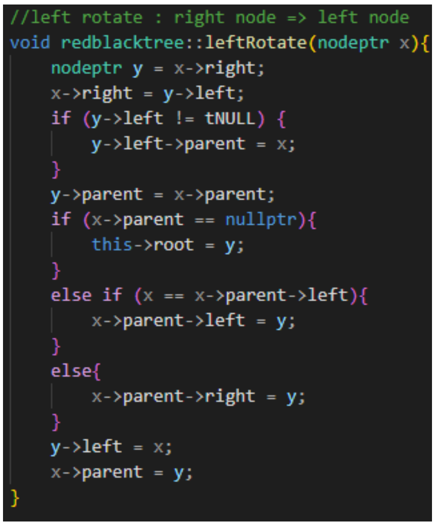




```
//right rotate : left node => right node
void redblacktree::rightRotate(nodeptr x){
    nodeptr y = x->left;
    x->left = y->right;
    if (y->right != tNULL){
        y->right->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr){
        this->root = y;
    }
    else if (x == x->parent->right){
        x->parent->right = y;
    }
    else{
        x->parent->left = y;
    }
    y->right = x;
    x->parent = y;
}
```

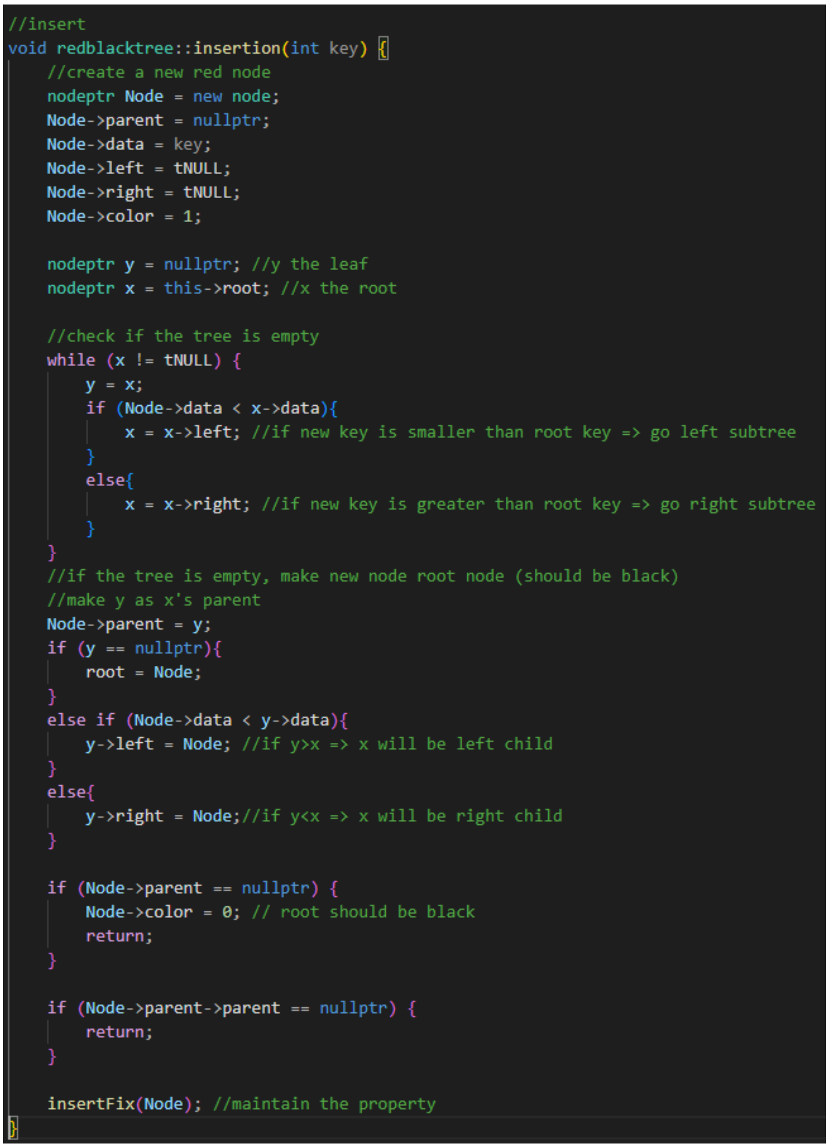
## left rotate



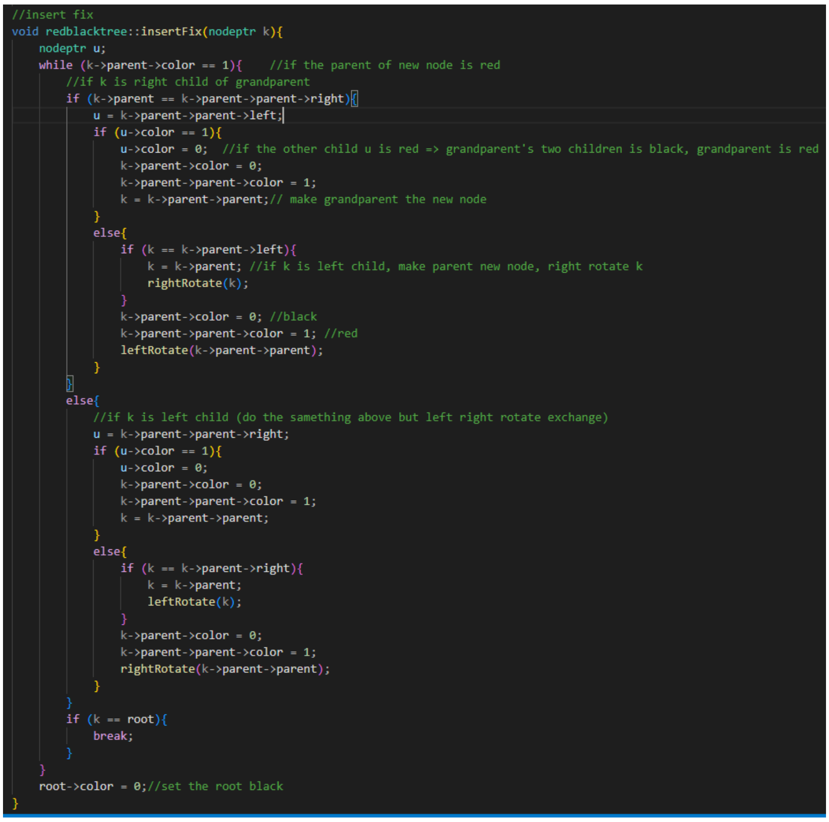


```
//left rotate : right node => left node
void redblacktree::leftRotate(nodeptr x){
    nodeptr y = x->right;
    x->right = y->left;
    if (y->left != tNULL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == nullptr){
        this->root = y;
    }
    else if (x == x->parent->left){
        x->parent->left = y;
    }
    else{
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
```

## insert & insertion fix

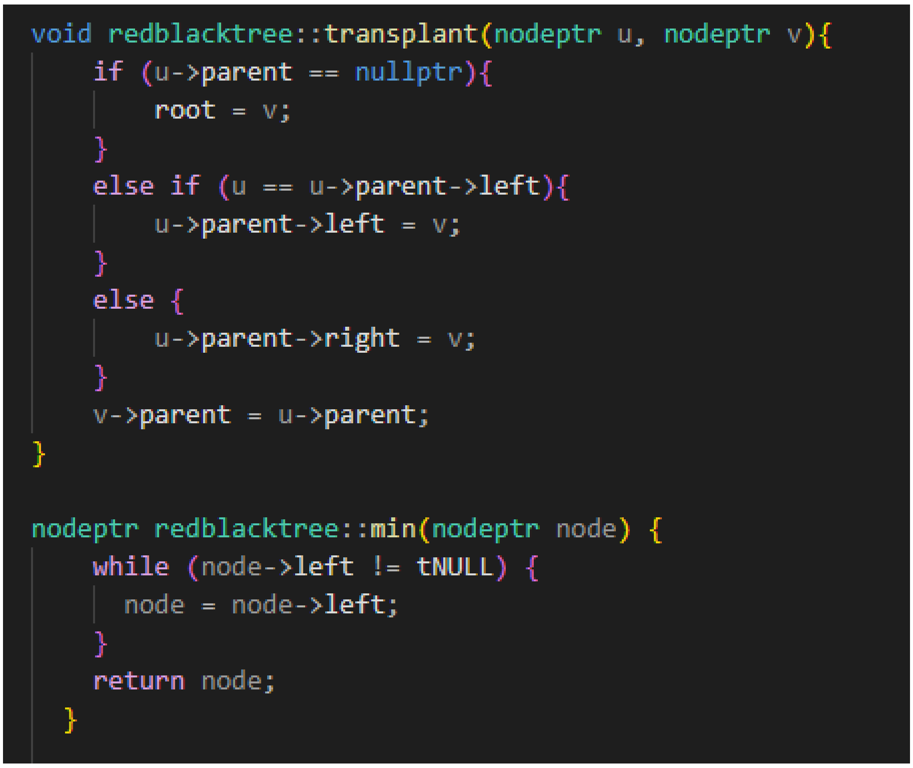


```
//insert
void redblacktree::insertion(int key) {
    //create a new red node
    nodeptr Node = new node;
    Node->parent = nullptr;
    Node->data = key;
    Node->left = tNULL;
    Node->right = tNULL;
    Node->color = 1;

    nodeptr y = nullptr; //y the leaf
    nodeptr x = this->root; //x the root

    //check if the tree is empty
    while (x != tNULL) {
        y = x;
        if (Node->data < x->data){
            x = x->left; //if new key is smaller than root key => go left subtree
        }
        else{
            x = x->right; //if new key is greater than root key => go right subtree
        }
    }
    //if the tree is empty, make new node root node (should be black)
    //make y as x's parent
    Node->parent = y;
    if (y == nullptr){
        root = Node;
    }
    else if (Node->data < y->data){
        y->left = Node; //if y>x => x will be left child
    }
    else{
        y->right = Node;//if y<x => x will be right child
    }

    if (Node->parent == nullptr) {
        Node->color = 0; // root should be black
        return;
    }

    if (Node->parent->parent == nullptr) {
        return;
    }

    insertFix(Node); //maintain the property
}
```



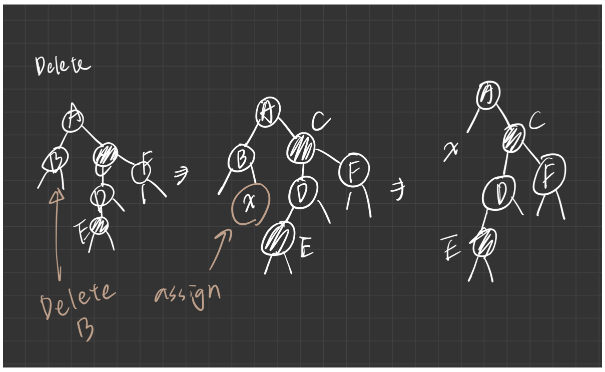
```cpp
//insert fix
void redblacktree::insertFix(nodeptr k){
    nodeptr u;
    while (k->parent->color == 1){    //if the parent of new node is red
        //if k is right child of grandparent
        if (k->parent == k->parent->parent->right){
            u = k->parent->parent->left;
            if (u->color == 1){
                u->color = 0;  //if the other child u is red => grandparent's two children is black, grandparent is red
                k->parent->color = 0;
                k->parent->parent->color = 1;
                k = k->parent->parent;// make grandparent the new node
            }
            else{
                if (k == k->parent->left){
                    k = k->parent; //if k is left child, make parent new node, right rotate k
                    rightRotate(k);
                }
                k->parent->color = 0; //black
                k->parent->parent->color = 1; //red
                leftRotate(k->parent->parent);
            }
        }
        else{
            //if k is left child (do the samething above but left right rotate exchange)
            u = k->parent->parent->right;
            if (u->color == 1){
                u->color = 0;
                k->parent->color = 0;
                k->parent->parent->color = 1;
                k = k->parent->parent;
            }
            else{
                if (k == k->parent->right){
                    k = k->parent;
                    leftRotate(k);
                }
                k->parent->color = 0;
                k->parent->parent->color = 1;
                rightRotate(k->parent->parent);
            }
        }
        if (k == root){
            break;
        }
    }
    root->color = 0;//set the root black
}
```

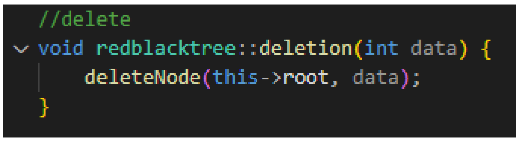
transplant & min (used in deleteNode)



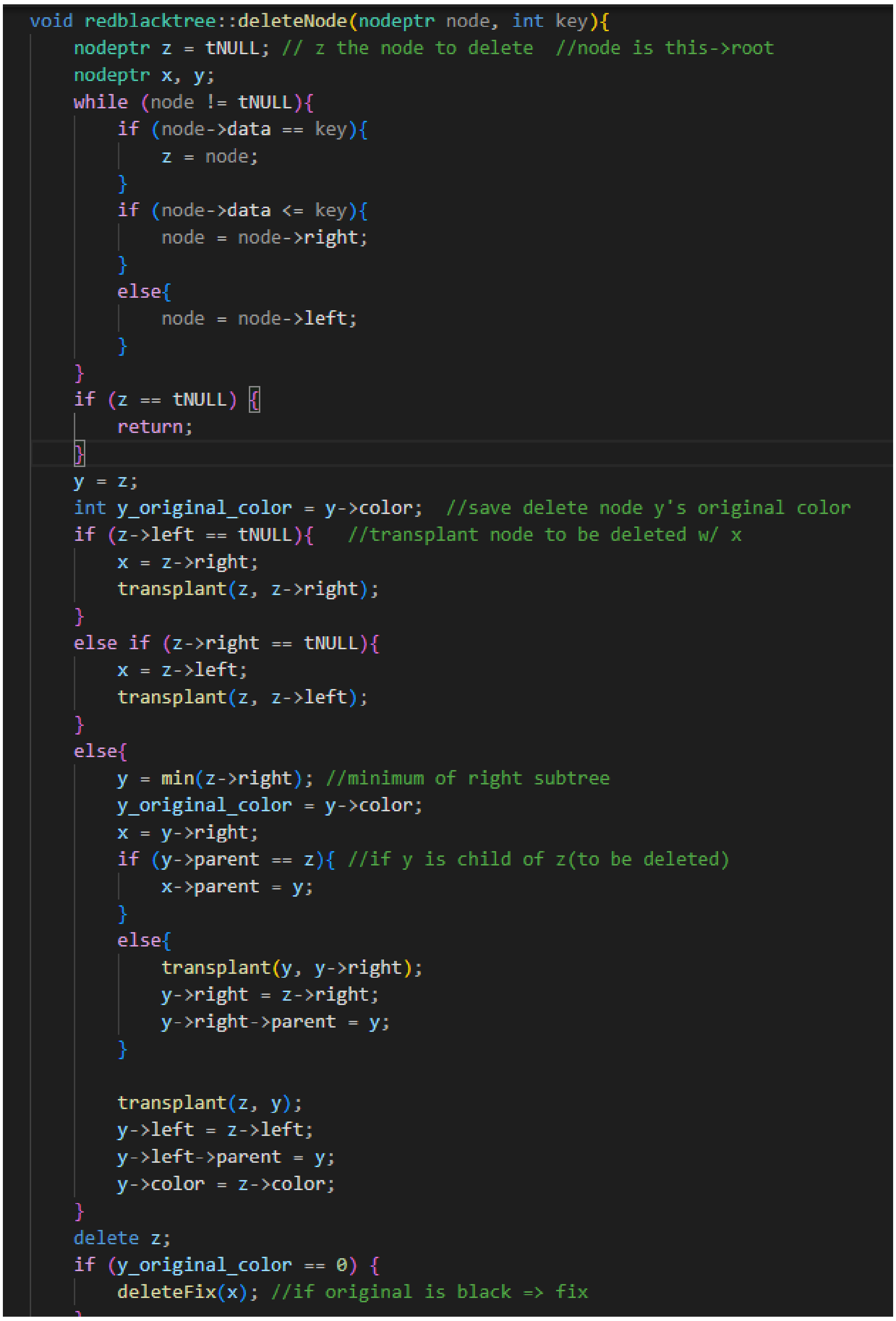
```cpp
void redblacktree::transplant(nodeptr u, nodeptr v){
    if (u->parent == nullptr){
        root = v;
    }
    else if (u == u->parent->left){
        u->parent->left = v;
    }
    else {
        u->parent->right = v;
    }
    v->parent = u->parent;
}

nodeptr redblacktree::min(nodeptr node) {
    while (node->left != tNULL) {
      node = node->left;
    }
    return node;
  }
```

delete & deleteNode & deletion fix

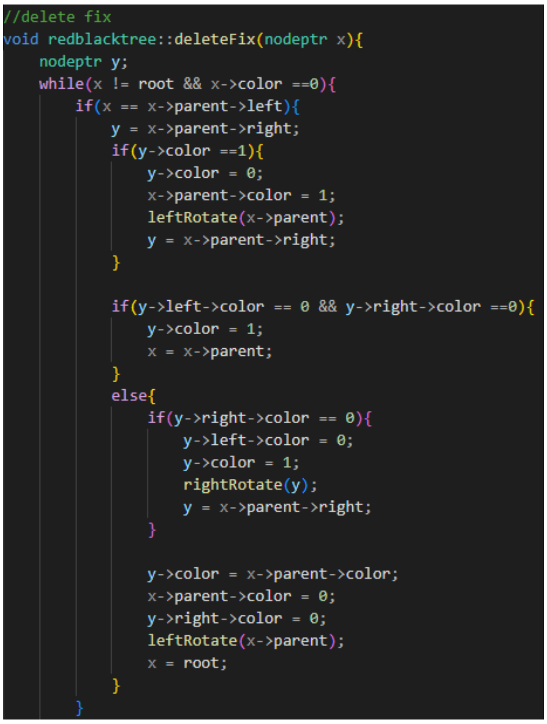




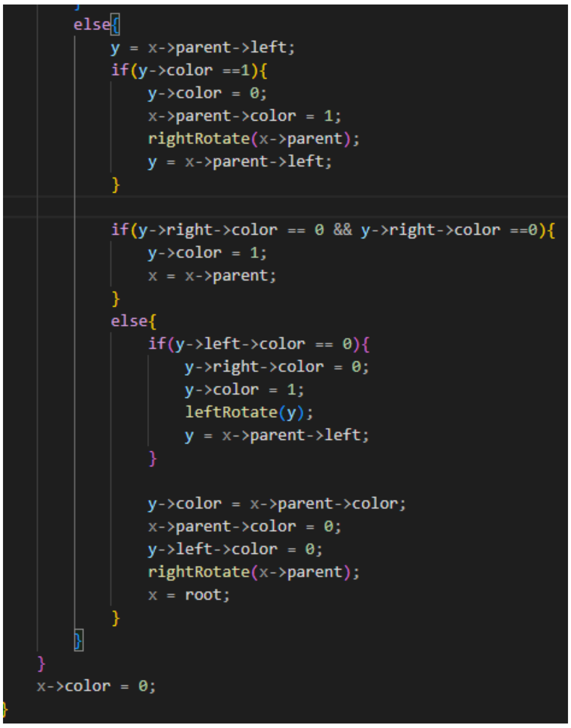
```
//delete
void redblacktree::deletion(int data) {
    deleteNode(this->root, data);
}
```



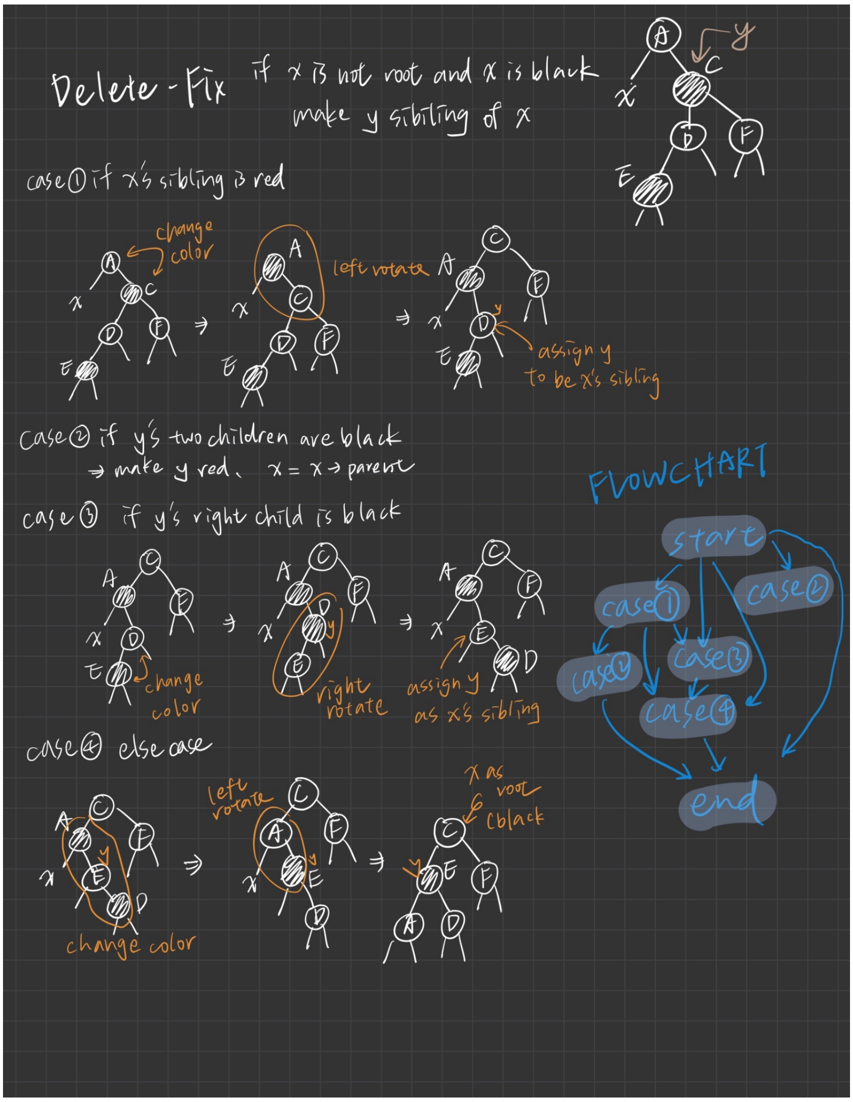
```
void redblacktree::deleteNode(nodeptr node, int key){
    nodeptr z = tNULL; // z the node to delete  //node is this->root
    nodeptr x, y;
    while (node != tNULL){
        if (node->data == key){
            z = node;
        }
        if (node->data <= key){
            node = node->right;
        }
        else{
            node = node->left;
        }
    }
    if (z == tNULL) {
        return;
    }
    y = z;
    int y_original_color = y->color;  //save delete node y's original color
    if (z->left == tNULL){   //transplant node to be deleted w/ x
        x = z->right;
        transplant(z, z->right);
    }
    else if (z->right == tNULL){
        x = z->left;
        transplant(z, z->left);
    }
    else{
        y = min(z->right); //minimum of right subtree
        y_original_color = y->color;
        x = y->right;
        if (y->parent == z){ //if y is child of z(to be deleted)
            x->parent = y;
        }
        else{
            transplant(y, y->right);
            y->right = z->right;
            y->right->parent = y;
        }

        transplant(z, y);
        y->left = z->left;
        y->left->parent = y;
        y->color = z->color;
    }
    delete z;
    if (y_original_color == 0) {
        deleteFix(x); //if original is black => fix
```



```
//delete fix
void redblacktree::deleteFix(nodeptr x){
    nodeptr y;
    while(x != root && x->color ==0){
        if(x == x->parent->left){
            y = x->parent->right;
            if(y->color ==1){
                y->color = 0;
                x->parent->color = 1;
                leftRotate(x->parent);
                y = x->parent->right;
            }

            if(y->left->color == 0 && y->right->color ==0){
                y->color = 1;
                x = x->parent;
            }
            else{
                if(y->right->color == 0){
                    y->left->color = 0;
                    y->color = 1;
                    rightRotate(y);
                    y = x->parent->right;
                }

                y->color = x->parent->color;
                x->parent->color = 0;
                y->right->color = 0;
                leftRotate(x->parent);
                x = root;
            }
        }
```



```
        }
        else{
            y = x->parent->left;
            if(y->color ==1){
                y->color = 0;
                x->parent->color = 1;
                rightRotate(x->parent);
                y = x->parent->left;
            }

            if(y->right->color == 0 && y->right->color ==0){
                y->color = 1;
                x = x->parent;
            }
            else{
                if(y->left->color == 0){
                    y->right->color = 0;
                    y->color = 1;
                    leftRotate(y);
                    y = x->parent->left;
                }

                y->color = x->parent->color;
                x->parent->color = 0;
                y->left->color = 0;
                rightRotate(x->parent);
                x = root;
            }
        }
    }
    x->color = 0;
}
```



Delete-Fix if x is not root and x is black
make y sibling of x

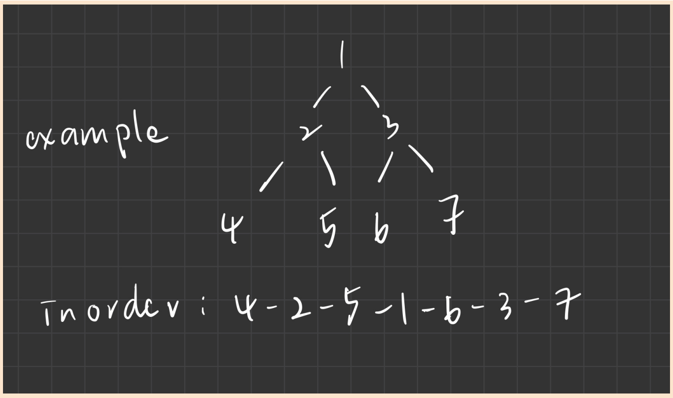
case① if x's sibling is red

case② if y's two children are black
→ make y red, x = x→parent

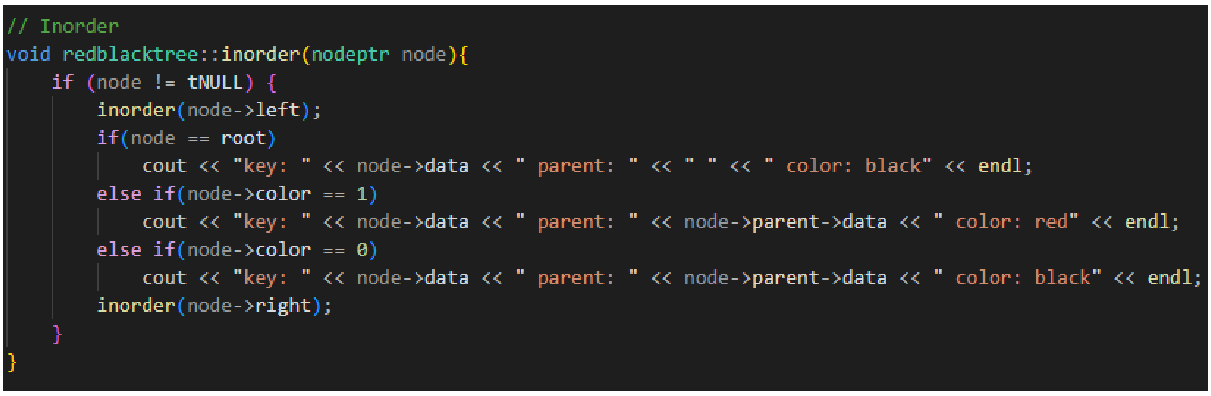
case③ if y's right child is black

case④ else case

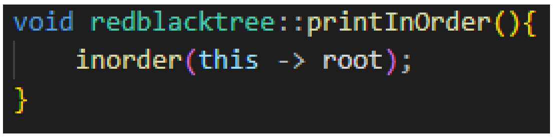
output in order



Inorder : 4-2-5-1-6-3-7



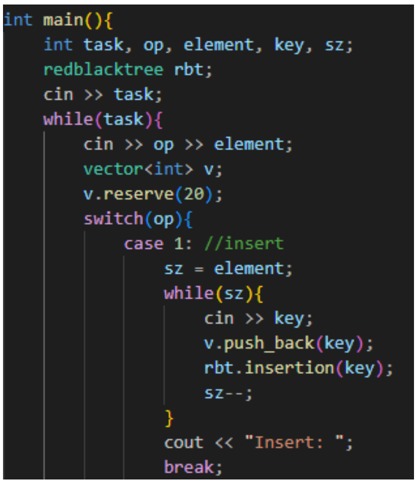
```
// Inorder
void redblacktree::inorder(nodeptr node){
    if (node != tNULL) {
        inorder(node->left);
        if(node == root)
            cout << "key: " << node->data << " parent: " << " " << " color: black" << endl;
        else if(node->color == 1)
            cout << "key: " << node->data << " parent: " << node->parent->data << " color: red" << endl;
        else if(node->color == 0)
            cout << "key: " << node->data << " parent: " << node->parent->data << " color: black" << endl;
        inorder(node->right);
    }
}
```



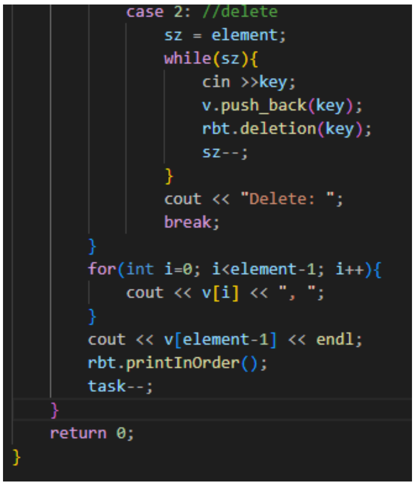
```
void redblacktree::printInOrder(){
    inorder(this -> root);
}
```

main

I used vector to save input numbers, switch cases to choose insert/delete

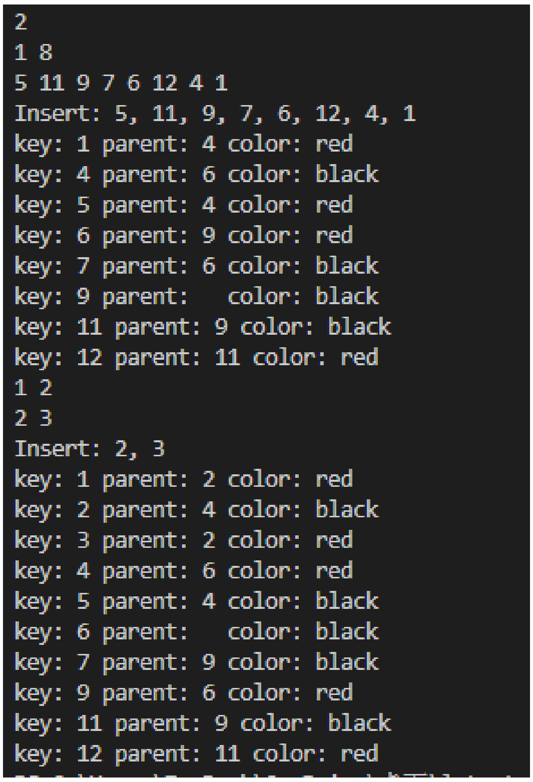


```
int main(){
    int task, op, element, key, sz;
    redblacktree rbt;
    cin >> task;
    while(task){
        cin >> op >> element;
        vector<int> v;
        v.reserve(20);
        switch(op){
            case 1: //insert
                sz = element;
                while(sz){
                    cin >> key;
                    v.push_back(key);
                    rbt.insertion(key);
                    sz--;
                }
                cout << "Insert: ";
                break;
```

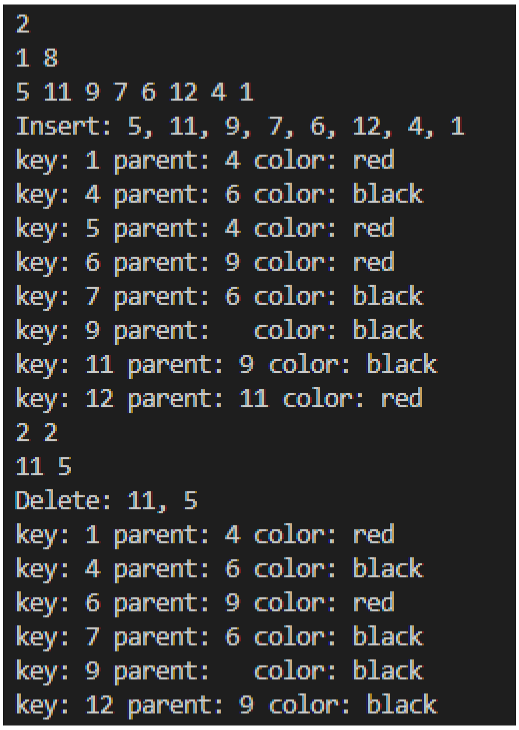


```
            case 2: //delete
                sz = element;
                while(sz){
                    cin >>key;
                    v.push_back(key);
                    rbt.deletion(key);
                    sz--;
                }
                cout << "Delete: ";
                break;
        }
        for(int i=0; i<element-1; i++){
            cout << v[i] << ", ";
        }
        cout << v[element-1] << endl;
        rbt.printInOrder();
        task--;
    }
    return 0;
}
```

input/output



```
2
1 8
5 11 9 7 6 12 4 1
Insert: 5, 11, 9, 7, 6, 12, 4, 1
key: 1 parent: 4 color: red
key: 4 parent: 6 color: black
key: 5 parent: 4 color: red
key: 6 parent: 9 color: red
key: 7 parent: 6 color: black
key: 9 parent:   color: black
key: 11 parent: 9 color: black
key: 12 parent: 11 color: red
1 2
2 3
Insert: 2, 3
key: 1 parent: 2 color: red
key: 2 parent: 4 color: black
key: 3 parent: 2 color: red
key: 4 parent: 6 color: red
key: 5 parent: 4 color: black
key: 6 parent:   color: black
key: 7 parent: 9 color: black
key: 9 parent: 6 color: red
key: 11 parent: 9 color: black
key: 12 parent: 11 color: red
```



```
2
1 8
5 11 9 7 6 12 4 1
Insert: 5, 11, 9, 7, 6, 12, 4, 1
key: 1 parent: 4 color: red
key: 4 parent: 6 color: black
key: 5 parent: 4 color: red
key: 6 parent: 9 color: red
key: 7 parent: 6 color: black
key: 9 parent:   color: black
key: 11 parent: 9 color: black
key: 12 parent: 11 color: red
2 2
11 5
Delete: 11, 5
key: 1 parent: 4 color: red
key: 4 parent: 6 color: black
key: 6 parent: 9 color: red
key: 7 parent: 6 color: black
key: 9 parent:   color: black
key: 12 parent: 9 color: black
```