111550113 謝詠晴

Algorithms Exercise #2

a. Environment

I. OS: Windows 11

II. Compiler: GNU GCC Compiler

III. IDE: Code::Blocks 20.03

b. Results

I. Overall Structure:

i. Node structure:

Each node in the tree is represented by a struct (*'Node'*), which includes data, pointers to parent, left child, and right child nodes, and a color indicator (0 for black, 1 for red).

```
struct Node{
    int data;
    Node* parent;
    Node* left;
    Node* right;
    int color; //0 black, 1 red
};
```

ii. Red Black Tree class (*'RBT'*):

Implementing RBT by class with private and public methods. And I initialize two nodes (*'root'*), (*'tnull'*) to represent root and NULL leaves in the tree.

```
class RBT{
private:
    Node* root;
    Node* tnull;
```

iii. Main function ('main'):

(1) Using while loop to operate each operation.

(2) Using '*vector<int> nodelist*' to store the nodes needed to be inserted or deleted.

```
    while(t!=0){
        int op, n; // operation, num of element
        cin>>op>>n;
        vector<int> nodelist(n);
        for(int i=0;i<n;i++){
            int node;
            cin>>node;
            nodelist[i]=node;
        }
        if(op==1){
            cout<<"Insert: ";
            for(int i=0;i<n;i++){
                if(i!=n-1){
                    cout<<nodelist[i]<<", ";
                }
                else{cout<<nodelist[i]<<"\n";}
            }
            for(int i=0;i<n;i++){
                rbt.INSERT(nodelist[i]);
            }
        }
```

iv. Initialization:

The constructor for the RBT class initializes the RBT class by create a null leaves node and set the root to this node.

```
public:
    RBT(){
        tnull=new Node;
        tnull->left=nullptr;
        tnull->right=nullptr;
        tnull->color=0; //black
        root=tnull;
    }
```

II. Insert:

i. INSERT:

(1) Allocates memory for a new node, which is the inserted node.

(2) Declare two node pointer *'y'* and *'x'*, y is the pointer to the parent node and x is the pointer starts at the root and will be used to traverse the tree to find the correct position.

(3) Using while loop to find the correct position for the inserted node.

(4) *'if (y == nullptr) { ... } else if (node->data < y->data) { ... } else { ... }'* is to insert the node to the correct position. And we set the color of this node to red.

(5) Check if the parent or grandparent is null, if so then return; if not then call the *'insert_fix'* function to fix the violation to RBT.

```
    void INSERT(int key){
        Node*node = new Node;
        node->data=key;
        Node* y=nullptr;
        Node* x=this->root;

        while(x!=tnull){
            y=x;
            if(node->data < x->data){
                x=x->left;
            }
            else{
                x=x->right;
            }
        }
        node->parent=y;
        if(y==nullptr){ //x is tnull, set node to root
            root=node;
        }
        else if(node->data < y->data){
            y->left=node;
        }
        else{y->right=node;}
        node->left=tnull;
        node->right=tnull;
        node->color=1; //red
        if (node->parent == nullptr) {
            node->color = 0;
            return;
        }

        if (node->parent->parent == nullptr) {
            return;
        }
        insert_fix(node);
    }
```

ii. Left rotate:

(1) Store the right child of current node as *‘y’*, and update the right child of current node with the left child of *'y'*.

(2) Update the parent pointer of *'y'* to point to the current node's parent and the parent's child pointer to point to 'y' accordingly.

(3) Set the left child of 'y' to be the current node and set the parent of the current node to be 'y'.

```
void left_rotate(Node*cur){
    Node* y =cur->right;
    cur->right=y->left;
    if(y->left!=tnull){
        y->left->parent=cur;
    }
    y->parent=cur->parent;
    if(cur->parent==nullptr){ //set y as root
        this->root=y;
    }
    else if(cur==cur->parent->left){
        cur->parent->left=y;
    }
    else{
        cur->parent->right=y;
    }
    y->left = cur;
    cur->parent = y;
}
```

iii. Right rotate:

Like the left rotate, only exchanging left to right.

iv. Insert fix:

(1) Maintain the RBT property after insertion.

(2) Ensure the root node is always black.

(3) There’re two circumstances when parent is the left or right child of grandparent. And under these circumstances, there’re three cases. We consider uncle *‘u’* as the other child of grandparent.

(4) Case 1: If the uncle node is red. We then recolor it and parent of the current node to black, and color the grandparent node red.

(5) Case 2: If the uncle node is black and the current node is right child. We then perform left rotation. Turning it to the Case 3.

(6) Case 3: If the uncle node is black and the current node is left child. We recolor parent to black and grandparent to red, and perform right rotation of the grandparent node.

```
void insert_fix(Node* cur){
    while(cur->parent->color==1){
        if(cur->parent==cur->parent->parent->left){ //parent is left
            Node* u=cur->parent->parent->right; //uncle
            if(u->color==1){ //case 1
                cur->parent->color=0;
                u->color=0;
                cur->parent->parent->color=1;
                cur=cur->parent->parent;
            }
            else {
                if(cur==cur->parent->right){ // case 2
                    cur=cur->parent;
                    left_rotate(cur);
                }
                cur->parent->color=0;
                cur->parent->parent->color=1;
                right_rotate(cur->parent->parent);
            }
        }
        else{ //parent is right
            Node* u=cur->parent->parent->left; //uncle
            if(u->color==1){ //case 1
                cur->parent->color=0;
                u->color=0;
                cur->parent->parent->color=1;
                cur=cur->parent->parent;
            }
            else {
                if(cur==cur->parent->left){ // case 2
                    cur=cur->parent;
                    right_rotate(cur);
                }
                cur->parent->color=0;
                cur->parent->parent->color=1;
                left_rotate(cur->parent->parent);
            }
        }
        if (cur==root) {
            break;
        }
    }
    root->color=0; // ensure root's color is black
```

III. Delete:

i. Transplant:

The function is used to replace one subtree with another, it facilitates the process of deleting node.

```
void transplant(Node*u, Node*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;
}
```

ii. Minimum:

It is used to find the minimum node in the tree by searching the left node iteratively and return it.

```
Node* minimum(Node* node){
    while(node->left!=tnull){
        node=node->left;
    }
    return node;
}
```

iii. Delete helper:

(1) The public function *'DELETE( )'* will call this private function.

(2) First, declare three node pointer x, y, z. *'x'* points to the deleted node's child, *'y'* points to the deleted node and 'z' points to the real deleted node that will be released memory. Using while loop to search for the node to be deleted.

(3) There're three cases. If the deleted node has zero or one child, it uses the transplant method to replace the node with its child.

(4) If the deleted node has two children, it finds the node y with the smallest key in the right subtree of z, and then replaces z with y.

(5) The original color of the deleted node *'z'* is stored in *'y_original_color'*, if the color is black then we'll need to fix the tree by calling *'delete_fix'* function.

```
void delete_helper(Node*cur, int key){
    Node* x; //x:delete node's child,
    Node* y; //y:real delete node
    Node* z=tnull; //delete node
    //search delete node
    while(cur!=tnull){
        if(key==cur->data){
            z=cur;
            break;
        }
        else if(key<cur->data){
            cur=cur->left;
        }
        else{
            cur=cur->right;
        }
    }
    y=z;
    int y_original_color = y->color;
    if(z->left==tnull){ // case:zero or one child
        x=z->right;
        transplant(z,z->right);
    }
    else if(z->right==tnull){ // case:one child
        x=z->left;
        transplant(z,z->left);
    }
    else{ // case:two children
        y=minimum(z->right);
        y_original_color=y->color;
        x=y->right;
        if(y->parent==z){ // y is directly delete node's child
            x->parent=y;
        }
        else{
            transplant(y,y->right); // y-right replace y
            y->right=z->right; // let original z's child be y's child
            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){
        delete_fix(x);
    }
}
```

iv. Delete fix:

(1) There're two circumstances when current node is the left or right child. And under these circumstances, there're four cases.
(2) We consider siblings *'s'* as the other child of parent.
(3) Case 1: If the siblings is red, then we recolor it to black and its parent to red, and then perform left rotation. It will turn to case 2,3 or 4.
(4) Case 2: If siblings is black and both its children are black, then we recolor it to red.
(5) Case 3: If siblings is black and the right child of it is black, then we recolor the other child to black and siblings to red. And perform right rotation. It will turn to case 4.
(6) Case 4: If siblings is black and the right child of it is red. Then recolor the siblings to its parent's color, color its parent to black, and color siblings' right child to black. Afterward, performing left rotation to parent and set cur to root.

```
void delete_fix(Node* cur){
    while(cur!=root && cur->color==0){ //cur is black and not root->draw it as red
        if(cur==cur->parent->left){ // cur is left
            Node* s=cur->parent->right; //sibilings
            if(s->color==1){ //case 1: sibilings is red
                s->color=0;
                cur->parent->color=1;
                left_rotate(cur->parent);
                s=cur->parent->right;
            }
            if(s->left->color==0 && s->right->color==0){ //case 2: both sibilings' children are black
                s->color=1;
                cur=cur->parent;
            }
            else{ //sibilings' children: 1 black 1 red
                if(s->right->color==0){ //case 3: right black left red
                    s->left->color=0;
                    s->color=1;
                    right_rotate(s);
                    s=cur->parent->right;
                }
                s->color=cur->parent->color; //case 4: right red left black
                cur->parent->color=0;
                s->right->color=0;
                left_rotate(cur->parent);
                cur=root;
            }
        }
        else{               // cur is right
            Node* s=cur->parent->left;
            if(s->color==1){ //case 1: sibilings is red
```

IV. Output:

i. Print( ):

(1) The public function ‘*print( )*’ will called private function ‘*print_helper*’.

(2) Using recursive and inorder traversal to print the tree.

(3) Because the parent of root needed to be printed space, so I use *to_string* to turn the integer data to string type.

```
void print_helper(Node* root){
    if(root!=tnull){
        print_helper(root->left);
        string par;
        if(root->parent==nullptr){
            par=" ";
        }else{
            Node* p=root->parent;
            par=to_string(p->data);
        }
        string col;
        if(root->color==0){
            col="black";
        }else{
            col="red";
        }
        cout<<"key: "<<root->data<<" parent: "<<par<<" color: "<<col<<"\n";
        print_helper(root->right);
    }
}
```