Practical 5

# Student Details

* Name : Tushar Harsora
* Roll : 19BCE509

# Aim

Implementing the Split operation in Balanced Binary Search tree. With 2 options 1) splitting at give search key, 2) splitting from root. The operation should be performed in logarithmic complexity to height of the tree.

# Code

* AVL.h (Header File)

#pragma once

#include <algorithm>

#include <stack>

#include <iostream>

#include <cstdint>

#include <cassert>

using namespace std;

template<typename Key, typename Value>

struct Node {

public:

    Node() = default;

    Node(Key key, Value value, int64\_t height) : key(key), value(value), height(height),

        left(nullptr), right(nullptr) { }

    Key key;

    Value value;

    int64\_t height;

    Node<Key, Value>\* left;

    Node<Key, Value>\* right;

    friend ostream& operator<<(ostream& out, const Node<Key, Value>\* const node) {

        if (node == nullptr)

            return out << "\n";

        // Or Instead You Can Prematurely Check if left or right child exists

        // if yes then call operator<< recursively

        out << node->left;

        out << node->key << " : " << node->value << " Height : " << node->height;

        out << node->right;

        return out;

    }

    friend void swap(Node<Key, Value>& lhs, Node<Key, Value>& rhs) {

        using std::swap;

        swap(lhs.key, rhs.key);

        swap(lhs.value, rhs.value);

    }

};

template<typename Key, typename Value>

struct AVL {

public:

    Node<Key, Value>\* root;

private:

    static int64\_t getHeight(Node<Key, Value>\* node) {

        return node == nullptr ? 0 : node->height;

    }

    static Node<Key, Value>\* \_right\_rotate(Node<Key, Value>\* x) {

        /\*

                         x                                   y

                        / \                                 / \

                       y   xr         ==>                  /   \

                      / \                                 z     x

                     z   yr                              / \   / \

                    / \                                 zl zr yr xr

                   zl  zr

        \*/

        auto y = x->left;

        auto yr = x->left->right;

        y->right = x;

        x->left = yr;

        x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));

        y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));

        return y;

    }

    static Node<Key, Value>\* \_left\_rotate(Node<Key, Value>\* x) {

        /\*

                         x                                    y

                        / \                                  / \

                      xl   y          ==>                   /   \

                          / \                              x     z

                         yl  z                            / \   / \

                            / \                          xl yl zl zr

                           zl  zr

        \*/

        auto y = x->right;

        auto yl = x->right->left;

        y->left = x;

        x->right = yl;

        x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));

        y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));

        return y;

    }

    Node<Key, Value>\* \_insert(Node<Key, Value>\* root, Node<Key, Value>\* new\_node) {

        if (root == nullptr)

            return new\_node;

        if (new\_node->key < root->key)                      // If Key Compares Less than then Left Subtree

            root->left = \_insert(root->left, new\_node);

        else

            root->right = \_insert(root->right, new\_node);   // If Key Compares greater equal then Right Subtree

        root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));

        auto balance = getHeight(root->left) - getHeight(root->right);

        /\*

        \* There Are 4 cases That are possible

        \* 1) Height of left Subtree is Greater

        \*       a) key of grand child is less than key of child             (Right Rotate)

        \*       b) key of grand child is greater equal to key of child      (Left Right Rotate)

        \* 2) Height of right Subtree is Greater

        \*       a) key of grand child is less than key of child             (Right Left Rotate)

        \*       b) key of grand child is greater equal to key of child      (Left Rotate)

        \*/

        if (balance > 1) {                              // Height Of left subtree is greater

            if (new\_node->key < root->left->key) {      // Right Rotate

                return \_right\_rotate(root);

            }

            else {                                      // Left Right Rotate

                root->left = \_left\_rotate(root->left);

                return \_right\_rotate(root);

            }

        }

        else if (balance < -1) {                        // Height Of right subtree is greater

            if (new\_node->key < root->right->key) {     // Right Left Rotate

                root->right = \_right\_rotate(root->right);

                return \_left\_rotate(root);

            }

            else {                                      // Left Rotate

                return \_left\_rotate(root);

            }

        }

        // There Was No Balancing Issues return this node

        return root;

    }

    Node<Key, Value>\* \_delete(Node<Key, Value>\* root, const Key& key) {

        if (root == nullptr)

            return root;

        if (key < root->key) {

            root->left = \_delete(root->left, key);

        }

        else if (key > root->key) {

            root->right = \_delete(root->right, key);

        }

        else {      // Delete This

            auto exactly\_one\_child = (root->left == nullptr || root->right == nullptr) &&

                !(root->left == nullptr && root->right == nullptr); // A exor B

            if (exactly\_one\_child) {

                auto child\_node = root->left ? root->left : root->right;

                delete root;

                return child\_node;

            }

            else if (root->left == nullptr && root->right == nullptr) {

                delete root;

                return nullptr;

            }

            else {

                Node<Key, Value>\* inorder\_successor = root->right;

                Node<Key, Value>\* inorder\_successor\_parent = root;

                while (inorder\_successor->left) {

                    inorder\_successor\_parent = inorder\_successor;

                    inorder\_successor = inorder\_successor->left;

                }

                swap(\*inorder\_successor, \*root);

                if (inorder\_successor\_parent->right == inorder\_successor)

                    inorder\_successor\_parent->right = nullptr;

                else

                    inorder\_successor\_parent->left = nullptr;

                delete inorder\_successor;

            }

        }

        auto leftHeight = getHeight(root->left);

        auto rightHeight = getHeight(root->right);

        root->height = max(leftHeight, rightHeight) + 1;

        auto balance = get\_balance(root);

        if (balance > 1) {

            auto left\_balance = get\_balance(root->left);

            if (left\_balance >= 0) {

                return \_right\_rotate(root);

            }

            else {

                root->left = \_left\_rotate(root->left);

                return \_right\_rotate(root);

            }

        }

        else if (balance < -1) {

            auto right\_balance = get\_balance(root->right);

            if (right\_balance <= 0) {

                return \_left\_rotate(root);

            }

            else {

                root->right = \_right\_rotate(root->right);

                return \_left\_rotate(root);

            }

        }

        return root;

    }

    static auto get\_balance(Node<Key, Value>\* root) {

        return getHeight(root->left) - getHeight(root->right);

    }

public:

    AVL() : root(nullptr) { }

    AVL(Node<Key, Value>\* root) : root(root) { }

    void insert(const Key& key, const Value& value) {

        Node<Key, Value>\* node = new Node<Key, Value>(key, value, 1);

        root = \_insert(root, node);

    }

    void erase(const Key& key) {

        root = \_delete(root, key);

    }

    static Node<Key, Value>\* join(AVL<Key, Value> lhs, AVL<Key, Value> rhs) {

        if (lhs.root == nullptr)

            return rhs.root;

        else if (rhs.root == nullptr)

            return lhs.root;

        // Make Sure Tree in lhs has every key less than smallest key in rhs

        auto minlhs = lhs.root;

        auto maxlhs = lhs.root;

        auto minrhs = rhs.root;

        auto maxrhs = rhs.root;

        while (minlhs->left)    minlhs = minlhs->left;

        while (maxlhs->right)   maxlhs = maxlhs->right;

        while (minrhs->left)    minrhs = minrhs->left;

        while (maxrhs->right)   maxrhs = maxrhs->right;

#ifdef \_DEBUG

        assert(maxlhs->key < minrhs->key);

        if (lhs.root == rhs.root)

            assert(false);

#endif // \_DEBUG

        auto rebalance = [](Node<Key, Value>\* root) -> Node<Key, Value>\*{

            auto leftHeight = getHeight(root->left);

            auto rightHeight = getHeight(root->right);

            root->height = max(leftHeight, rightHeight) + 1;

            auto balance = get\_balance(root);

            if (balance > 1) {

                auto left\_balance = get\_balance(root->left);

                if (left\_balance >= 0) {

                    return \_right\_rotate(root);

                }

                else {

                    root->left = \_left\_rotate(root->left);

                    return \_right\_rotate(root);

                }

            }

            else if (balance < -1) {

                auto right\_balance = get\_balance(root->right);

                if (right\_balance <= 0) {

                    return \_left\_rotate(root);

                }

                else {

                    root->right = \_right\_rotate(root->right);

                    return \_left\_rotate(root);

                }

            }

            else {

                return root;

            }

        };

        auto lhs\_height = lhs.root->height, rhs\_height = rhs.root->height;

        if (abs(lhs\_height - rhs\_height) <= 1) {        // Case 1

            Node<Key, Value>\* new\_root = new Node<Key, Value>(minrhs->key, minrhs->value, 1);   // TODO: set valid height

            rhs.erase(minrhs->key);

            new\_root->left = lhs.root;

            new\_root->right = rhs.root;

            new\_root = rebalance(new\_root);

            new\_root->height = 1 + max(getHeight(new\_root->left), getHeight(new\_root->right));

            return new\_root;

        }

        else if (lhs\_height <= rhs\_height - 2) {        // Case 2 And 3 Because we are sure height of left tree is smaller

            Node<Key, Value>\* new\_root = new Node<Key, Value>(minrhs->key, minrhs->value, 1);   // TODO: set valid height

            rhs.erase(minrhs->key);

            lhs\_height = getHeight(lhs.root), rhs\_height = getHeight(rhs.root);

            Node<Key, Value>\* ptr = rhs.root;

            stack<Node<Key, Value>\*> stck;

            while (getHeight(ptr) > lhs\_height) {

                stck.push(ptr);

                ptr = ptr->left;

            }

            //TODO: Stack will not always have top element

            if (getHeight(ptr) == lhs\_height && stck.top()->height == lhs\_height + 2) {     // Case 2.1 OR 3.1

                new\_root->height = lhs\_height + 1;

                stck.top()->left = new\_root;

                new\_root->left = lhs.root;

                new\_root->right = ptr;

            }

            else if (getHeight(ptr) == lhs\_height && stck.top()->height == lhs\_height + 1) {

                new\_root->height = lhs\_height + 1;

                stck.top()->left = new\_root;

                new\_root->left = lhs.root;

                new\_root->right = ptr;

                while (!stck.empty()) {

                    stck.top() = rebalance(stck.top());

                    Node<Key, Value>\* curr = stck.top();

                    stck.pop();

                    if (stck.size() == 0)

                        rhs.root = curr;

                    else

                        stck.top()->left = curr;

                }

            }

            else if (getHeight(ptr) == lhs\_height - 1 && stck.top()->height == lhs\_height + 1) {

                new\_root->height = lhs\_height + 1;

                stck.top()->left = new\_root;

                new\_root->left = lhs.root;

                new\_root->right = ptr;

                while (!stck.empty()) {

                    stck.top() = rebalance(stck.top());

                    Node<Key, Value>\* curr = stck.top();

                    stck.pop();

                    if (stck.size() == 0)

                        rhs.root = curr;

                    else

                        stck.top()->left = curr;

                }

            }

            else {

                assert(false);

            }

            return rhs.root;

        }

        else if (rhs\_height <= lhs\_height - 2) {

            Node<Key, Value>\* new\_root = new Node<Key, Value>(minrhs->key, minrhs->value, 1);

            rhs.erase(minrhs->key);

            lhs\_height = getHeight(lhs.root), rhs\_height = getHeight(rhs.root);

            Node<Key, Value>\* ptr = lhs.root;

            stack<Node<Key, Value>\*> stck;

            while (getHeight(ptr) > rhs\_height) {

                stck.push(ptr);

                ptr = ptr->right;

            }

            if (getHeight(ptr) == rhs\_height && stck.top()->height == rhs\_height + 2) {

                new\_root->height = rhs\_height + 1;

                stck.top()->right = new\_root;

                new\_root->left = ptr;

                new\_root->right = rhs.root;

            }

            else if (getHeight(ptr) == rhs\_height && stck.top()->height == rhs\_height + 1) {

                new\_root->height = rhs\_height + 1;

                stck.top()->right = new\_root;

                new\_root->left = ptr;

                new\_root->right = rhs.root;

                // Root change thai che!!

                while (!stck.empty()) {

                    stck.top() = rebalance(stck.top());

                    Node<Key, Value>\* curr = stck.top();

                    stck.pop();

                    if (stck.size() == 0)

                        lhs.root = curr;

                    else

                        stck.top()->right = curr;

                }

            }

            else if (getHeight(ptr) == rhs\_height - 1 && stck.top()->height == rhs\_height + 1) {

                new\_root->height = rhs\_height + 1;

                stck.top()->right = new\_root;

                new\_root->left = ptr;

                new\_root->right = rhs.root;

                while (!stck.empty()) {

                    stck.top() = rebalance(stck.top());

                    Node<Key, Value>\* curr = stck.top();

                    stck.pop();

                    if (stck.size() == 0)

                        lhs.root = curr;

                    else

                        stck.top()->right = curr;

                }

            }

            else {

                assert(false);

            }

            return lhs.root;

        }

        else {

            assert(false);

        }

        return nullptr;

    }

    static void \_split(Node<Key, Value>\* root, const Key& splitKey, deque<Node<Key, Value>\*>& left\_tree\_list, deque<Node<Key, Value>\*>& right\_tree\_list) {

        if (!root)

            return;

        if (root->key == splitKey) {

            right\_tree\_list.push\_front(root->right);

            left\_tree\_list.push\_back(root->left);

            root->left = root->right = nullptr;

            root->height = 1;

            right\_tree\_list.push\_front(root);

            return;

        }

        else if (root->key < splitKey) {        // Goto Right Subtree

            Node<Key, Value>\* right\_child = root->right;

            left\_tree\_list.push\_back(root->left);

            root->left = root->right = nullptr;

            root->height = 1;

            left\_tree\_list.push\_back(root);

            \_split(right\_child, splitKey, left\_tree\_list, right\_tree\_list);

            return;

        }

        else {                                  // Goto Left Subtree

            Node<Key, Value>\* left\_child = root->left;

            right\_tree\_list.push\_front(root->right);

            root->left = root->right = nullptr;

            root->height = 1;

            right\_tree\_list.push\_front(root);

            \_split(left\_child, splitKey, left\_tree\_list, right\_tree\_list);

            return;

        }

    }

    static pair<AVL<Key, Value>, AVL<Key, Value>> split(AVL<Key, Value> lhs, const Key& splitKey) {

        deque<Node<Key, Value>\*> left\_tree\_list, right\_tree\_list;

        \_split(lhs.root, splitKey, left\_tree\_list, right\_tree\_list);

        // Removing All Null Node Pointers

        auto it1 = remove\_if(left\_tree\_list.begin(), left\_tree\_list.end(), [](auto ele) { return ele == nullptr; });

        if(it1 != left\_tree\_list.end())

            left\_tree\_list.erase(it1);

        auto it2 = remove\_if(right\_tree\_list.begin(), right\_tree\_list.end(), [](auto ele) { return ele == nullptr; });

        if(it2 != right\_tree\_list.end())

            right\_tree\_list.erase(it2);

        AVL<Key, Value> left\_tree = nullptr;

        for (auto tree : left\_tree\_list) {

            AVL<Key, Value> tmp(tree);

            left\_tree.root = join(left\_tree, tmp);

        }

        AVL<Key, Value> right\_tree = nullptr;

        for (auto tree : right\_tree\_list) {

            AVL<Key, Value> tmp(tree);

            right\_tree.root = join(right\_tree, tmp);

        }

        return make\_pair(left\_tree, right\_tree);

    }

    friend ostream& operator<<(ostream& out, const AVL<Key, Value>& avl) {

        out << avl.root;

        return out;

    }

};

* AVL Test (main function)

#include <iostream>

#include "avl.h"

int main() {

auto prepare\_data = []() {

AVL<int, int> avl1;

for (int i = 1; i <= 5; i++) {

avl1.insert(i, i \* 10);

}

for (int i = 11; i <= 15; i++) {

AVL<int, int> temp;

temp.insert(i, i \* 10);

avl1.root = AVL<int, int>::join(avl1, temp);

}

return avl1;

};

{

// Split At Given Key left tree will be All keys Less Than 10, right tree will be keys Greater Than Equal To 10

cout << "Splitting On Key 10" << endl;

auto splitted = AVL<int, int>::split(prepare\_data(), 10);

cout << splitted.first << endl << splitted.second;

}

{

// Split From Root

auto avl\_tree = prepare\_data();

cout << "\n\nSplitting From Root\nThe Key At Root is : " << avl\_tree.root->key;

auto splitted = AVL<int, int>::split(avl\_tree, avl\_tree.root->key);

cout << splitted.first << endl << splitted.second;

}

return 0;

}

# Inputs

This Program Doesn’t take any input the output is attached below

# Screenshots of output

# Splitting On Key 10

# 1 : 10 Height : 1

# 2 : 20 Height : 3

# 3 : 30 Height : 1

# 4 : 40 Height : 2

# 5 : 50 Height : 1

# 11 : 110 Height : 1

# 12 : 120 Height : 2

# 13 : 130 Height : 1

# 14 : 140 Height : 3

# 15 : 150 Height : 1

# Splitting From Root

# The Key At Root is : 4

# 1 : 10 Height : 1

# 2 : 20 Height : 2

# 3 : 30 Height : 1

# 4 : 40 Height : 1

# 5 : 50 Height : 2

# 11 : 110 Height : 3

# 12 : 120 Height : 1

# 13 : 130 Height : 4

# 14 : 140 Height : 2

# 15 : 150 Height : 1

# Conclusion

Here we learned about complicated split and join operations in the balanced trees. The program used join operations in balanced bst and using 2 list merge them and keep track of new roots.