

EUROPEAN UNIVERSITY

OF LEFKE

***Name :*** *Honest**Chirozva*

***Department :*** *Software Engineering*

***Student Number :*** *154 425*

***Course :*** *Advanced Data Structures (COMP466)*

***Date :*** 12/04/19

*Binary Search Trees – Balancing Trees*

*2018-2019 Spring Semester*

***Tree.hpp***

*#ifndef BINARY\_TREE\_H*

*#define BINARY\_TREE\_H*

*#include <algorithm>*

*#include <cstdlib>*

*#include <ctime>*

*#include <iostream>*

*#include <stack>*

*#include <vector>*

*struct node {*

*int info;*

*struct node \*left, \*right;*

*};*

*typedef struct node Node;*

*class BinaryTree {*

*// private data members*

*Node \*root\_;*

*std::vector<int> data\_;*

*// private methods*

*Node \*makeTree(const int);*

*void setLeft(Node \*, const int);*

*void setRight(Node \*, const int);*

*Node \*insert(Node \*, const int); // recursive method*

*void inorder(Node \*);*

*Node \*free(Node \*);*

*Node \*getItemNode(const int);*

*Node \*getPreviousNode(const int);*

*int depth(Node \*);*

*void sortVectorItems();*

*bool removeByMerging(Node \*);*

*Node \*getMinNode(Node \*);*

*Node \*removeRecursive(Node \*, int);*

*public:*

*BinaryTree();*

*void insertIterative(const int);*

*void insertRecursive(const int);*

*void traverseRecursive();*

*void traverseIterative();*

*void insertRandomIterative(const int);*

*void insertRandom(const int);*

*// default M - delete by Merging, R for recursive delete*

*bool deleteItem(const int, const char = 'M');*

*bool deleteRandom(const int, const char = 'M');*

*void vectorInitialisation(std::vector<int>, int, int);*

*void balanceTree();*

*int height();*

*~BinaryTree();*

*};*

*// implementation*

*// PRIVATE METHODS*

*Node \*BinaryTree::makeTree(const int x) {*

*Node \*temp = new Node;*

*temp->info = x;*

*temp->left = temp->right = nullptr;*

*return temp;*

*};*

*void BinaryTree::setLeft(Node \*p, const int x) {*

*if (p == nullptr)*

*std::cout << ".....EMPTY NODE.....\n";*

*else if (p->left != nullptr)*

*std::cout << ".....NON-EMPTY LEFT NODE.....\n";*

*else*

*p->left = makeTree(x);*

*}*

*void BinaryTree::setRight(Node \*p, const int x) {*

*if (p == nullptr)*

*std::cout << ".....EMPTY NODE.....\n";*

*else if (p->right != nullptr)*

*std::cout << ".....NON-EMPTY RIGHT NODE.....\n";*

*else*

*p->right = makeTree(x);*

*}*

*Node \*BinaryTree::insert(Node \*node, const int x) {*

*if (node == nullptr)*

*return makeTree(x);*

*else if (x < node->info)*

*node->left = insert(node->left, x);*

*else if (x > node->info)*

*node->right = insert(node->right, x);*

*else {*

*std::cout << x << " : is a duplicate therefore deprecated!\n";*

*data\_.pop\_back(); // pop item from vector if it's duplicate*

*}*

*return node;*

*}*

*void BinaryTree::inorder(Node \*p) {*

*if (p == nullptr) return;*

*inorder(p->left);*

*std::cout << p->info << " ";*

*inorder(p->right);*

*}*

*Node \*BinaryTree::free(Node \*p) {*

*if (p == nullptr) return nullptr;*

*{*

*free(p->left);*

*free(p->right);*

*delete p;*

*}*

*return nullptr;*

*}*

*Node \*BinaryTree::getItemNode(const int x) {*

*Node \*p = root\_;*

*while (p != nullptr && p->info != x) {*

*p = (p->info < x) ? p->right : p->left;*

*}*

*return p;*

*}*

*Node \*BinaryTree::getPreviousNode(const int x) {*

*Node \*p = root\_, \*prev = nullptr;*

*while (p != nullptr && p->info != x) {*

*prev = p;*

*p = (p->info < x) ? p->right : p->left;*

*}*

*return prev;*

*}*

*int BinaryTree::depth(Node \*p) {*

*if (p == nullptr) return 0;*

*int left\_depth = depth(p->left);*

*int right\_depth = depth(p->right);*

*return (left\_depth > right\_depth) ? left\_depth + 1 : right\_depth + 1;*

*}*

*void BinaryTree::sortVectorItems() { std::sort(data\_.begin(), data\_.end()); }*

*bool BinaryTree::removeByMerging(Node \*p) {*

*Node \*node, \*t, \*prev;*

*prev = getPreviousNode(p->info);*

*node = p;*

*if (p != nullptr) {*

*if (node->right == nullptr) {*

*t = node;*

*node = node->left;*

*} else if (p->left == nullptr) {*

*t = node;*

*node = node->right;*

*} else {*

*t = node->left;*

*while (t->right != nullptr) t = t->right;*

*t->right = node->right;*

*t = node;*

*node = node->left;*

*}*

*// Merging*

*if (p == root\_)*

*root\_ = node;*

*else if (prev->left == p)*

*prev->left = node;*

*else*

*prev->right = node;*

*delete t;*

*return true;*

*}*

*return false;*

*}*

*Node \*BinaryTree::getMinNode(Node \*p) {*

*Node \*node = p;*

*if (root\_ == nullptr) return nullptr;*

*while (node->left != nullptr) node = node->left;*

*return node;*

*}*

*Node \*BinaryTree::removeRecursive(Node \*p, const int x) {*

*if (p == nullptr) return p;*

*if (x < p->info)*

*p->left = removeRecursive(p->left, x);*

*else if (x > p->info)*

*p->right = removeRecursive(p->right, x);*

*else {*

*if (p->left == nullptr) {*

*Node \*t = p->right;*

*free(p);*

*return t;*

*} else if (p->right == nullptr) {*

*Node \*t = p->left;*

*delete p;*

*return t;*

*}*

*Node \*t = getMinNode(p->right);*

*p->info = t->info;*

*p->right = removeRecursive(p->right, t->info);*

*}*

*return p;*

*}*

*// PUBLIC METHODS*

*BinaryTree::BinaryTree() {*

*root\_ = nullptr;*

*srand(time(NULL));*

*};*

*void BinaryTree::insertIterative(const int x) {*

*Node \*p, \*q;*

*// store the items in vector for sorting and deletion purposes*

*data\_.push\_back(x);*

*if (root\_ == nullptr)*

*root\_ = makeTree(x);*

*else {*

*p = q = root\_;*

*while (x != p->info && q != nullptr) {*

*p = q;*

*if (x < p->info)*

*q = p->left;*

*else*

*q = p->right;*

*}*

*if (x == p->info) {*

*std::cout << x << " : is a duplicate therefore deprecated!\n";*

*return;*

*} else if (x < p->info)*

*setLeft(p, x);*

*else*

*setRight(p, x);*

*}*

*}*

*void BinaryTree::insertRecursive(const int x) {*

*// store the items in vector for sorting and deletion purposes*

*data\_.push\_back(x);*

*root\_ = insert(root\_, x);*

*}*

*void BinaryTree::traverseRecursive() {*

*inorder(root\_);*

*std::cout << std::endl;*

*}*

*void BinaryTree::traverseIterative() {*

*Node \*p = root\_;*

*std::stack<Node \*> stack;*

*while (p != nullptr) {*

*while (p != nullptr) {*

*if (p->right != nullptr) stack.push(p->right);*

*stack.push(p);*

*p = p->left;*

*}*

*p = stack.top();*

*stack.pop();*

*while (!stack.empty() && p->right == nullptr) {*

*// std::cout << p->info << " ";*

*p = stack.top();*

*stack.pop();*

*}*

*// std::cout << p->info << " ";*

*if (!stack.empty()) {*

*p = stack.top();*

*stack.pop();*

*} else*

*p = nullptr;*

*}*

*}*

*void BinaryTree::insertRandomIterative(const int size) {*

*for (int i = 0; i < size; ++i) insertIterative(rand());*

*}*

*void BinaryTree::insertRandom(const int size) {*

*for (int i = 0; i < size; ++i) insertRecursive(rand());*

*}*

*bool BinaryTree::deleteItem(const int x, const char deleteType) {*

*Node \*node = getItemNode(x);*

*// std::cout << "Attempting delete: " << x << std::endl;*

*if (root\_ == nullptr)*

*std::cout << "\*\*\*EMPTY TREE\*\*\*" << std::endl;*

*else if (node != nullptr && node->info == x) {*

*if (deleteType == 'R' || deleteType == 'r')*

*removeRecursive(root\_, x);*

*else*

*removeByMerging(node); // default delete type is by merging*

*return true;*

*} else if (node == nullptr)*

*std::cout << "Element: " << x << " is not in the tree!!" << std::endl;*

*}*

*bool BinaryTree::deleteRandom(const int number\_of\_items,*

*const char deleteType) {*

*int index;*

*if (number\_of\_items > data\_.size()) return false;*

*for (int i = 0; i < number\_of\_items; ++i) {*

*index = rand() % data\_.size();*

*// std::cout << "Index: " << index << std::endl;*

*// std::cout << "Size: " << data\_.size() << std::endl;*

*deleteItem(data\_[index], deleteType);*

*data\_.erase(data\_.begin() + index);*

*}*

*return true;*

*}*

*void BinaryTree::vectorInitialisation(std::vector<int> data, int start,*

*int end) {*

*int middle;*

*if (start <= end) {*

*middle = (start + end) / 2;*

*root\_ = insert(root\_, data[middle]);*

*vectorInitialisation(data, start, middle - 1);*

*vectorInitialisation(data, middle + 1, end);*

*}*

*}*

*void BinaryTree::balanceTree() {*

*sortVectorItems();*

*root\_ = free(root\_);*

*vectorInitialisation(data\_, 0, data\_.size() - 1);*

*}*

*int BinaryTree::height() { return depth(root\_); }*

*BinaryTree::~BinaryTree() { root\_ = free(root\_); };*

*#endif*

***Main.cpp***

*#include <cstdlib>*

*#include <ctime>*

*#include <iomanip>*

*#include <iostream>*

*#include "Tree.hpp"*

*int main(void) {*

*BinaryTree bst;*

*for (int i = 0; i < 100; ++i) {*

*bst.insertRandom(20);*

*bst.deleteRandom(4);*

*}*

*std::cout << " TREE 1 " << std::endl;*

*std::cout << "Unbalanced Tree height: " << bst.height() << std::endl;*

*bst.balanceTree();*

*std::cout << "Balanced Tree height: " << bst.height() << std::endl;*

*BinaryTree bst2;*

*for (int i = 0; i < 100; ++i) {*

*bst2.insertRandom(45);*

*bst2.deleteRandom(7);*

*}*

*std::cout << " TREE 2 " << std::endl;*

*std::cout << "Unbalanced Tree height: " << bst2.height() << std::endl;*

*bst2.balanceTree();*

*std::cout << "Balanced Tree height: " << bst2.height() << std::endl;*

*return 0;*

*}*

***OUTPUT***

