

EUROPEAN UNIVERSITY

OF LEFKE

***Name :*** *Honest*

***Surname :*** *Chirozva*

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

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

***Course :*** *Advanced Data Structures*

***Date :*** 02/03/19

*Binary Search Trees*

*2018-2019 Spring Semester*

***Tree.hpp***

*#ifndef BINARY\_TREE\_H*

*#define BINARY\_TREE\_H*

*#include <iostream>*

*#include <stack>*

*#include <cstdlib>*

*#include <ctime>*

*struct node*

*{*

*int info;*

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

*};*

*typedef struct node Node;*

*class BinaryTree*

*{*

*Node \*root\_;*

*//private methods*

*Node \*makeTree(int);*

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

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

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

*void inorder(Node \*);*

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

*public:*

*BinaryTree();*

*void insertIterative(int);*

*void insertRecursive(int);*

*void traverseRecursive();*

*void traverseIterative();*

*void insertRandomIterative(int);*

*void insertRandom(int);*

*~BinaryTree();*

*};*

*//implementation*

*BinaryTree::BinaryTree()*

*{*

*root\_ = nullptr;*

*srand(time(NULL));*

*};*

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

*{*

*Node \*temp = new Node;*

*temp->info = x;*

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

*return temp;*

*};*

*void BinaryTree::setLeft(Node \*p, 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, 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, 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";*

*return node;*

*}*

*void BinaryTree::insertRecursive(int x) { root\_ = insert(root\_, x); }*

*void BinaryTree::insertIterative(int x)*

*{*

*Node \*p, \*q;*

*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::inorder(Node \*p)*

*{*

*if (p == nullptr)*

*return;*

*inorder(p->left);*

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

*inorder(p->right);*

*}*

*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(int size)*

*{*

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

*insertIterative(rand()%3);*

*}*

*void BinaryTree::insertRandom(int size)*

*{*

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

*insertRecursive(rand());*

*}*

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

*{*

*if (p == nullptr)*

*return nullptr;*

*{*

*free(p->left);*

*free(p->right);*

*delete p;*

*}*

*return nullptr;*

*}*

*BinaryTree::~BinaryTree()*

*{*

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

*};*

*#endif*

***Main.cpp***

*#include "Tree.hpp"*

*#include <iostream>*

*#include <iomanip>*

*#include <cstdlib>*

*#include <ctime>*

*int main(void)*

*{*

*clock\_t start, end;*

*BinaryTree tree\_100, tree\_1000, tree\_10000;*

*tree\_100.insertRandom(100);*

*tree\_1000.insertRandom(1000);*

*tree\_10000.insertRandom(10000);*

*std::cout << std::setprecision(9) << std::fixed;*

*start = clock();*

*tree\_100.traverseRecursive();*

*end = clock();*

*std::cout << "Recursive Traversal - 100 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*start = clock();*

*tree\_100.traverseIterative();*

*end = clock();*

*std::cout << "Iterative Traversal - 100 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*start = clock();*

*tree\_1000.traverseRecursive();*

*end = clock();*

*std::cout << "Recursive Traversal - 1000 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*start = clock();*

*tree\_1000.traverseIterative();*

*end = clock();*

*std::cout << "Iterative Traversal - 1000 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*start = clock();*

*tree\_10000.traverseRecursive();*

*end = clock();*

*std::cout << "Recursive Traversal - 10000 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*start = clock();*

*tree\_10000.traverseIterative();*

*end = clock();*

*std::cout << "Iterative Traversal - 10000 : " << static\_cast<double>(end - start) / CLOCKS\_PER\_SEC << std::endl;*

*return 0;*

*}*

***OUTPUT***

*Recursive Traversal - 100 : 0.000023000*

*Iterative Traversal - 100 : 0.000015000*

*Recursive Traversal - 1000 : 0.000021000*

*Iterative Traversal - 1000 : 0.000101000*

*Recursive Traversal - 10000 : 0.000246000*

*Iterative Traversal - 10000 : 0.001186000*

|  |  |  |  |
| --- | --- | --- | --- |
| Method | No. of Elements | | |
|  | 100 | 1000 | 10000 |
| Recursive | 0.000023 | 0.000021 | 0.000246 |
| Iterative | 0.000015 | 0.000101 | 0.001186 |

*The two methods were compared using time complexity analysis. In conclusion, recursive method performs better than the iterative way during traversals.*