**Лесечко Олеся(ДА-02)**

Лабораторна робота №4

“Дослідження структури даних бінарне дерево пошуку”

**Мета роботи:** Ознайомитись і дослідити структури даних бінарне дерево пошуку та префіксне дерево, розглянути механізми балансування дерева. Набути навичок реалізації бінарного дерева пошуку мовою програмування С++, порівняти

власну реалізацію з готовим бібліотечним рішенням STL.**Варіант: Задача 1**

Структура Гравець має наступні поля: нікнейм, ранг, кількість досвіду, розмір донату тощо. Створити відсортовану “базу даних” гравців, в якій можна швидко перевіряти наявність потрібного гравця та знаходити всіх гравців на вказаному проміжку.

Хід виконання роботи:

A) УМОВА

1. Створити структуру для зберігання об’єктів з характеристиками відповідно до обраної задачі, обрати критерій для порівняння двох об’єктів та перевантажити необхідні оператори.

2. Реалізувати бінарне дерево пошуку

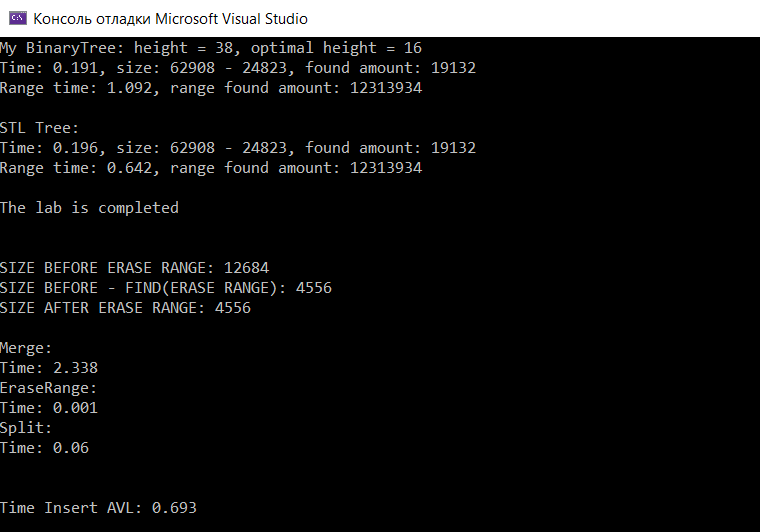
3. Провести тестування, використавши вказану нижче функцію testBinarySearchTree(). Перевірити правильність та швидкість роботи, порівнявши з готовим бібліотечним рішенням STL set.

4. Реалізувати методи бінарного дерева пошуку для роботи з піддеревами

Реалізувати у BinarySearchTree логіку балансування по типу АВЛ-дерева або іншого збалансованого дерева (червоно-чорне дерево, splay дерево)

Додаткові завдання: Реалізувати префіксне дерево для роботи програми автодоповнення слів.

B) СКРІНШОТ РЕЗУЛЬТАТУ





С) ЛІСТИНГ ПРОГРАМИ (КОД)

ASD.cpp

#include<iostream>

#include "testBST.h"

using namespace std;

int main() {

testBinarySearchTree();

additionalTask();

testAVL();

testPrefixtree();

return 0;

}

myBST.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include "myBST.h"

using namespace std;

bool BinarySearchTree::insert(const Data& value) {

Node\* added = findNode(value, root);

if (added != NULL) {

return false;

}

root = addNode(value, root, NULL);

counter++;

return true;

}

Node\* BinarySearchTree::addNode(const Data& value,Node\* node, Node\* p) {

if (node == NULL) {

node = new Node;

node->data = value;

node->LeftChild = NULL;

node->RightChild = NULL;

node->parent = p;

}

if (value < node->data){

node->LeftChild = addNode(value, node->LeftChild, node);

}

else if (value > node->data){

node->RightChild = addNode(value,node->RightChild, node);

}

return node;

}

int BinarySearchTree::size() {

return counter;

}

int BinarySearchTree::height() {

return heightTree(root);

}

int BinarySearchTree::heightTree(Node\* node) {

if (node == NULL) {

return 0;

}

else {

return 1 + max(heightTree(node->LeftChild), heightTree(node->RightChild));

}

}

bool BinarySearchTree::find(const Data& value) {

return (findNode(value, root) != NULL) ? 1 : 0;

}

Node\* BinarySearchTree::findNode(const Data& value, Node\* node) {

if (node == NULL) {

return NULL;

}

else {

if (value > node->data) {

return findNode(value, node->RightChild);

}

else if(value < node->data){

return findNode(value, node->LeftChild);

}

else {

return node;

}

}

return NULL;

}

bool BinarySearchTree::erase(const Data& value) {

Node\* deleted = findNode(value, root);

if (deleted == NULL) {

return false;

}

counter--;

Node\* p = deleted->parent;

if (deleted->LeftChild == NULL && deleted->RightChild == NULL) { // немає дітей

if (p == NULL) {

root = NULL;

}

else if (p->LeftChild != NULL && p->LeftChild->data == value) {

p->LeftChild = NULL;

}

else if (p->RightChild != NULL && p->RightChild->data == value) {

p->RightChild = NULL;

}

}

else if (deleted->LeftChild == NULL || deleted->RightChild == NULL) { // одне диття

if (p == NULL) {

if (deleted->LeftChild != NULL) {

root = deleted->LeftChild;

root->parent = NULL;

}

else {

root = deleted->RightChild;

root->parent = NULL;

}

}

else{

if (p->LeftChild != NULL && p->LeftChild->data == value) {

if (deleted->LeftChild != NULL) {

p->LeftChild = deleted->LeftChild;

if (deleted->LeftChild != NULL) {

deleted->LeftChild->parent = p;

}

}

else {

p->LeftChild = deleted->RightChild;

if (deleted->RightChild != NULL) {

deleted->RightChild->parent = p;

}

}

}

else {

if (deleted->LeftChild != NULL) {

p->RightChild = deleted->LeftChild;

if (deleted->LeftChild != NULL) {

deleted->LeftChild->parent = p;

}

}

else {

p->RightChild = deleted->RightChild;

if (deleted->RightChild != NULL) {

deleted->RightChild->parent = p;

}

}

}

}

}

else { // два дитя

Node\* replaceable = minimum(deleted->RightChild);

deleted->data = replaceable->data;

if (replaceable->parent->LeftChild == replaceable) {

replaceable->parent->LeftChild = replaceable->RightChild;

if (replaceable->RightChild != NULL) {

replaceable->RightChild->parent = replaceable->parent;

}

}

else {

replaceable->parent->RightChild = replaceable->RightChild;

if (replaceable->RightChild != NULL) {

replaceable->RightChild->parent = replaceable->parent;

}

}

replaceable->data = { 0,0,0,0 };

delete replaceable;

}

return true;

}

Node\* BinarySearchTree::minimum(Node\* node) {

if (node->LeftChild == NULL) {

return node;

}

else {

return minimum(node->LeftChild);

}

}

Node\* BinarySearchTree::maximum(Node\* node) {

if (node->RightChild == NULL) {

return node;

}

else {

return maximum(node->RightChild);

}

}

void BinarySearchTree::print() {

printTree(root);

}

void BinarySearchTree::printTree(Node\* node) {

if (node != NULL) {

printTree(node->LeftChild);

cout << node->data.nickname << " " << node->data.rank << " " << node->data.amountOfExperience << " " << node->data.sizeDonut << endl;

printTree(node->RightChild);

}

}

int BinarySearchTree::findInRange(const Data& value1, const Data& value2) {

count = 0;

findInRangeTree(root,value1,value2);

return count;

}

int BinarySearchTree::findInRangeTree(Node\* node, const Data& value1, const Data& value2) {

if (node == NULL) {

return 0;

}

else {

if (value1 <= node->data) {

findInRangeTree(node->LeftChild, value1, value2);

}

if (value1 <= node->data && value2 >= node->data) {

count++;

}

if (value2 >= node->data) {

findInRangeTree(node->RightChild, value1, value2);

}

}

return 0;

}

void BinarySearchTree::merge(BinarySearchTree tree1, BinarySearchTree tree2) {

BinarySearchTree tree3;

vector<Data> mergedVec;

tree1.addElementsTree(tree1.root, &mergedVec);

tree2.addElementsTree(tree2.root, &mergedVec);

sort(mergedVec.begin(), mergedVec.end());

binaryAddition(&mergedVec);

}

void BinarySearchTree::addElementsTree(Node\* node, vector<Data>\* v) {

if (node != NULL) {

addElementsTree(node->LeftChild,v);

v->push\_back(node->data);

addElementsTree(node->RightChild,v);

}

}

void BinarySearchTree::binaryAddition(vector <Data>\* v) {

int left = 0;

int right = v->size();

arrayToBST(\*v, left, right);

}

void BinarySearchTree::arrayToBST(vector <Data> v, int l, int r) {

if (l < r) {

int mid = (l + r) / 2;

insert(v[mid]);

arrayToBST(v, l, mid - 1);

arrayToBST(v, mid + 1, r);

}

if (r == l && r != v.size()) {

int mid = (l + r) / 2;

insert(v[mid]);

}

}

void BinarySearchTree::eraseRange(const Data& minObject, const Data& maxObject) {

eraseInRangeTree(root, minObject, maxObject);

}

void BinarySearchTree::eraseInRangeTree(Node\* node, const Data& value1, const Data& value2) {

if (node != NULL) {

if (value1 <= node->data) {

eraseInRangeTree(node->LeftChild, value1, value2);

}

if (value1 <= node->data && value2 >= node->data) {

erase(node->data);

}

if (value2 >= node->data) {

eraseInRangeTree(node->RightChild, value1, value2);

}

if (value1 <= node->data && value2 >= node->data) {

erase(node->data);

}

}

}

twoTrees split(BinarySearchTree tree1, const Data& object) {

Node\* root1 = NULL;

Node\* root2 = NULL;

vector<Data> lessObject;

vector<Data> moreObject;

tree1.distributeElements(object, tree1.root, &lessObject, &moreObject);

for (int i = 0; i < lessObject.size(); i++) {

root1 = insertAfterSplit(lessObject[i], root1, NULL);

}

for (int i = 0; i < moreObject.size(); i++) {

root2 = insertAfterSplit(moreObject[i], root2, NULL);

}

return {root1, root2};

}

void BinarySearchTree::distributeElements(const Data& object, Node\* node, vector <Data>\* vecLess, vector <Data>\* vecMore) {

if (node != NULL) {

distributeElements(object, node->LeftChild, vecLess, vecMore);

if (node->data < object) {

vecLess->push\_back(node->data);

}

else {

vecMore->push\_back(node->data);

}

distributeElements(object, node->RightChild, vecLess, vecMore);

}

}

Node\* insertAfterSplit(const Data& value, Node\* node, Node\* p) {

if (node == NULL) {

node = new Node;

node->data = value;

node->LeftChild = NULL;

node->RightChild = NULL;

node->parent = p;

}

if (value < node->data) {

node->LeftChild = insertAfterSplit(value, node->LeftChild, node);

}

else if (value > node->data) {

node->RightChild = insertAfterSplit(value, node->RightChild, node);

}

return node;

}

AVL.cpp

#include <iostream>

#include <algorithm>

#include "AVL.h"

using namespace std;

Node\* AVLTree::smallLeftRotate(Node\* node) {

Node\* R = node->RightChild;

Node\* RL = R->LeftChild;

R->LeftChild = node;

node->RightChild = RL;

R->height = 1 + max(height(R->LeftChild), height(R->RightChild));

node->height = 1 + max(height(node->LeftChild), height(node->RightChild));

return R;

}

Node\* AVLTree::smallRightRotate(Node\* node) {

Node\* L = node->LeftChild;

Node\* LR = L->RightChild;

L->RightChild = node;

node->LeftChild = LR;

L->height = 1 + max(height(L->LeftChild), height(L->RightChild));

node->height = 1 + max(height(node->LeftChild), height(node->RightChild));

return L;

}

bool AVLTree::insert(const Data& value) {

Node\* added = findNode(value, root);

if (added != NULL) {

return false;

}

root = addNode(value, root);

return true;

}

Node\* AVLTree::addNode(const Data& value, Node\* node) {

if (node == NULL) {

node = new Node;

node->data = value;

node->LeftChild = NULL;

node->RightChild = NULL;

node->height = 1;

}

if (value < node->data) {

node->LeftChild = addNode(value, node->LeftChild);

}

else if (value > node->data) {

node->RightChild = addNode(value, node->RightChild);

}

node->height = 1 + max(height(node->LeftChild), height(node->RightChild));

int heightDifference = height(node->RightChild) - height(node->LeftChild);

if (heightDifference > 1 && node->RightChild != NULL && value > node->RightChild->data) {

return smallLeftRotate(node);

}

if (heightDifference > 1 && node->RightChild != NULL && value < node->RightChild->data) {

node->RightChild = smallRightRotate(node->RightChild);

return smallLeftRotate(node);

}

if (heightDifference < -1 && node->LeftChild != NULL && value < node->LeftChild->data) {

return smallRightRotate(node);

}

if (heightDifference < -1 && node->LeftChild != NULL && value > node->LeftChild->data) {

node->LeftChild = smallLeftRotate(node->LeftChild);

return smallRightRotate(node);

}

return node;

}

int AVLTree::height(Node\* node) {

if (node == NULL) {

return 0;

}

else {

return 1 + max(height(node->LeftChild), height(node->RightChild));

}

}

Node\* AVLTree::findNode(const Data& value, Node\* node) {

if (node == NULL) {

return NULL;

}

else {

if (value > node->data) {

return findNode(value, node->RightChild);

}

else if (value < node->data) {

return findNode(value, node->LeftChild);

}

else {

return node;

}

}

return NULL;

}

void AVLTree::print() {

printTree(root);

}

void AVLTree::printTree(Node\* node) {

if (node != NULL) {

printTree(node->LeftChild);

cout << node->data.nickname << " " << node->data.rank << " " << node->data.amountOfExperience << " " << node->data.sizeDonut << " " << node->height <<endl;

printTree(node->RightChild);

}

}

PrefixTree.cpp

#include "PrefixTree.h"

using namespace std;

void Trie::insert(const string& s) {

TrieNode\* node = root;

for (auto x : s) {

if (node->myMap.count(x) == 0) {

node->myMap[x] = new TrieNode;

}

node = node->myMap[x];

}

node->lastWord = true;

}

vector<string> Trie::findByPrefix(const string& s) {

TrieNode\* node = root;

unordered\_map<char, TrieNode\*>:: iterator itTmp;

vector <string> prefixWords;

for (auto x : s) {

node = node->myMap[x];

}

string str = s;

print(node, str);

return prefixWords;

}

void Trie::print(TrieNode\* node, string str) {

for (auto it = node->myMap.begin(); it != node->myMap.end(); it++) {

if (it->second->lastWord == 0) {

cout << str + it->first << " ";

}

else {

print(it->second, str + it->first);

}

}

}

Висновки:

Ознайомилась і дослідила структуру даних бінарне дерево пошуку та префіксне дерево, розглянутла механізми балансування дерева. Набула навичок реалізації бінарного дерева пошуку мовою програмування С++, порівняла власну реалізацію з готовим бібліотечним рішенням STL.