

Алгоритми та структури даних

**Лабораторна робота №8**

**“Дослідження структури даних бінарне дерево пошуку”**

Робота : Лещинського Б.Д. группа КА-07

06.05.2021

Варіант 13

**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.

**Хід виконання роботи:**

Базове завдання:

Обрати одну із запропонованих задач та реалізувати бінарне дерево пошуку для ефективнного вирішення цієї задачі.

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:

2.1 Створити базовий елемент дерева Node, що буде містити в собі дані та вказівники на двох нащадків – лівого та правого. За необхідності додати вказівник на батьківський вузол.

2.2 Створити структуру BinarySearchTree з основними методами бінарного дерева пошуку без балансування:

• insert(object) додати новий елемент в дерево (без повторень) • find(object) перевірити наявність елемента в дереві

• erase(object) видалити елемент з дерева

• size() знайти кількість елементів в дереві

2.3 Реалізувати додаткові методи для роботи з деревом:

• print() вивести всі елементи дерева у відсортованому порядку

• height() знайти висоту дерева

• findInRange(minObject, maxObject) знайти всі елементи в

дереві на проміжку [minObject, maxObject], повернути їх

кількість (або динамічний масив (vector) з цими елементами)

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

Додаткові завдання:

1. Реалізувати методи бінарного дерева пошуку для роботи з піддеревами:

• merge(tree1, tree2) об’єднати два дерева в одне

• split(tree, object) розділити бінарне дерево пошуку по ключу на два інших, в першому всі елементи < object, в другому >= object

• eraseRange(minObject, maxObject) видалити всі елементи дерева на проміжку [minObject, maxObject], використавши створені вище методи

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

4 Доповнене завдання

Реалізувати префіксне дерево для роботи програми автодоповнення слів. Приклад виконання:

Input: > algorit

Output: > algorithm, algorithmic, algorithmically, algorithms

1. Створити базовий елемент дерева Node, що буде містити хеш-таблицю, де ключ – це текстовий символ, значення – вказівник на відповідний Node, а також булеву помітку чи є цей вузол кінцем слова. Замість хеш-таблиці можна використати статичний або динамічний масив, хоча це не дуже оптимально.

2. Створити структуру Trie та реалізувати основні методи префіксного дерева:

• insert(word) додати нове слово в дерево

• findByPrefix(word) знайти всі слова, які починаються на заданий префікс

3. Зчитати з файлу всі існуючі слова та побудувати з ними префіксне дерево. Файл зі словами можна взяти тут: https://github.com/dwyl/english-words/raw/master/words\_alpha.txt <https://github.com/dwyl/english-words/blob/master/words_alpha.zip?raw=true>

4. Протестувати правильність роботи префіксного дерева, використавши різні префікси, що вводяться з клавіатури.

**Код програми(базове завдання та додаткові завдання):**

#include <iostream>

#include <set>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <cmath>

#include <string>

#include <iomanip>

using namespace std;

long long generateRandLong() {

long long generated = rand() % 9 + 1;

for (int i = 1; i < 10; i++) {

generated \*= 10;

generated += rand() % 10;

}

return generated;

}

std::string possibleNicks[] = { "User111","Skired","MonsterKiller","CatBoiCami" };

struct Data {

std::string username;

int rank;

int experience;

int communityConr;

Data() {

username = possibleNicks[rand() % 3];

rank = rand() % 10000 + 1;

experience = rand() % 10000;

communityConr = rand() % 1000;

}

bool operator< (const Data other) const {

return (this->rank + this->experience + this->communityConr) < (other.rank + other.experience + other.communityConr);

}

bool operator> (const Data other) const {

return (this->rank + this->experience + this->communityConr) > (other.rank + other.experience + other.communityConr);

}

bool operator== (const Data other) const {

return (this->rank + this->experience + this->communityConr) == (other.rank + other.experience + other.communityConr);

}

};

struct TreeNode {

Data player;

TreeNode\* left = nullptr;

TreeNode\* right = nullptr;

int height = 1;

};

struct BinarySearchTree {

TreeNode\* root = nullptr;

int entities = 0;

//

bool search(Data& somePlayer, TreeNode\*& currentNode) {

if (currentNode == nullptr) return false;

if (currentNode->player == somePlayer) return true;

if (currentNode->player < somePlayer)

return search(somePlayer, currentNode->right);

else return search(somePlayer, currentNode->left);

}

bool find(Data& somePlayer) {

return search(somePlayer, root);

}

//

//

TreeNode\* avl\_insert(TreeNode\*& sub\_root, TreeNode\*& insertingPlayer) {

if (insertingPlayer->player > sub\_root->player) {

if (sub\_root->right == nullptr) {

sub\_root->right = insertingPlayer;

}

else {

sub\_root->right = avl\_insert(sub\_root->right, insertingPlayer);

}

}

else {

if (sub\_root->left == nullptr) {

sub\_root->left = insertingPlayer;

}

else {

sub\_root->left = avl\_insert(sub\_root->left, insertingPlayer);

}

}

return balanceNode(sub\_root);

}

void insert(Data& newPlayer) {

if (find(newPlayer)) return; //unique check

TreeNode\* playerToInsert = new TreeNode;

playerToInsert->player = newPlayer;

entities++;

if (root == nullptr) { //empty tree case

root = playerToInsert;

return;

}

else {

root = avl\_insert(root, playerToInsert);

}

}

//

//functions needed for erase() (now in AVL-tree)

TreeNode\* minimum(TreeNode\*& someTree) {

if (someTree->left == nullptr) return someTree;

else return minimum(someTree->left);

}

TreeNode\* removeMinimal(TreeNode\*& node) {

if (node->left == nullptr) {

return node->right;

}

node->left = removeMinimal(node->left);

return balanceNode(node);

}

TreeNode\* avl\_erase(TreeNode\*& sub\_root, Data& deletePlayer) {

if (sub\_root == nullptr) return 0;

if (deletePlayer < sub\_root->player) sub\_root->left = avl\_erase(sub\_root->left, deletePlayer);

else if(deletePlayer > sub\_root->player) sub\_root->right = avl\_erase(sub\_root->right, deletePlayer);

else {

TreeNode\* rightChild = sub\_root->right;

TreeNode\* leftChild = sub\_root->left;

delete sub\_root;

sub\_root = nullptr;

if (rightChild == nullptr) return leftChild;

TreeNode\* minimal = minimum(rightChild);

minimal->right = removeMinimal(rightChild);

minimal->left = leftChild;

return balanceNode(minimal);

}

return balanceNode(sub\_root);

}

void erase(Data& deletePlayer) {

if (!find(deletePlayer)) return;

root = avl\_erase(root, deletePlayer);

entities--;

}

//

int size() { return entities; }

//

void inorderTraversal(TreeNode\*& currNode) {

if (currNode != nullptr) {

inorderTraversal(currNode->left);

std::cout << std::setw(15) << "username:" << currNode->player.username << "\trank:" << currNode->player.rank << "\texpetience:" << currNode->player.experience << "\tcommunity contribution:" << currNode->player.communityConr << std::endl;

inorderTraversal(currNode->right);

}

}

void print() {

inorderTraversal(root);

}

//

//

int maxDepth(TreeNode\* &treeRoot) {

if (treeRoot == nullptr) {

return 0;

}

else {

int leftDepth = maxDepth(treeRoot->left);

int rightDepth = maxDepth(treeRoot->right);

if (leftDepth > rightDepth) return leftDepth + 1;

else return rightDepth + 1;

}

}

int height() {

return maxDepth(root);

}

//

//

void lookInRange(Data& minObj, Data& maxObj, TreeNode\*& treeRoot, int& inRange) {

if (treeRoot != nullptr) {

if (treeRoot->player < minObj) lookInRange(minObj, maxObj, treeRoot->right, inRange);

if (treeRoot->player > maxObj) lookInRange(minObj, maxObj, treeRoot->left, inRange);

if ((treeRoot->player > minObj || treeRoot->player == minObj) && (treeRoot->player < maxObj || treeRoot->player == maxObj)) {

lookInRange(minObj, maxObj, treeRoot->left, inRange);

inRange++;

lookInRange(minObj, maxObj, treeRoot->right, inRange);

}

}

}

int findInRange(Data& minObj, Data& maxObj) {

int numberInrange = 0;

lookInRange(minObj, maxObj, root, numberInrange);

return numberInrange;

}

//

//balancing methods(ritations for AVL tree)

int getHeight(TreeNode\*& node) {

return node ? node->height : 0;

}

int balanceFactor(TreeNode\*& node) {

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

}

void fixHeight(TreeNode\*& node) {

int rightH = getHeight(node->right);

int leftH = getHeight(node->left);

node->height = (rightH > leftH ? rightH : leftH) + 1;

}

TreeNode\* rotateRight(TreeNode\*& node) {

TreeNode\* leftChild = node->left;

node->left = leftChild->right;

leftChild->right = node;

if (node == root) {

root = leftChild;

}

fixHeight(node);

fixHeight(leftChild);

return leftChild;

}

TreeNode\* rotateLeft(TreeNode\*& node) {

TreeNode\* rightChild = node->right;

node->right = rightChild->left;

rightChild->left = node;

if (node == root) {

root = rightChild;

}

fixHeight(node);

fixHeight(rightChild);

return rightChild;

}

TreeNode\* balanceNode(TreeNode\*& node) {

fixHeight(node);

if (balanceFactor(node) == 2) {

if (balanceFactor(node->right) < 0) node->right = rotateRight(node->right);

node = rotateLeft(node);

}

if (balanceFactor(node) == -2) {

if (balanceFactor(node->left) > 0) node->left = rotateLeft(node->left);

node = rotateRight(node);

}

// otherwise it is already balanced

return node;

}

//

// some additional methods:

//

void deleteRange(Data& minObj, Data& maxObj, TreeNode\*& treeRoot) {

if (treeRoot != nullptr) {

if (treeRoot->player < minObj) deleteRange(minObj, maxObj, treeRoot->right);

if (treeRoot->player > maxObj) deleteRange(minObj, maxObj, treeRoot->left);

if ((treeRoot->player > minObj || treeRoot->player == minObj) && (treeRoot->player < maxObj || treeRoot->player == maxObj)) {

erase(treeRoot->player);

deleteRange(minObj, maxObj, root);

}

}

}

void eraseRange(Data& minObj, Data& maxObj) {

deleteRange(minObj, maxObj, root);

}

//

//

void inorderAdding(TreeNode\*& currNode) {

if (currNode != nullptr) {

inorderAdding(currNode->left);

insert(currNode->player);

inorderAdding(currNode->right);

}

}

void merge(BinarySearchTree& otherTree) {

inorderAdding(otherTree.root);

}

//

//

void spliting(TreeNode\* sub\_root, Data& obj, BinarySearchTree\* tree1, BinarySearchTree\* tree2) {

if (sub\_root != nullptr) {

spliting(sub\_root->left, obj, tree1, tree2);

if (sub\_root->player < obj) tree1->insert(sub\_root->player);

else tree2->insert(sub\_root->player);

spliting(sub\_root->right, obj, tree1, tree2);

}

}

pair<BinarySearchTree\*, BinarySearchTree\*> split(Data& obj) {

BinarySearchTree\* tree1 = new BinarySearchTree;

BinarySearchTree\* tree2 = new BinarySearchTree;

spliting(root, obj, tree1, tree2);

return { tree1, tree2 };

}

//

};

bool testBinarySearchTree()

{

srand(time(NULL));

const int iters = 80000;

const int keysAmount = iters \* 2;

const int itersToRangeQueries = 1000;

vector<Data> data(keysAmount);

vector<Data> dataToInsert(iters);

vector<Data> dataToErase(iters);

vector<Data> dataToFind(iters);

vector<pair<Data, Data>> dataToRangeQueries;

for (int i = 0; i < iters; i++)

{

dataToInsert[i] = data[generateRandLong() % keysAmount];

dataToErase[i] = data[generateRandLong() % keysAmount];

dataToFind[i] = data[generateRandLong() % keysAmount];

}

for (int i = 0; i < itersToRangeQueries; i++)

{

Data minData = Data();

Data maxData = Data();

if (maxData < minData)

{

swap(minData, maxData);

}

dataToRangeQueries.push\_back({ minData, maxData });

}

BinarySearchTree myTree;

clock\_t myStart = clock();

for (int i = 0; i < iters; i++)

{

myTree.insert(dataToInsert[i]);

}

int myInsertSize = myTree.size();

int myTreeHeight = myTree.height();

int optimalTreeHeight = log2(myInsertSize) + 1;

for (int i = 0; i < iters; i++)

{

myTree.erase(dataToErase[i]);

}

int myEraseSize = myInsertSize - myTree.size();

int myFoundAmount = 0;

for (int i = 0; i < iters; i++)

{

if (myTree.find(dataToFind[i]))

{

myFoundAmount++;

}

}

clock\_t myEnd = clock();

float myTime = (float(myEnd - myStart)) / CLOCKS\_PER\_SEC;

set<Data> stlTree;

clock\_t stlStart = clock();

for (int i = 0; i < iters; i++)

{

stlTree.insert(dataToInsert[i]);

}

int stlInsertSize = stlTree.size();

for (int i = 0; i < iters; i++)

{

stlTree.erase(dataToErase[i]);

}

int stlEraseSize = stlInsertSize - stlTree.size();

int stlFoundAmount = 0;

for (int i = 0; i < iters; i++)

{

if (stlTree.find(dataToFind[i]) != stlTree.end())

{

stlFoundAmount++;

}

}

clock\_t stlEnd = clock();

float stlTime = (float(stlEnd - stlStart)) / CLOCKS\_PER\_SEC; clock\_t myRangeStart = clock();

int myRangeFoundAmount = 0;

for (int i = 0; i < itersToRangeQueries; i++)

{

myRangeFoundAmount += myTree.findInRange(dataToRangeQueries[i].first,

dataToRangeQueries[i].second);

}

clock\_t myRangeEnd = clock();

float myRangeTime = (float(myRangeEnd - myRangeStart)) / CLOCKS\_PER\_SEC;

clock\_t stlRangeStart = clock();

int stlRangeFoundAmount = 0;

for (int i = 0; i < itersToRangeQueries; i++)

{

const auto& low = stlTree.lower\_bound(dataToRangeQueries[i].first);

const auto& up = stlTree.upper\_bound(dataToRangeQueries[i].second);

stlRangeFoundAmount += distance(low, up);

}

clock\_t stlRangeEnd = clock();

float stlRangeTime = (float(stlRangeEnd - stlRangeStart)) / CLOCKS\_PER\_SEC;

cout << "My BinaryTree: height = " << myTreeHeight << ", optimal height = " << optimalTreeHeight << endl;

cout << "Time: " << myTime << ", size: " << myInsertSize << " - " << myEraseSize << ", found amount : " << myFoundAmount << endl;

cout << "Range time: " << myRangeTime << ", range found amount: " << myRangeFoundAmount << endl << endl;

cout << "STL Tree:" << endl;

cout << "Time: " << stlTime << ", size: " << stlInsertSize << " - " << stlEraseSize << ", found amount: " << stlFoundAmount << endl;

cout << "Range time: " << stlRangeTime << ", range found amount: " << stlRangeFoundAmount << endl << endl;

if (myInsertSize == stlInsertSize && myEraseSize == stlEraseSize && myFoundAmount == stlFoundAmount && myRangeFoundAmount == stlRangeFoundAmount)

{

cout << "The lab is completed" << endl;

return true;

}

cerr << ":(" << endl;

return false;

}

int main()

{

srand(time(NULL));

testBinarySearchTree();

cout << endl;

vector<Data> players(10000);

for (int i = 0; i < 10000; i++) {

players[i] = Data();

}

vector<pair<Data, Data>> dataToRangeQueries(10);

for (int i = 0; i < 10; i++)

{

Data minData = Data();

Data maxData = Data();

if (maxData < minData)

{

swap(minData, maxData);

}

dataToRangeQueries.push\_back({ minData, maxData });

}

BinarySearchTree playersTree;

for (int i = 0; i < 10000; i++) {

playersTree.insert(players[i]);

}

std::cout << std::endl;

for (int i = 0; i < 5000; i++) {

playersTree.erase(players[i]);

}

int myRangeFoundAmount = 0;

int beforeRangeErase = playersTree.size();

for (int i = 0; i < 10; i++)

{

myRangeFoundAmount += playersTree.findInRange(dataToRangeQueries[i].first, dataToRangeQueries[i].second);

playersTree.eraseRange(dataToRangeQueries[i].first, dataToRangeQueries[i].second);

}

bool correctRangeErase = false;

beforeRangeErase - myRangeFoundAmount == playersTree.size() ? correctRangeErase = true : correctRangeErase = false;

cout << "Tree size before rangeErase(): " << beforeRangeErase << "\tmyRangeFoundAmount:" << myRangeFoundAmount <<"\tAfter: "<<playersTree.size()<< "\ncorrectRangeErase: " << correctRangeErase;

cout << endl;

}

**Код програми(доповнене завдання):**

#include <iostream>

#include <fstream>

#include <iomanip>

#include <string>

#include <unordered\_map>

#include <vector>

struct Node {

bool isLeaf = false;

std::unordered\_map<char, Node\*> children;

std::vector<char> children\_keys;

};

struct Trie {

Node\* root;

Trie() {

root = new Node;

root->isLeaf = false;

}

void insert(std::string word) {

Node\* initialRoot = root;

for (int i = 0; i < word.length(); i++) {

if (root->children[word[i]] == 0) {

root->children\_keys.push\_back(word[i]);

root->children[word[i]] = new Node;

if(i == word.length() - 1) {

root->children[word[i]]->isLeaf = true;

root = initialRoot;

return;

}

}

root = root->children[word[i]];

}

root = initialRoot;

}

void findSuffixes(std::vector<std::string>\* resultWords, Node\* sub\_root, std::string prefix) {

for (int i = 0; i < sub\_root->children\_keys.size(); i++) {

std::string initialPrefix = prefix;

findSuffixes(resultWords, sub\_root->children[sub\_root->children\_keys[i]], prefix.append(1, sub\_root->children\_keys[i]));

prefix = initialPrefix;

}

if(sub\_root->isLeaf == true) {

resultWords->push\_back(prefix);

}

}

std::vector<std::string> findByPrefix(std::string prefix) {

std::vector<std::string> resultWords;

Node\* initialRoot = root;

for (int i = 0; i < prefix.length(); i++) {

if(root->children[prefix[i]] == 0){

std::cout << "No such prefix in Trie((";

return {}; // returns an empty vector

}

root = root->children[prefix[i]];

}

findSuffixes(&resultWords, root, prefix);

root = initialRoot;

return resultWords;

}

};

void fillTree(Trie\* trie) {

std::string word;

std::ifstream file("words\_alpha.txt");

while (getline(file, word)) {

trie->insert(word);

}

file.close();

}

void printOutVector(std::vector<std::string> wordsVec) {

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

std::cout << std::setw(5) << wordsVec[i] << ", ";

}

}

int main()

{

Trie trie;

std::string word;

int choise = 0;

std::vector<std::string> foundWords;

fillTree(&trie);

std::cout << trie.root->children.size();

while (true) {

system("cls");

std::cout << "Enter a word for autocompletion: ";

std::cin >> word;

foundWords = trie.findByPrefix(word);

printOutVector(foundWords);

std::cout << std::endl;

std::cout << "Countinue? (1/0) :";

std::cin >> choise;

if (choise == 0) {

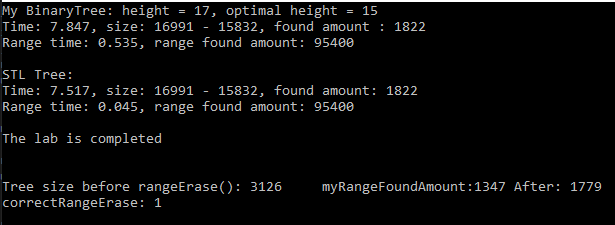
break;

}

}

}

**Результати роботи програми:**





**Висновки:**

Я ознайомився і дослідив структури даних бінарне дерево пошуку та префіксне дерево, розглянув механізми балансування дерева. Набув навичок реалізації бінарного дерева пошуку мовою програмування С++, порівняв власну реалізацію з готовим бібліотечним рішенням STL. А також як додаткове завдання розібрав та імпдементував префіксне дерево для зберігання слів та реалзіції функції автодоповнення сліва на його базію