Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра системного проектування

Алгоритми та структури даних 2

Лабораторна робота №4

“Дослідження структур даних бінарне дерево пошуку”

Аносов Ілля

ДА-21

Київ

2023

**Мета:** 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 1**

**Завдання**

Структура Гравець має наступні поля: нікнейм, ранг, кількість досвіду, статистика перемог. Створити відсортовану “базу даних” гравців, в якій можна швидко перевіряти наявність потрібного гравця та знаходити всіх гравців на вказаному проміжку (між двома гравцями).

1. **Створити структуру Data** 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

**Додаткові** **завдання**

Доповнити реалізацію бінарного дерева пошуку з основного завдання:

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

2. Розв’язати задачу “Однакові дерева”:

* Реалізувати функцію bool isSameTree(Node\* t1, Node\* t2) – перевірити чи являються два вхідні дерева однаковими по структурі та по значенням у вузлах
* Перевірити своє рішення на тестах в системі:3. 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], з використанням створених вище методів

Зображення, що містить текст, знімок екрана, програмне забезпечення, комп’ютер

Автоматично згенерований опис

Виконання основного завдання

Зображення, що містить програмне забезпечення, Мультимедійне програмне забезпечення, Графічний редактор, Редагування

Автоматично згенерований опис

Додаткове завдання «Порівняння дерев»

**Код програми**

*binarySearchTree.h*

#pragma once

#include <iostream>

#include <string>

#include <vector>

using namespace std;

struct Data {

    string nickname;

    int rang;

    int exp;

    int victoryStat;

    Data() {

        nickname = "-";

        rang = 0;

        exp = 0;

        victoryStat = 0;

    }

    Data(string nickname) {

        this->nickname = nickname;

        rang = 0;

        exp = 0;

        victoryStat = 0;

    }

    Data(int exp) {

        nickname = "-";

        rang = 0;

        this->exp = exp;

        victoryStat = 0;

    }

    Data(string nickname, int rang, int exp, int victoryStat) :

        nickname(nickname),

        rang(rang),

        exp(exp),

        victoryStat(victoryStat) {}

    bool operator<(const Data& second) const {

        if (this->nickname != second.nickname)

            return this->nickname < second.nickname;

        if (this->rang != second.rang)

            return this->rang < second.rang;

        if (this->exp != second.exp)

            return this->exp < second.exp;

        if (this->victoryStat != second.victoryStat)

            return this->victoryStat < second.victoryStat;

        return false;

    }

    bool operator>(const Data& second) const {

        if (this->nickname != second.nickname)

            return this->nickname > second.nickname;

        if (this->rang != second.rang)

            return this->rang > second.rang;

        if (this->exp != second.exp)

            return this->exp > second.exp;

        if (this->victoryStat != second.victoryStat)

            return this->victoryStat > second.victoryStat;

        return false;

    }

    bool operator==(const Data& second) const {

        if (this->nickname == second.nickname && this->rang == second.rang && this->exp == second.exp && this->victoryStat == second.victoryStat)

            return true;

        return false;

    }

    bool operator>=(const Data& second) const {

        return !(\*this<(second));

    }

    bool operator<=(const Data& second) const {

        return !(\*this>(second));

    }

    bool operator!=(const Data& second) const {

        if (\*this == second)

            return false;

        return true;

    }

};

struct Node {

    Data gamer;

    Node\* leftN;

    Node\* rightN;

    Node() {

        this->gamer = Data();

        this->leftN = nullptr;

        this->rightN = nullptr;

    }

    Node(Data gamer) {

        this->gamer = gamer;

        this->leftN = nullptr;

        this->rightN = nullptr;

    }

};

struct BinarySearchTree {

    Node\* headNode = nullptr;

    Node\* insert(Data& data);

    bool find(const Data& data);

    void erase(Data& data);

    int size();

    void print();

    int height();

    vector<Data> findVectorInRange(Data minObject, Data maxObject);

    int findInRange(Data minObject, Data maxObject);

    bool isSameTree(Node\* curNode1, Node\* curNode2);

    Node\* split(const Data data);

    void merge(Node\* tree1, Node\* tree2);

    void eraseRange(Data minObject, Data maxObject);

private:

    Node\* search(Node\* curNode, const Data& data);

    Node\* insertInner(Node\* curNode, const Data& data);

    bool findInner(Node\* curNode, const Data& data);

    void printInner(Node\* curNode);

    void printKey(Data\* data);

    int sizeInner(Node\* curNode);

    int heightInner(Node\* curNode);

    int findInnerInRange(Node\* curNode, Data\* minObject, Data\* maxObject);

    vector<Data> findInnerVectorInRange(Node\* curNode, Data\* minObject, Data\* maxObject);

    void balance(Node\* parentNode);

    void bigLeftRot(Node\* parentNode);

    void bigRightRot(Node\* parentNode);

    void smallLeftRot(Node\* parentNode);

    void smallRightRot(Node\* parentNode);

};

*binarySearchTree.cpp*

#include "binarySearchTree.h"

Node\* BinarySearchTree::insert(Data& data) {

    if(!headNode) {

        Node\* newNode = new Node(data);

        headNode = newNode;

        return nullptr;

    }

    insertInner(headNode, data);

    balance(headNode);

}

Node\* BinarySearchTree::insertInner(Node\* curNode, const Data& data) {

    if (!curNode) {

        Node\* newNode = new Node(data);

        curNode = newNode;

        return newNode;

    }

    if (curNode->gamer < data)

        curNode->rightN = insertInner(curNode->rightN, data);

    if (curNode->gamer > data)

        curNode->leftN = insertInner(curNode->leftN, data);

    return curNode;

}

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

    return findInner(headNode, data);

}

bool BinarySearchTree::findInner(Node\* curNode, const Data& data) {

    if (!curNode)

        return false;

    if (curNode->gamer < data)

        return findInner(curNode->rightN, data);

    if (curNode->gamer > data)

        return findInner(curNode->leftN, data);

    return true;

}

void BinarySearchTree::erase(Data& data) {

    Node\* found = search(headNode, data);

    if (!found)

        return;

    if (!found->leftN && !found->rightN) {//No child

        if (found == headNode) {

            delete headNode;

            headNode = nullptr;

            return;

        }

        delete found;

        found = nullptr;

        return;

    }

    if (found->leftN && !found->rightN) {//Only left child

        Node\* deleteNode = found->leftN;

        found->gamer = deleteNode->gamer;

        found->leftN = deleteNode->leftN;

        found->rightN = deleteNode->rightN;

        delete deleteNode;

        return;

    }

    if (!found->leftN && found->rightN) {//Only right child

        Node\* deleteNode = found->rightN;

        found->gamer = deleteNode->gamer;

        found->leftN = deleteNode->leftN;

        found->rightN = deleteNode->rightN;

        delete deleteNode;

        return;

    }

    //Two child

    Node\* curNode = found->rightN;

    Node\* parentCurNode = found;

    while (curNode->leftN) {//Find right min element

        parentCurNode = curNode;

        curNode = curNode->leftN;

    }

    found->gamer = curNode->gamer;

    if (curNode->rightN) {//If min has right child

        Node\* deleteNode = curNode->rightN;

        curNode->gamer = deleteNode->gamer;

        curNode->leftN = deleteNode->leftN;

        curNode->rightN = deleteNode->rightN;

        delete deleteNode;

        return;

    }

    //If min has not any child

    delete curNode;

    parentCurNode->rightN = nullptr;

    return;

}

Node\* BinarySearchTree::search(Node\* curNode, const Data& data) {

    if (!curNode || curNode->gamer == data)

        return curNode;

    if (curNode->gamer < data)

        return search(curNode->rightN, data);

    if (curNode->gamer > data)

        return search(curNode->leftN, data);

}

int BinarySearchTree::size() {

    return sizeInner(headNode);

}

int BinarySearchTree::sizeInner(Node\* curNode) {

    if (!curNode)

        return 0;

    return 1 + sizeInner(curNode->leftN) + sizeInner(curNode->rightN);

}

void BinarySearchTree::print() {

    printInner(headNode);

}

void BinarySearchTree::printInner(Node\* curNode) {

    if (!curNode)

        return;

    printInner(curNode->leftN);

    printKey(&curNode->gamer);

    printInner(curNode->rightN);

}

void BinarySearchTree::printKey(Data\* data) {

    cout << data->nickname << ": rang " << data->rang << ", " << data->exp << " exp, statistic: " << data->victoryStat << endl;

}

int BinarySearchTree::height() {

    return heightInner(headNode) - 1;

}

int BinarySearchTree::heightInner(Node\* curNode) {

    if (!curNode)

        return 0;

    int hL = heightInner(curNode->leftN);

    int hR = heightInner(curNode->rightN);

    if (hL >= hR)

        return 1 + hL;

    return 1 + hR;

}

int BinarySearchTree::findInRange(Data minObject, Data maxObject) {

    return findInnerInRange(headNode, &minObject, &maxObject);

}

int BinarySearchTree::findInnerInRange(Node\* curNode, Data\* minObject, Data\* maxObject) {

    if (!curNode)

        return 0;

    int N = 0;

    if (curNode->gamer >= \*minObject)

        N += findInnerInRange(curNode->leftN, minObject, maxObject);

    if (curNode->gamer >= \*minObject && curNode->gamer <= \*maxObject)

        N++;

    if (curNode->gamer <= \*maxObject)

        N += findInnerInRange(curNode->rightN, minObject, maxObject);

    return N;

}

vector<Data> BinarySearchTree::findVectorInRange(Data minObject, Data maxObject) {

    return findInnerVectorInRange(headNode, &minObject, &maxObject);

}

vector<Data> BinarySearchTree::findInnerVectorInRange(Node\* curNode, Data\* minObject, Data\* maxObject) {

    if (!curNode)

        return {};

    vector<Data> get1, get2;

    if(curNode->gamer >= \*minObject)

        get1 = findInnerVectorInRange(curNode->leftN, minObject, maxObject);

    if (curNode->gamer >= \*minObject && curNode->gamer <= \*maxObject)

        get1.push\_back(curNode->gamer);

    if (curNode->gamer <= \*maxObject) {

        get2 = findInnerVectorInRange(curNode->rightN, minObject, maxObject);

        get1.insert(get1.end(), get2.begin(), get2.end());

    }

    return get1;

}

void BinarySearchTree::balance(Node\* parentNode) {

    if (!parentNode)

        return;

    int hL = heightInner(parentNode->leftN);

    int hR = heightInner(parentNode->rightN);

    while(hR - hL < -1) {

        bigRightRot(parentNode);

        //

        hL = heightInner(parentNode->leftN);

        hR = heightInner(parentNode->rightN);

    }

    while(hR - hL > 1) {

        bigLeftRot(parentNode);

        //

        hL = heightInner(parentNode->leftN);

        hR = heightInner(parentNode->rightN);

    }

    balance(parentNode->leftN);

    balance(parentNode->rightN);

    return;

}

void BinarySearchTree::smallLeftRot(Node\* parentNode) {

    if (!parentNode || !parentNode->rightN)

        return;

    Node\* newLeft = new Node();

    Node\* toDelete = parentNode->rightN;

    newLeft->gamer = parentNode->gamer;

    newLeft->leftN = parentNode->leftN;

    parentNode->leftN = newLeft;

    parentNode->gamer = toDelete->gamer;

    parentNode->rightN = toDelete->rightN;

    delete toDelete;

}

void BinarySearchTree::smallRightRot(Node\* parentNode) {

    if (!parentNode || !parentNode->leftN)

        return;

    Node\* newRight = new Node();

    Node\* toDelete = parentNode->leftN;

    newRight->gamer = parentNode->gamer;

    newRight->rightN = parentNode->rightN;

    parentNode->rightN = newRight;

    parentNode->gamer = toDelete->gamer;

    parentNode->leftN = toDelete->leftN;

    delete toDelete;

}

void BinarySearchTree::bigLeftRot(Node\* parentNode) {

    if (!parentNode || !parentNode->rightN)

        return;

    if (!parentNode->rightN->leftN) {

        smallLeftRot(parentNode);

        return;

    }

    if (parentNode->rightN->leftN->rightN) {

        while (parentNode->rightN->leftN->rightN) {

            bigLeftRot(parentNode->rightN->leftN);

        }

    }

    if (parentNode->rightN->leftN) {

        while (parentNode->rightN->leftN) {

            bigRightRot(parentNode->rightN);

        }

    }

    smallLeftRot(parentNode);

    return;

}

void BinarySearchTree::bigRightRot(Node\* parentNode) {

    if (!parentNode || !parentNode->leftN)

        return;

    if (!parentNode->leftN->rightN) {

        smallRightRot(parentNode);

        return;

    }

    if (parentNode->leftN->rightN->leftN) {

        while (parentNode->leftN->rightN->leftN) {

            bigRightRot(parentNode->leftN->rightN);

        }

    }

    if (parentNode->leftN->rightN) {

        while (parentNode->leftN->rightN) {

            bigLeftRot(parentNode->leftN);

        }

    }

    smallLeftRot(parentNode);

    return;

}

bool BinarySearchTree::isSameTree(Node\* p, Node\* q) {

    if (!p && !q)

        return true;

    if (!p || !q)

        return false;

    if (p->gamer != q->gamer)

        return false;

    if (!isSameTree(p->leftN, q->leftN) || !isSameTree(p->rightN, q->rightN))

        return false;

    return true;

}

Node\* BinarySearchTree::split(const Data data) {

    if (!headNode)

        return nullptr;

    Node\* toReturn = headNode;

    if (headNode->gamer == data) {//split on head

        headNode = headNode->leftN;

        toReturn->leftN = nullptr;

        return toReturn;

    }

    if (headNode->gamer > data) {//split on left

        Node\* endLeftSearch = headNode;

        Node\* rightSearch = headNode;

        while (endLeftSearch->leftN && endLeftSearch->leftN->gamer > data) {

            endLeftSearch = endLeftSearch->leftN;

        }

        if (!endLeftSearch->leftN) {

            headNode = nullptr;

            return toReturn;

        }

        headNode = endLeftSearch->leftN;

        rightSearch = headNode;

        while (rightSearch->rightN && rightSearch->rightN->gamer < data) {

            rightSearch = rightSearch->rightN;

        }

        if (!endLeftSearch) {

            toReturn = rightSearch->rightN;

            if (rightSearch->rightN)

                rightSearch->rightN = rightSearch->rightN->leftN;

            return toReturn;

        }

        endLeftSearch->leftN = rightSearch->rightN;

        if (rightSearch->rightN)

            rightSearch->rightN = rightSearch->rightN->leftN;

        return toReturn;

    }

    else {//split on right

        Node\* current = nullptr;

        Node\* rightSearch = headNode;

        while (rightSearch->rightN && rightSearch->rightN->gamer < data) {

            rightSearch = rightSearch->rightN;

        }

        toReturn = rightSearch->rightN;

        if (!toReturn)

            return nullptr;

        if (!toReturn->leftN)

            return toReturn;

        current = toReturn->leftN;

        return nullptr;

        while (current->leftN && current->leftN->gamer > data) {

            current = current->leftN;

        }

    }

}

void BinarySearchTree::merge(Node\* tree1, Node\* tree2) {

    Node\* curT1 = tree1;

    Node\* curT2 = tree2;

    if (!tree1 || !tree2)

        return;

    if (tree1->gamer < tree2->gamer) {

        while (curT1->rightN)

            curT1 = curT1->rightN;

        curT1->rightN = tree2;

        return;

    }

    while (curT2->rightN)

        curT2 = curT2->rightN;

    curT2->rightN = tree1;

}

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

    Node\* rightRange = split(maxObject);

    Node\* splited = split(minObject);

    merge(rightRange, headNode);

}

**Висновок:**

В результаті виконання лабораторної роботи №4 було створено бінарне дерево з додатковим балансуванням по типу АВЛ-дерева. До стандартних функцій додано функції порівняння двох дерев, об’єднання або розділення на певному значення та видалення його частини на певному проміжку. Перевизначено оператори всі оператори порівняння для полегшеного створення структури та її коректної роботи. Одним з головних переваг такого дерева є те, що всі елементи відсортовані за принципом: менший елемент стоїть ліворуч, більший – праворуч. Таким чином, можна отримати будь-який елемент за короткий проміжок часу, легко вивести всі елементи у відсортованому порядку, отримати впорядковані значення на певному проміжку та інше. В більшості випадків основні функції, такі як додавання, видалення, пошук, працюють за асимптотикою O(logN), що дозволяє значно зменшити витрати часу при роботі з великими масивами.