**САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО**

**ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И КИБЕРБЕЗОПАСНОСТИ**

**ВЫСШАЯ ШКОЛА ПРОГРАММНОЙ ИНЖЕНЕРИИ**

**Лабораторная работа №3**

**«Бинарное дерево поиска»**

**по предмету «Алгоритмы и структуры данных»**

**Выполнил: студент гр. 5130904/30002 Севостьянова А.В.**

**Руководитель: Череповский Д.К.**

**Санкт-Петербург**

**2024 г.**

Оглавление

[1. Общая постановка задачи 3](#_Toc162211006)

[2. Детальные требования и тест план 6](#_Toc162211007)

[Детальные требования: 6](#_Toc162211008)

[Таблица с детальными требованиями и тест планом 6](#_Toc162211009)

[3. Программа 6](#_Toc162211010)

[Main.cpp 7](#_Toc162211011)

[BinarySearchTree.h 7](#_Toc162211012)

[Structs.h 12](#_Toc162211013)

[Приложение 15](#_Toc162211014)

[Вывод 16](#_Toc162211015)

1. **Общая постановка задачи**

***1. Реализовать структуру данных бинарное дерево поиска. Для этого создать шаблон класса BinarySearchTree.***

*1.1 Для описания узла дерева используйте тип Node, в котором поля:*

key\_ - значение ключа узла,

left\_ - указатель на левое поддерево,

right\_ - указатель на правое поддерево,

p\_ - указатель на родителя (может не использоваться).

Тип Node может использоваться только в классе BinarySearchTree.

Класс BinarySearchTree должен содержать поле root\_ - указатель на корневой узел.

*1.2 В классе должны быть:*

•конструктор по умолчанию, создающий пустое дерево,

•конструктор перемещения,

•оператор перемещающего присваивания,

•деструктор.

Конструктор копирования и оператор присваивания (с копированием) должны быть запрещены.

*1.3 В классе должны быть методы:*

1.поиска по ключу (итеративный)

2.вставки нового элемента в дерево (итеративный)

3.удаления элемента из дерева (итеративный)

4.вывода строкового изображения дерева

5.определения количества узлов дерева (рекурсивный)

6.определения высоты дерева (рекурсивный)

7.инфиксного обхода дерева (итеративный)

8.инфиксного обхода дерева (рекурсивный)

9.итеративный метод обхода двоичного дерева по уровням (в ширину). В реализации использовать класс очередь

10.определения, являются ли два дерева похожими. Похожими будем называть деревья поиска, содержащие одинаковые наборы ключей. Рекомендация: параллельно обходить инфиксным обходом сравниваемые деревья

11.определения, есть одинаковые ключи в двух деревьях поиска. Рекомендация: параллельно обходить инфиксным обходом сравниваемые деревья

Два набора методов:

1. private для работы с узлами – для разработчика класса

2. public работы со значениями (ключами) – для пользователя

Методы могут быть перегруженными, т. е. можно использовать одно и то же имя для private и public методов.

***2. Пример реализации метода определения количества узлов***

// private: Рекурсивная функция определения количества узлов дерева

size\_t getNumberOfNodes (Node \*node) const

{if (node == nullptr) {return 0;}

return (1 + getNumberOfNodes(node->left\_) +

getNumberOfNodes(node->right\_));

}

//// public: Определение количества узлов дерева

size\_t getNumberOfNodes() const

{return getNumberOfNodes (this->root);};

***3.Написать функции для тестирования методов.***

***4.В реализации использовать приведенные ниже заготовки:***

Файл BinarySearchTree.h

#ifndef \_BINARY\_SEARCH\_TREE\_H

#define \_BINARY\_SEARCH\_TREE\_H template <class T>

class BinarySearchTree

{

public:

BinarySearchTree();// "по умолчанию" создает пустое дерево

BinarySearchTree(const BinarySearchTree<T> & scr) = delete;

BinarySearchTree(BinarySearchTree<T>&& scr) noexcept;

BinarySearchTree <T>& operator= (const BinarySearchTree <T>& src) = delete;

BinarySearchTree <T>& operator= (BinarySearchTree <T>&& src) noexcept;

virtual ~BinarySearchTree();

// 1.1 Функция поиска по ключу в бинарном дереве поиска

bool searchKeyIterative (const T& key) const;

// 2 Вставка нового элемента в дерево: true, если элемент добавлен;

// false, если элемент уже был

bool insertNode(const T& key);

// 3.1 Удаление элемента c заданным ключом, не нарушающее порядок элементов

// true, если элемент удален; false, если элемента не было

bool deleteNode(const T& key);

// 4.1 Вывод структуры (строкового изображения дерева) в выходной поток out,

// использовать скобки, чтобы показать структуру дерева

void output(std::ostream& out) const;

// 5.1 Определение количества узлов дерева

int getNumberOfNodes () const;

// 6.1 Определение высоты дерева

int getHeight() const;

// 7 Инфиксный обход дерева (итеративный)

void inorderWalkIterative () const;

// 8.1 Инфиксный обход дерева (рекурсивный)

void inorderWalk() const;

// 9 Обход двоичного дерева по уровням (в ширину).

void walkByLevels() const;

// 10 Являются ли два дерева похожими

bool isSimilar(const BinarySearchTree<T> & other) const;

// 11 Есть одинаковые ключи в двух деревьях поиска

bool isIdenticalKey(const BinarySearchTree<T> & other) const;

private:

// 1.2 Функция поиска адреса узла по ключу в бинарном дереве поиска

Node<T>\* searchNodeIterative (const T& key) const;

// 4.2 Рекурсивная функция для вывода структуры дерева в выходной поток

void output(std::ostream& out, Node<T>\* root) const;

// 5.2 Рекурсивная функция определения количества узлов дерева

int getNumberOfNodes(const Node<T>\* node) const;

// 6.2 Рекурсивная функция определения высоты дерева

int getHeight(const Node<T>\* node) const;

// 8.2 Рекурсивная функция для инфиксного обхода узлов дерева.

void inorderWalk(Node<T>\* node) const; template <class T>

struct Node {

T key\_; // значение ключа, содержащееся в узле

Node<T> \*left\_; // указатель на левое поддерево

Node<T> \*right\_; // указатель на правое поддерево

Node<T> \*p\_; // указатель на родителя !!! не используется

// Конструктор узла

Node(T key, Node \*left = nullptr, Node \*right= nullptr,Node \*p =nullptr):

key\_(key), left\_ (left), right\_(right), p\_(p)

{ }

};

Node<T> \*root\_; // Указатель на корневой узел

}; // конец шаблона класса BinarySearchTree

1. **Детальные требования и тест план**

## *Детальные требования:*

Отсутствуют

## *Таблица с детальными требованиями и тест планом*

|  |  |  |  |
| --- | --- | --- | --- |
| Требования | Детальные требования | Данные | Ожидаемый результат |
| - | - | Все данные заданы корректно | Вывод результата работы всех методов |

1. **Программа**

## *Main.cpp*

#include <iostream>

#include"BinarySearchTree.h"

int main()

{

setlocale(LC\_ALL , "ru");

BinarySearchTree<int> tr;

tr.output(std::cout);

tr.insertNode(5);

tr.insertNode(3);

tr.insertNode(8);

tr.insertNode(1);

tr.insertNode(4);

tr.insertNode(12);

tr.output(std::cout);

tr.searchKeyIterative(10) ? std::cout << "Элемент в коллекции\n" : std::cout << "Элемент отсутствует\n";

if (tr.deleteNode(3))

{

std::cout << "Элемент 3 удален:\t";

tr.output(std::cout);

}

else std::cout << "Элемент отсутствует\n";

std::cout << tr.getNumberOfNodes() << "\tузлов\n";

std::cout << "Высота дерева:\t" << tr.getHeight()<<"\n";

std::cout << "Инфиксный обход дерева(итеративный): ";

tr.inorderWalkIterative();

std::cout << "Инфиксный обход дерева(рекурсивный): ";

tr.inorderWalk();

std::cout << "Обход дерева в ширину: ";

tr.walkByLevels();

BinarySearchTree<int> tr1 ;

tr1.insertNode(rand()%30);

tr1.insertNode(rand() % 30);

tr1.insertNode(rand() % 30);

tr1.insertNode(rand() % 30);

tr1.insertNode(rand() % 30);

std::cout << "\n\nСтарое дерево:\t";

tr.output(std::cout);

std::cout << "Новое дерево:\t";

tr1.output(std::cout);

tr.isSimilar(tr1)? std::cout << "Деревья совпадают\n":std::cout << "Деревья не совпадают\n";

tr.isIdenticalKey(tr1)? std::cout << "Есть одинаковые ключи\n" : std::cout << "Одинаковые ключи отсутсвуют\n";

return EXIT\_SUCCESS;

}

## *BinarySearchTree.h*

#ifndef BINARYSEARCHTREE\_H

#define BINARYSEARCHTREE\_H

#include <sstream>

#include"Structs.h"

template<typename T>

class BinarySearchTree

{

public:

BinarySearchTree(): root(nullptr) {};

BinarySearchTree(const BinarySearchTree& scr) = delete;

BinarySearchTree( BinarySearchTree&& other) noexcept

{

root = other.root;

other.root = nullptr;

}

BinarySearchTree& operator = (const BinarySearchTree& scr) = delete;

BinarySearchTree& operator = (const BinarySearchTree&& other) noexcept

{

root = other.root;

other.root = nullptr;

return \*this;

}

virtual ~BinarySearchTree() { deleteNodes(root); }

bool searchKeyIterative(const T& key) const

{

Node<T>\* ptr = root;

if (!root)

{

std::cerr << EMPTY\_COLLECTION;

return false;

}

else

while (ptr != nullptr && ptr->key\_ != key)

{

if (key < ptr->key\_) ptr = ptr->left\_;

else ptr = ptr->right\_;

}

return (ptr != nullptr && ptr->key\_ == key) ? true : false;

}

bool insertNode(const T& key)

{

Node<T>\* new\_element = new Node<T>(key);

if (!root)

{

root = new\_element;

return true;

}

else

{

Node<T>\* ptr = root;

while (ptr)

{

Node<T>\* p\_ptr = ptr;

if (key < ptr->key\_)

{

if (!ptr->left\_)

{

ptr->left\_ = new\_element;

new\_element->p\_ = ptr;

return true;

}

ptr = ptr->left\_;

}

else if (key > ptr->key\_)

{

if (!ptr->right\_)

{

ptr->right\_ = new\_element;

new\_element->p\_ = ptr;

return true;

}

ptr = ptr->right\_;

}

else

{

delete new\_element;

return false;

}

}

}

}

void output(std::ostream& out) const

{

if (!root) std::cerr << EMPTY\_COLLECTION;

else

{

output(out, root);

out << "\n";

}

}

bool deleteNode(const T& key)

{

if(!searchNodeIterative(key)) return false;

else

{

Node<T>\* ptr = searchNodeIterative(key);

if (!ptr->left\_ && !ptr->right\_)

{

if (ptr->p\_->left\_ && ptr->p\_->left\_->key\_ == key) ptr->p\_->left\_ = nullptr;

else if (ptr->p\_->right\_ && ptr->p\_->right\_->key\_ == key) ptr->p\_->right\_ = nullptr;

delete ptr;

}

if (ptr->left\_ && !ptr->right\_)

{

if (ptr->p\_->left\_ && ptr->p\_->left\_->key\_ == key) ptr->p\_->left\_ = ptr->left\_;

else if (ptr->p\_->right\_ && ptr->p\_->right\_->key\_ == key) ptr->p\_->right\_ = ptr->left\_;

ptr->left\_->p\_ = ptr->p\_;

delete ptr;

}

if (!ptr->left\_ && ptr->right\_)

{

if (ptr->p\_->left\_ && ptr->p\_->left\_->key\_ == key) ptr->p\_->left\_ = ptr->right\_;

else if (ptr->p\_->right\_ && ptr->p\_->right\_->key\_ == key) ptr->p\_->right\_ = ptr->right\_;

ptr->right\_->p\_ = ptr->p\_;

delete ptr;

}

if (ptr->left\_ && ptr->right\_)

{

Node<T>\* min = ptr->right\_;

while (min->left\_) min = min->left\_;

if (min->p\_->left\_ && min->p\_->left\_->key\_ == min->key\_) min->p\_->left\_ = nullptr;

else if (min->p\_->right\_ && min->p\_->right\_->key\_ == min->key\_) min->p\_->right\_ = nullptr;

ptr->key\_ = min->key\_;

delete min;

}

return true;

}

}

int getNumberOfNodes() const { return getNumberOfNodes(root); }

int getHeight() const { return getHeight(root); }

void inorderWalkIterative() const

{

if (root)

{

Stack<Node<T>\*> stack;

Node<T>\* ptr = root;

while (ptr || !stack.isEmpty())

{

while (ptr)

{

stack.push(ptr);

ptr = ptr->left\_;

}

ptr = stack.getTop();

stack.pop();

std::cout << ptr->key\_ << " ";

ptr = ptr->right\_;

}

std::cout << "\n";

}

else std::cerr << EMPTY\_COLLECTION;

}

void inorderWalk() const

{

if (!root) std::cerr << EMPTY\_COLLECTION;

else

{

inorderWalk(root);

std::cout << "\n";

}

}

void walkByLevels() const

{

if (root)

{

Queue<Node<T>\*> queue;

queue.push(root);

Node<T>\* ptr = nullptr;

while (!queue.isEmpty())

{

ptr = queue.getFront();

std::cout << ptr->key\_ << " ";

queue.pop();

if (ptr->left\_) queue.push(ptr->left\_);

if (ptr->right\_) queue.push(ptr->right\_);

}

std::cout << std::endl;

}

else std::cerr << EMPTY\_COLLECTION;

}

bool isSimilar(const BinarySearchTree<T>& other) const

{

std::ostringstream ostr1;

inorderWalk(ostr1, root);

std::ostringstream ostr2;

inorderWalk(ostr2, other.root);

return ostr1.str() == ostr2.str() ? true : false;

}

bool isIdenticalKey(const BinarySearchTree<T>& other) const

{

std::ostringstream ostr1;

inorderWalk(ostr1, root);

std::ostringstream ostr2;

inorderWalk(ostr2, other.root);

for (char key2 : ostr2.str())

for (char key1 : ostr1.str())

if (key1 == key2) return true;

return false;

}

private:

template<typename T>

struct Node

{

T key\_;

Node<T>\* left\_;

Node<T>\* right\_;

Node<T>\* p\_;

Node(T key, Node<T>\* left = nullptr, Node<T>\* right = nullptr, Node<T>\* p = nullptr) :

key\_(key), left\_(left), right\_(right), p\_(p) {};

Node(const Node<T>&& other) noexcept

{

left\_ = other.left\_;

other.left\_ = nullptr;

right\_ = other.rigth\_;

other.rigth\_ = nullptr;

key\_ = other.key\_;

p\_ = other.p\_;

other.p\_ = nullptr;

}

};

Node<T> \*root;

Node<T>\* searchNodeIterative(const T& key) const

{

if (!root) return nullptr;

else

{

Node<T>\* ptr = root;

while (ptr && ptr->key\_ != key)

{

if (key < ptr->key\_) ptr = ptr->left\_;

else ptr = ptr->right\_;

}

return ptr;

}

}

void output(std::ostream& out, Node<T>\* ptr) const

{

if (!ptr) return;

out << ptr->key\_;

if (ptr->left\_ != nullptr || ptr->right\_ != nullptr)

{

out << " (";

output(out, ptr->left\_);

if (ptr->left\_ && ptr->right\_) out << ", ";

output(out, ptr->right\_);

out << ")";

}

}

int getNumberOfNodes(const Node<T>\* node) const

{

if (!node) return 0;

else return 1 + getNumberOfNodes(node->right\_) + getNumberOfNodes(node->left\_);

}

int getHeight(const Node<T>\* node) const

{

if (!node) return 0;

else return 1 + std::max(getHeight(node->left\_), getHeight(node->right\_));

}

void inorderWalk(Node<T>\* node) const

{

if (!node) return;

else

{

inorderWalk(node->left\_);

std::cout << node->key\_ << " ";

inorderWalk(node->right\_);

}

}

void inorderWalk(std::ostringstream &oss,Node<T>\* node) const

{

if (!node) return;

else

{

inorderWalk(oss,node->left\_);

oss<<node->key\_<<" ";

inorderWalk(oss,node->right\_);

}

}

void deleteNodes(Node<T>\* node)

{

if (!node) return;

else

{

deleteNodes(node->left\_);

deleteNodes(node->right\_);

delete node;

}

}

};

#endif

## *Structs.h*

#ifndef STRUCTS\_H

#define STRUCTS\_H

const std::string EMPTY\_COLLECTION = "Элементы отсутствуют\n";

template <typename T>

struct Node

{

T key\_;

Node<T>\* next\_;

Node(T key, Node<T>\* next = nullptr) : key\_(key), next\_(next) {};

};

template <class T>

class Queue

{

friend struct Node<T>;

public:

Queue(): head(nullptr) {};

~Queue()

{

if (head)

{

Node<T>\* previous = head;

Node<T>\* ptr = previous->next\_;

while (ptr)

{

delete previous;

previous = ptr;

ptr = ptr->next\_;

}

delete previous;

}

}

bool isEmpty() const { return !head ? true : false; }

void push(const T key)

{

Node<T>\* new\_el = new Node<T>(key);

if (!head) head = new\_el;

else

{

Node<T>\* ptr = head;

while (ptr->next\_) ptr = ptr->next\_;

ptr->next\_ = new\_el;

}

}

T getFront()

{

if (head) return head->key\_;

else std::cerr << EMPTY\_COLLECTION;

};

void pop()

{

Node<T>\* ptr = head;

if (head) head = head->next\_;

else std::cerr << EMPTY\_COLLECTION;

delete ptr;

}

private:

Node<T>\* head;

};

template <typename T>

class Stack

{

friend struct Node<T>;

public:

Stack() : top(nullptr) {}

~Stack()

{

if (top)

{

Node<T>\* previous = top;

Node<T>\* ptr = top->next\_;

while (ptr)

{

delete previous;

previous = ptr;

ptr = ptr->next\_;

}

delete previous;

}

}

bool isEmpty() const { return !top ? true : false; }

void push(const T key)

{

Node<T>\* new\_el = new Node<T>(key);

if (!top) top = new\_el;

else

{

Node<T>\* ptr = top;

while (ptr->next\_) ptr = ptr->next\_;

ptr->next\_ = new\_el;

}

}

T getTop() const

{

if (top)

{

Node<T>\* ptr = top;

while (ptr->next\_) ptr = ptr->next\_;

return ptr->key\_;

}

else std::cerr << EMPTY\_COLLECTION;

}

void pop()

{

if (top)

{

Node<T>\* previous = top;

if (previous->next\_)

{

Node<T>\* ptr = top->next\_;

while (ptr->next\_)

{

previous = previous->next\_;

ptr = ptr->next\_;

}

previous->next\_ = nullptr;

delete ptr;

}

else

{

top = nullptr;

delete previous;

}

}

else std::cerr << EMPTY\_COLLECTION;

}

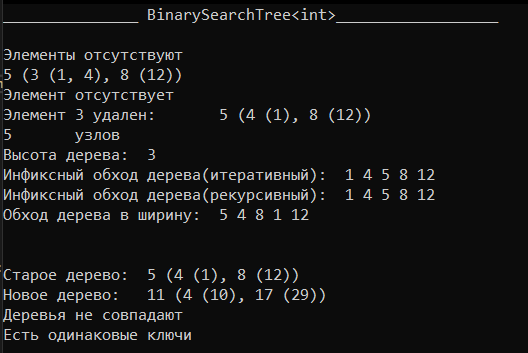
private:

Node<T>\* top;

};

#endif

# Приложение



*Рис. 1 Работа программы*

# Вывод

В ходе работы над программой были изучены:

1. Структура данных бинарное дерево поиска, написан шаблон данного класса, реализованы методы данного класса.
2. Написаны классы стек и очередь.