ПМИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: БДП

Студент гр. 8381 _____ Сахаров В.М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Ознакомиться с основными характеристиками и особенностями такой структуры данных, как БДП, изучить особенности ее реализации на языке программирования C++. Разработать программу, которая строит изображение БДП и вставляет в дерево заданный элемент.

Задание.

1) По заданному файлу F (типа file of Elem), все элементы которого различны, построить структуру данных определённого типа – БДП;

2) а) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если не входит, то добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Вставка и исключение элементов - Демонстрация

Основные теоретические положения.

Идеальное бинарное дерево поиска называется АВЛ-деревом: сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Выполнение работы.

Написание работы производилось на базе операционной системы Windows 10 в среде QTCreator.

Сначала происходило считывание введенных пользователем данных и проверка на режим работы программы. При выполнении в консоли (указан аргумент «-c») запускается считывание аргументов командной строки и сама сортировка с выводом результатов. При выполнении с графически запускается окно mainwindow, в котором даётся выбор между ручным вводом и загрузки дерева из файла.

Возможные режимы работы программы:

Вставка элемента, использует функцию tree.insertbalanced, восстанавливая условие АВЛ-дерева

Пошаговая вставка элемента, использующая машину состояний для аналогичного поведения

Удаление элемента, использует функцию tree.removebalanced, восстанавливая условие АВЛ-дерева

Пошаговое удаление элемента, использующая машину состояний для аналогичного поведения

Оценка эффективности алгоритма

Алгоритм создания БДП по строке является итеративным, каждый элемент строки обрабатывается один раз, а значит сложность алгоритма можно оценить как O(N).

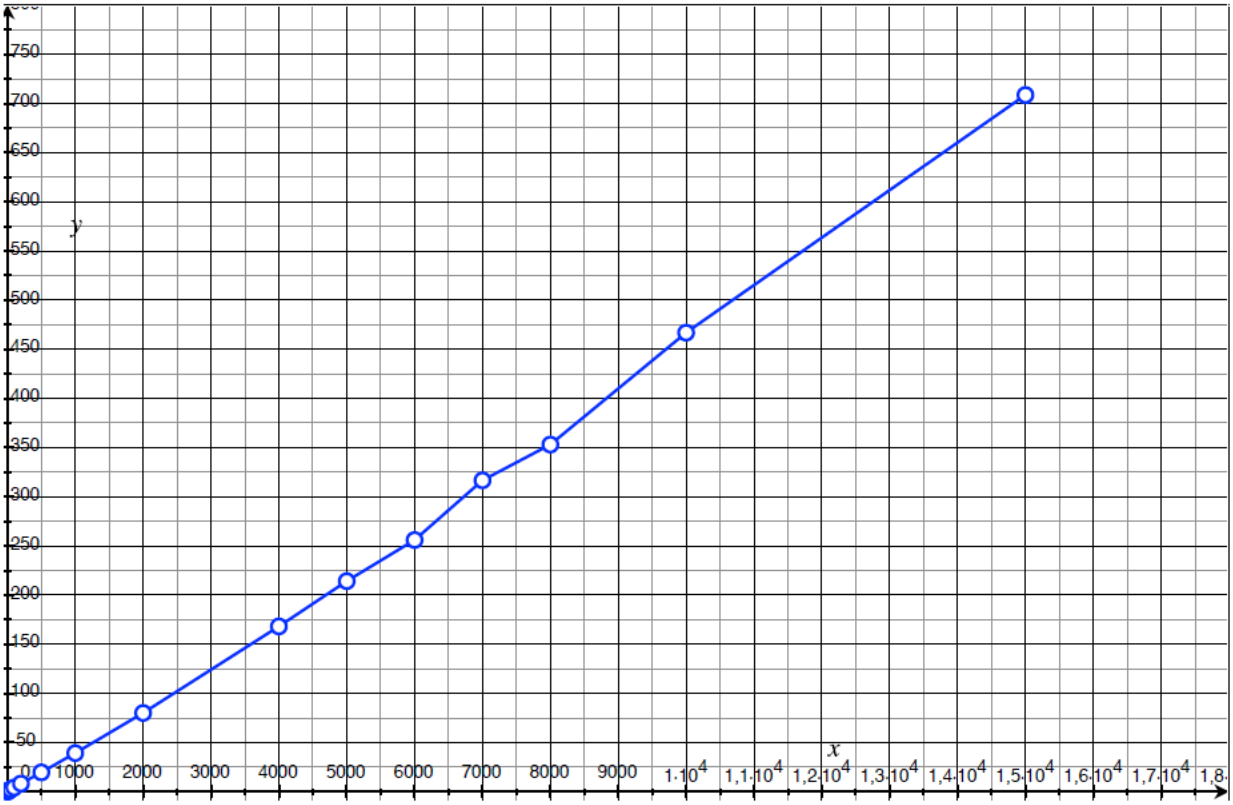


График 1 — Зависимость количества элементов к времени строительства

Алгоритмы нахождения и удаления заданного элемента является рекурсивным, каждый узел дерева обрабатывается один раз, следовательно, сложность алгоритма для бинарного дерева – $O(\log_2 n)$.

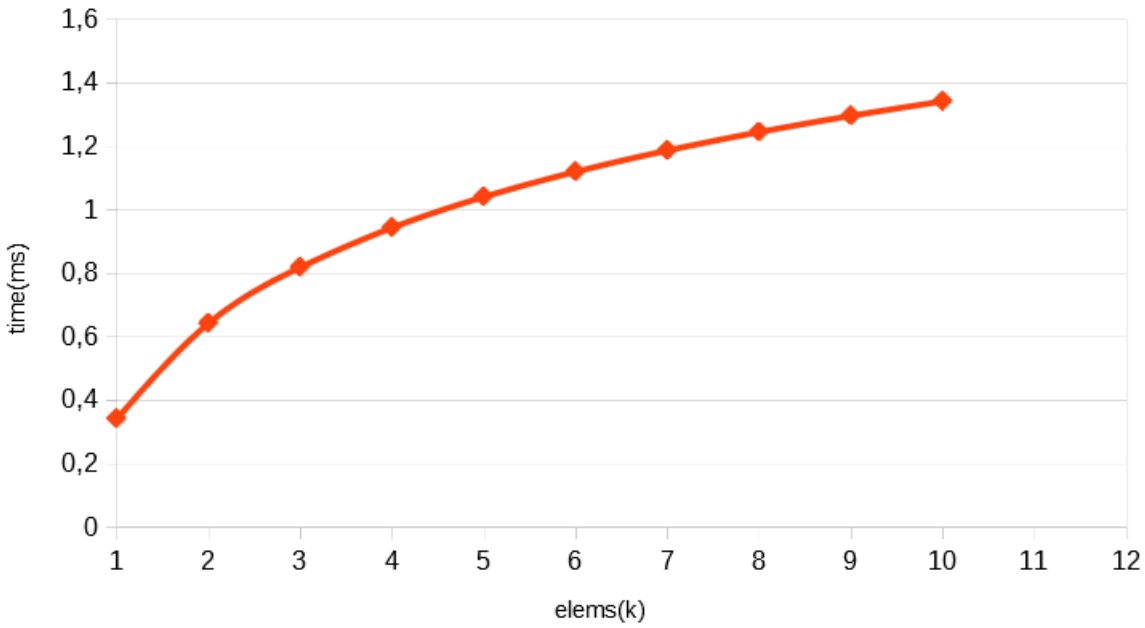


График 1 — Зависимость времени операций от количества элементов

Тестирование программы.

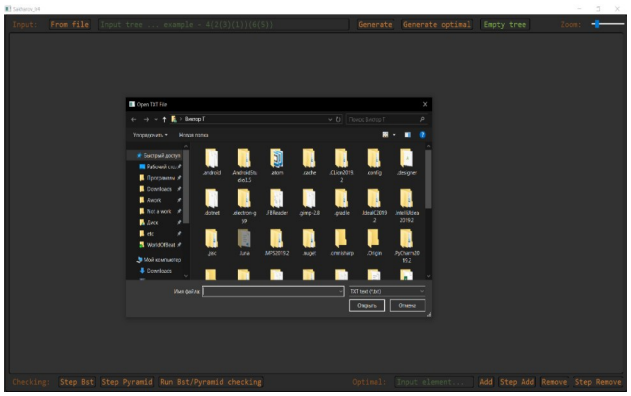
Консольный режим:

Таблица 1 — результаты работы консольного режима программы

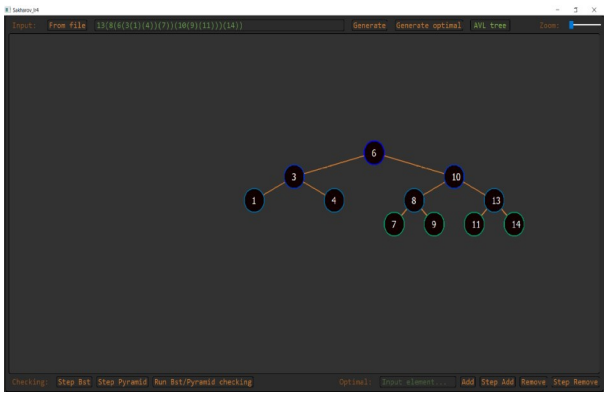
```
Starting in console mode...
Tree found: 9(8(6(3(1)(2))(5))(5(3)(1)))(7))
Tree: {9(8(6(3(1(^,^),2(^,^)),5(^,^)),5(3(^,^),1(^,^))),7(^,^))}
Insert: 15
Tree: {9(8(6(3(1(^,^),2(^,^)),5(^,^)),5(3(^,^),1(^,^))),7(^,15(^,^)))}
Insert: 4
Tree:
{8(2(3(1(^,^),^),6(4(^,^),5(^,^))),9(5(3(^,^),1(^,^)),7(^,15(^,^))))}
Remove: 8
Tree:
{8(2(3(1(^,^),^),6(4(^,^),5(^,^))),9(5(3(^,^),1(^,^)),7(^,15(^,^))))}
```

```
Remove: 3
Tree:
{8(2(3(1(^,^),^),6(4(^,^),5(^,^))),9(5(3(^,^),1(^,^)),7(^,15(^,^))))}
```

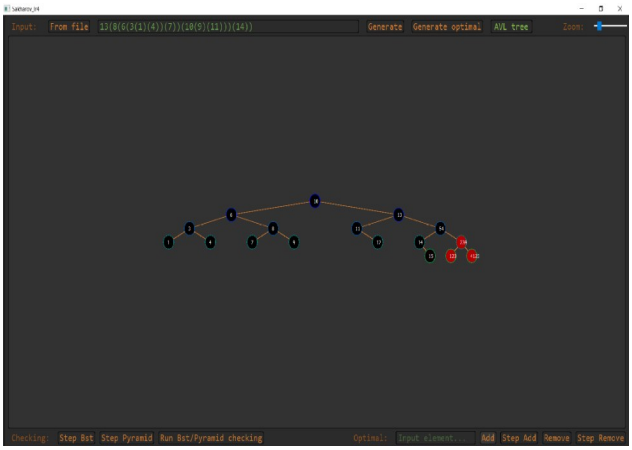
Графический интерфейс:



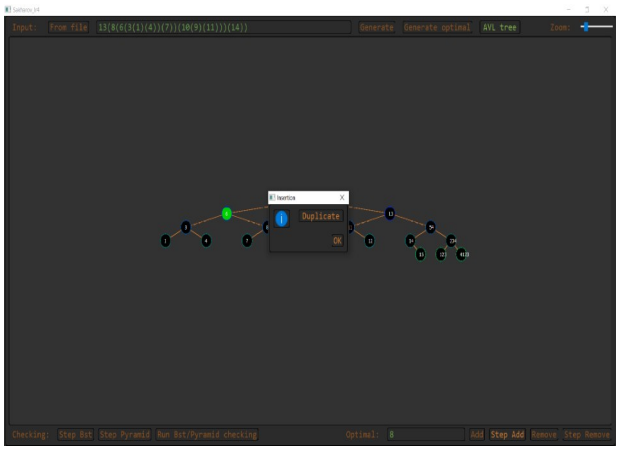
А)



Б)



В)



Г)

Скриншоты ввода через файл (Рис. А), сгенерированного дерева (Рис. Б), Процесса пошаговой работы алгоритма (Рис. В), результата алгоритма (Рис. Г):

Выводы.

В ходе выполнения лабораторной работы была разработана программа, осуществляющая считывание АВЛ-дерева, выбранным пользователем способом, и выводящая его графическое представление. Также в программе реализован функционал визуализации вставки и удаления элементов из полученного дерева.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include "mainwindow.h"
#include "utils_cli.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    for(int i = 0; i < argc; i++) {
        if(!strcmp("console", argv[i]) || !strcmp("-console", argv[i]) || !str-
cmp("-c", argv[i])) {
            return utils_cli::execute(argc - 1, argv + 1);
        }
    }
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Файл mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "utils_headers.h"
#include "utils_linked.h"
#include "utils_vector.h"
#include "utils_tree.h"
#include <cmath>

enum mode
{
    empty,
    bst,
    pyramid,
    step_add,
    step_remove,
};

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
    void UpdateGraphics();

private slots:
    void on_butFile_clicked();
```

```
    void on_butGenerate_clicked();

    void on_butRun_clicked();

    void on_horizontalSlider_sliderMoved(int position);

    void on_butStepBst_clicked();

    void on_butStepPyramid_clicked();

    void on_butAdd_clicked();

    void on_butStepAdd_clicked();

    void on_butRemove_clicked();

    void on_butGenerateOptimal_clicked();

    void on_butStepRemove_clicked();

    void updateTreeColor();

private:
    utils_tree<int>* tree;
    int current;
    bool is_bst;
    bool is_pyramid;
    utils_vector<node<int>*> bst_stepped;
    utils_vector<node<int>*> pyramid_stepped;
    utils_linked<node<int>*> stack;
    node<int>* state_node;
    int state;
    bool locked;
    bool lockedUpd;
    mode stepped_mode;

    Ui::MainWindow *ui;

    QGraphicsScene *mainGraphicsScene;
    QPen pen;
    QColor color;
    QBrush brush;
    QFont font;

    void DrawNode(node<int>* n, int maxdepth, int depth = 0, int x = 0, int y =
0);
    void SetActiveButtons(bool mode);
    bool ReadElement();
};

#endif // MAINWINDOW_H
```

Файл ilist.h:

```
#ifndef ILIST_H
#define ILIST_H

template <class T>
struct IList {
    virtual T operator[] (int index) = 0;
    virtual T at (int index) = 0;
    virtual void clean() = 0;
```

```
    virtual void insert(int index, T element) = 0;
    virtual T remove(int index) = 0;

    virtual T back() = 0;
    virtual void push_back(T element) = 0;
    virtual T pop_back() = 0;

    virtual T front() = 0;
    virtual void push_front(T element) = 0;
    virtual T pop_front() = 0;

    virtual int size() = 0;
    virtual bool empty() = 0;
    virtual ~IList(){}
};

#endif // ILIST_H
```

Файл utils_linked.h:

```
#ifndef UTILS_LINKED_H
#define UTILS_LINKED_H
#include "ilist.h"

template <class T = int>
struct node
{
    node* right;
    T data;

    int mode;

    node* left;

    node(T d = 0)
    {
        right = nullptr;
        data = d;
        mode = 0;
    }
};

template <class T = int>
class utils_linked : public IList<T>
{
private:
    node<T>* head;
    node<T>* tail;

public:
    utils_linked();
    utils_linked(const utils_linked& copy);
    T operator[] (int index) override;
    T at (int index) override;
    void clean() override;
    void insert(int index, T element) override;
    T remove(int index) override;

    T back() override;
    void push_back(T element) override;
    T pop_back() override;

    T front() override;
```

```
    void push_front(T element) override;
    T pop_front() override;

    int size() override;
    bool empty() override;
    ~utils_linked();
};


template<class T>
utils_linked<T>::utils_linked()
{
    head = nullptr;
    tail = nullptr;
}

template<class T>
utils_linked<T>::utils_linked(const utils_linked & copy)
{
//    head = nullptr;
//    tail = nullptr;
//    node<int>* copy_node = copy.head;
//    while (copy_node)
//    {
//        node<int>* t = new node<int>(copy_node->data);
//        head->prev = tail;
//        tail->next = head;
//        t->data
//    }
//    array = new T[capacity];
//    for (int i = 0; i < count; ++i)
//    {
//        *(array + i) = *(copy.__arr + i);
//    }
}


template <class T>
T utils_linked<T>::operator[] (int index)
{
    node<T>* t = head;
    for (int i = 0; i < index; i++)
    {
        t = t->right;
    }
    return t->data;
}

template <class T>
T utils_linked<T>::at (int index)
{
    return operator[](index);
}

template <class T>
void utils_linked<T>::clean ()
{
    node<T>* t = head;
    while (t)
    {
        delete t;
        t = t->right;
```

```
    }
    head = nullptr;
    tail = nullptr;
}

template <class T>
void utils_linked<T>::insert(int index, T element)
{
    node<T>* n = new node<T>(element);
    if (empty())
    {
        head = n;
        tail = n;
    }
    else if (index == 0)
    {
        push_front(element);
    }
    else if (index == size())
    {
        push_back(element);
    }
    else
    {
        node<T>* t = head;
        for (int i = 0; i < index; i++)
        {
            t = t->right;
        }
        n->right = t;
        n->left = t->left;
        t->left->right= n;
        t->left = n;
    }
}

template<class T>
T utils_linked<T>::remove(int index)
{
    T res = at(index);
    if (index == 0)
    {
        pop_front();
    }
    else if (index == size() - 1)
    {
        pop_back();
    }
    else {
        node<T> *t = head;
        for (int i = 0; i < index; i++) {
            t = t->right;
        }
        t->left->right = t->right;
        t->right->left = t->left;
        delete t;
    }
    return res;
}

template<class T>
T utils_linked<T>::back()
```

```
{
    return tail->data;
}

template<class T>
void utils_linked<T>::push_back(T element)
{
    node<T>* n = new node<T>(element);
    if (empty())
    {
        head = n;
        tail = n;
    }
    else
    {
        tail->right = n;
        n->left = tail;
        tail = n;
    }
}

template<class T>
T utils_linked<T>::pop_back()
{
    T data;
    if (size() == 1) {
        if (head != nullptr) {
            data = head->data;
            delete head;
            head = nullptr;
        } else if (tail != nullptr) {
            data = tail->data;
            delete tail;
            tail == nullptr;
        }
    }
    else {
        node<T> *n = tail;
        tail = tail->left;
        tail->right = nullptr;
        data = n->data;
        delete n;
    }
    return data;
}

template<class T>
T utils_linked<T>::front()
{
    return head->data;
}

template<class T>
void utils_linked<T>::push_front(T element)
{
    node<T>* n = new node<T>(element);
    if (empty())
    {
        head = n;
        tail = n;
    }
    else
```

```
    {
        head->left = n;
        n->right = head;
        head = n;
    }
}

template<class T>
T utils_linked<T>::pop_front()
{
    T data;
    if (size() == 1) {
        if (head != nullptr) {
            data = head->data;
            delete head;
            head = nullptr;
        } else if (tail != nullptr) {
            data = tail->data;
            delete tail;
            tail == nullptr;
        }
    }
    else {
        node<T> *n = head;
        head = head->right;
        head->left = nullptr;
        data = n->data;
        delete n;
    }
    return data;
}


template<class T>
int utils_linked<T>::size()
{
    int i = 0;
    node<T>* t = head;
    while (t)
    {
        t = t->right;
        i++;
    }
    return i;
}

template<class T>
bool utils_linked<T>::empty()
{
    return !size();
}

template<class T>
utils_linked<T>::~utils_linked()
{
    node<T>* t = head;
    while (t)
    {
        delete t;
        t = t->right;
    }
}
```

```
#endif // UTILS_LINKED_H
```

Файл utils_vector.h:

```
#ifndef UTILS_VECTOR_H
#define UTILS_VECTOR_H
#include "ilist.h"

template <class T = int>
class utils_vector : public IList<T>
{
private:
    T* array;
    int capacity;
    int count;
    void resize(int new_capacity);

public:
    utils_vector(int start_capacity = 4);
    utils_vector(const utils_vector& copy);
    T operator[] (int index) override;
    T at (int index) override;
    void clean() override;
    void insert(int index, T element) override;
    T remove(int index) override;

    T back() override;
    void push_back(T element) override;
    T pop_back() override;

    T front() override;
    void push_front(T element) override;
    T pop_front() override;

    int size() override;
    bool empty() override;
    ~utils_vector();
};

template<class T>
void utils_vector<T>::resize(int new_capacity)
{
    auto *arr = new T[count];
    for (int i = 0; i < count; ++i)
    {
        arr[i] = array[i];
    }
    delete [] array;
    array = new T[new_capacity];
    for (int i = 0 ; i < count; ++i)
    {
        array[i] = arr[i];
    }
    delete [] arr;
    capacity = new_capacity;
}

template<class T>
utils_vector<T>::utils_vector(int start_capacity)
{
    capacity = start_capacity;
    count = 0;
```

```
    array = new T[capacity];
}

template<class T>
utils_vector<T>::utils_vector(const utils_vector & copy) :
    count(copy.size),
    capacity(copy.capacity)
{
    array = new T[capacity];
    for (int i = 0; i < count; ++i)
    {
        *(array + i) = *(copy.__arr + i);
    }
}

template <class T>
T utils_vector<T>::operator[] (int index)
{
    return array[index];
}

template <class T>
T utils_vector<T>::at (int index)
{
    return operator[](index);
}

template <class T>
void utils_vector<T>::clean ()
{
    count = 0;
}

template <class T>
void utils_vector<T>::insert(int index, T element)
{
    if (capacity == count)
    {
        resize(count + 8);
    }
    if (count > 0) {
        for (int i = count; i > index; i--)
        {
            array[i] = array[i - 1];
        }
    }
    count++;
    array[index] = element;
}

template<class T>
T utils_vector<T>::remove(int index)
{
    auto temp = array[index];
    for (int i = index; i < count - 1; i++)
    {
        array[i] = array[i + 1];
    }
    count--;
    return temp;
}
```

```
template<class T>
T utils_vector<T>::back()
{
    return array[count - 1];
}

template<class T>
void utils_vector<T>::push_back(T element)
{
    if (capacity == count)
    {
        resize(count + 8);
    }
    array[count] = element;
    count++;
}

template<class T>
T utils_vector<T>::pop_back()
{
    return array[--count];
}

template<class T>
T utils_vector<T>::front()
{
    return *array;
}

template<class T>
void utils_vector<T>::push_front(T element)
{
    insert(0, element);
}

template<class T>
T utils_vector<T>::pop_front()
{
    return remove(0);
}

template<class T>
int utils_vector<T>::size()
{
    return count;
}

template<class T>
bool utils_vector<T>::empty()
{
    return !count;
}

template<class T>
utils_vector<T>::~utils_vector()
{
    delete [] array;
}

#endif //VECTOR_VECTOR_H
```

Файл utils_cli.h:

```
#ifndef UTILS_CLI_H
#define UTILS_CLI_H
#include <string>
#include <iostream>
#include "utils_headers.h"
#include "utils_tree.h"

class utils_cli
{
public:
    static int execute(int argc, char *argv[]);
private:
    utils_cli(){}
};

#endif // UTILS_CLI_H
```

Файл utils_headers.h:

```
#ifndef UTILS_HEADERS_H
#define UTILS_HEADERS_H

#include <iostream>
#include <vector>
#include <map>
#include <fstream>
#include <algorithm>
#include <memory>
#include <cstdint>
#include <cstring>
#include <string>
#include <cstdlib>
#include <unistd.h>
#include <exception>
#include <stdexcept>
#include <cstdio>
#include <cassert>
#include <regex>
#include <experimental/filesystem>
#include <cmath>
#include <unistd.h>
#include <iostream>
#include <algorithm>
#include <string>

#include <QObject>
#include <QMessageBox>
#include <QDebug>
#include <QString>
#include <QFileDialog>
#include <QGraphicsItem>
#include <QtGui>
#include <QDialog>
#include <QColorDialog>
#include <QString>
#include <QDebug>
#include <QPainter>
#include <QComboBox>
#include <QLabel>
#include <QPushButton>
#include <QFile>
#include <QWidget>
#include <QVBoxLayout>
```

```
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QGroupBox>
#include <QRadioButton>
#include <QTextEdit>
#include <QEventLoop>
#include <QTimer>
#include <QColor>
#include <QDebug>
#include <QGraphicsView>
#include <QFormLayout>

#endif // UTILS_HEADERS_H
```

Файл utils_tree.h:

```
#ifndef UTILS_TREE_H
#define UTILS_TREE_H
#include <string>
#include "utils_linked.h"
#include "utils_vector.h"

template <class T = int>
class utils_tree
{
public:
    utils_tree(std::string& str);
    void clean();
    void insert(T data);
    void remove(T data);
    bool search(T key);
    bool is_bst();
    bool is_pyramid();
    int max_depth();
    ~utils_tree();
    node<T>* root;
    node<T>* previous;
    bool is_bst_stepped(utils_vector<node<T>*>& v, node<T>* n, T min, T max);
    bool is_pyramid_stepped(utils_vector<node<T>*>& v, node<T>* n, int max);
    void to_optimal();
    void insertbalanced(T data);
    void removebalanced(T data);
    void go_darker();
    std::string node_string();
private:
    bool parse_tree(node<T>*& n, std::string &s, int &i);
    void clean(node<T>* n);
    void insert(node<T>*& n, T data);
    void remove(node<T>*& n, T data);
    node<T>* search(node<T>* n, T key);
    bool is_bst(node<T>* n, T min, T max);
    bool is_pyramid(node<T>* n, int max);
    int max_depth(node<T>* n, int i);
    void get_data(utils_vector<T>& vec, node<T>* n);
    void go_darker(node<T>* n);
    std::string node_string(node<T>* n);
    //
public:
    int height(node<T>* n);
    int bfactor(node<T>* n);
    void fixheight(node<T>* n);
    node<T>* rotateleft(node<T>* n);
```

```
    node<T>* rotateright(node<T>* n);
    node<T>* balance(node<T>* n);
    node<T>* findmin(node<T>* n);
    node<T>* removemin(node<T>* n);
    node<T>* insertbalanced(node<T>* n, T data);
    node<T>* removebalanced(node<T>* n, T data);


};


template<class T>
utils_tree<T>::utils_tree(std::string &str)
    : root(new node<T>())
{
    int i = 0;
    if(parse_tree(root, str, i))
    {
        delete root;
        root = nullptr;
    }
}

template<class T>
void utils_tree<T>::clean()
{
    clean(root);
}

template<class T>
void utils_tree<T>::insert(T data)
{
    insert(root, data);
}

template<class T>
void utils_tree<T>::remove(T data)
{
    remove(root, data);
}

template<class T>
bool utils_tree<T>::search(T key)
{
    return search(root, key) != nullptr;
}

template<class T>
bool utils_tree<T>::is_bst()
{
    return is_bst(root->left, INT_MIN, root->data) && is_bst(root->right, root-
>data, INT_MAX);
    // Костыль. Заменить INT_MIN/MAX на гендеро-нейтральный тип.
}

template<class T>
bool utils_tree<T>::is_pyramid()
{
    return is_pyramid(root->left, root->data) && is_pyramid(root->right, root-
>data);
}
```

```
template<class T>
int utils_tree<T>::max_depth()
{
    return max_depth(root, 1);
}

template<class T>
utils_tree<T>::~utils_tree()
{ // He protec
    clean(); // He attak
} // He destroy

template<class T>
bool utils_tree<T>::is_bst_stepped(utils_vector<node<T>*> &v, node<T> *n, T min,
T max)
{
    if (!n) return true;
    v.push_back(n);
    if (n->data <= min || n->data >= max) return false;
    return is_bst_stepped(v, n->left, min, n->data) && is_bst_stepped(v, n-
>right, n->data, max);
}

template<class T>
bool utils_tree<T>::is_pyramid_stepped(utils_vector<node<T>*> &v, node<T> *n,
int max)
{
    if (!n) return true;
    v.push_back(n);
    if (n->data >= max) return false;
    return is_pyramid_stepped(v, n->left, n->data) && is_pyramid_stepped(v, n-
>right, n->data);
}

template<class T>
void utils_tree<T>::to_optimal()
{
    utils_vector<int> datas = utils_vector<int>();
    get_data(datas, root);
    //datas.sort();
    delete root;
    root = new node<T>(datas[0]);
    for (int i = 1; i < datas.size(); i++)
    {
        insertbalanced(datas[i]);
    }
}

template<class T>
void utils_tree<T>::insertbalanced(T data)
{
    if (search(data)) return;
    if (!root) root = new node<T>(data);
    if (data < root->data) root->left = insertbalanced(root->left, data);
    else root->right = insertbalanced(root->right, data);
    root = balance(root);
}

template<class T>
void utils_tree<T>::removebalanced(T data)
{
    if (previous)
```

```
        root = removebalanced(root, data);
}

template<class T>
void utils_tree<T>::go_darker()
{
    go_darker(root);
}

template<class T>
std::string utils_tree<T>::node_string()
{
    return "Tree: {" + node_string(root) + "}";
}

// PRIVATE

template<class T>
bool utils_tree<T>::parse_tree(node<T>*& n, std::string &s, int &i) {
    if (i >= s.size() || s[static_cast<unsigned long>(i)] == ')')
    {
        delete n;
        n = nullptr;
        return false;
    }
    if (s[static_cast<unsigned long>(i)] == '(')
    {
        i++;
    }
    int num;
    int start = i;
    while (i != static_cast<int>(s.size()) && s[static_cast<unsigned long>(i)] !
= '(' && s[static_cast<unsigned long>(i)] != ')')
    {
        i++;
    }
    try
    {
        num = stoi(s.substr(static_cast<unsigned long>(start), static_cast<un-
signed long>(i) - static_cast<unsigned long>(start)));
    }
    catch (...)
    {
        return true;
    }
    n->data = num;
    n->left = new node<T>();
    n->right = new node<T>();
    if(parse_tree(n->left, s, i) || parse_tree(n->right, s, i)) return true;
    if (s[static_cast<unsigned long>(i)] == ')')
    {
        i++;
    }
    return false;
}


template<class T>
void utils_tree<T>::clean(node<T> *n)
{
    if (!n) return;
    clean(n->left);
```

```
    clean(n->right);
    delete root;
}

template<class T>
void utils_tree<T>::insert(node<T>*& n, T data)
{
    if (!n)
    {
        n = new node<T>(data);
    }
    else if (n->data < data)
    {
        insert(n->left, data);
    }
    else if (n->data > data)
    {
        insert(n->right, data);
    }
}

template<class T>
void utils_tree<T>::remove(node<T>*& n, T data)
{
    if (!n) return;
    if (data < n->data)
    {
        remove(n->left, data);
    }
    else if (data > n->data)
    {
        remove(n->right, data);
    }
    else
    {
        if (!n->left && n->right)
        {
            node<T>* temp = n->right;
            delete n;
            n = temp;
        }
        else if (!n->right && n->left)
        {
            node<T>* temp = n->left;
            delete n;
            n = temp;
        }
        node<T>* min = n->right;
        if (!min->left)
        {
            n->right = nullptr;
        }
        else
        {
            node<T>* t = min;
            while(t->left->left)
            {
                t = n->left;
            }
            min = t->left;
            t->left = nullptr;
        }
```

```
        n->data = min->data;
        delete min;
    }
}

template<class T>
node<T>* utils_tree<T>::search(node<T> *n, T key)
{
    if (!n) return nullptr;
    if (n->data == key) return n;
    node<T>* l = search(n->left, key);
    if (l) return l;
    node<T>* r = search(n->right, key);
    if (r) return r;
    return nullptr;
}

template<class T>
bool utils_tree<T>::is_bst(node<T> *n, T min, T max)
{
    if (!n) return true;
    if (n->data <= min || n->data >= max) return false;
    return is_bst(n->left, min, n->data) && is_bst(n->right, n->data, max);
}

template<class T>
bool utils_tree<T>::is_pyramid(node<T> *n, int max)
{
    if (!n) return true;
    if (n->data >= max) return false;
    return is_pyramid(n->left, n->data) && is_pyramid(n->right, n->data);
}

template<class T>
int utils_tree<T>::max_depth(node<T> *n, int i)
{
    if (!n) return i;
    int l = max_depth(n->left, i + 1);
    int r = max_depth(n->right, i + 1);
    if (l > r) return l;
    else return r;
}

template<class T>
void utils_tree<T>::get_data(utils_vector<T> &vec, node<T> *n)
{
    if (!n) return;
    vec.push_back(n->data);
    get_data(vec, n->left);
    get_data(vec, n->right);
}

template<class T>
void utils_tree<T>::go_darker(node<T> *n)
{
    if (!n) return;
    n->color.setHsv(n->color.hue(), n->color.saturation(), n->color.value() /
1.05f);
    go_darker(n->left);
    go_darker(n->right);
}
```

```
template<class T>
std::string utils_tree<T>::node_string(node<T> *n)
{
    if (!n) return "^";
    return std::to_string(n->data) + "(" + node_string(n->left) + "," +
node_string(n->right) + ")";
}

//АВЛ

template<class T>
int utils_tree<T>::height(node<T> *n)
{
    return n ? n->height : 0;
}

template<class T>
int utils_tree<T>::bfactor(node<T> *n)
{
    return height(n->right) - height(n->left);
}

template<class T>
void utils_tree<T>::fixheight(node<T> *n)
{
    int hl = height(n->left);
    int hr = height(n->right);
    n->height = (hl > hr ? hl : hr) + 1;
}

template<class T>
node<T>* utils_tree<T>::rotateleft(node<T> *n)
{
    n->color.setRgb(255, 0, 0);
    n->right->color.setRgb(255, 0, 0);
    //n->right->left->color.setRgb(255, 0, 0);
    node<T>* p = n->right;
    n->right = p->left;
    p->left = n;
    fixheight(n);
    fixheight(p);
    return p;
}

template<class T>
node<T>* utils_tree<T>::rotateright(node<T> *n)
{
    n->color.setRgb(255, 0, 0);
    n->left->color.setRgb(255, 0, 0);
    //n->left->right->color.setRgb(255, 0, 0);
    node<T>* q = n->left;
    n->left = q->right;
    q->right = n;
    fixheight(n);
    fixheight(q);
    return q;
}

template<class T>
node<T>* utils_tree<T>::balance(node<T> *n)
{
    fixheight(n);
```

```
    if (bfactor(n) == 2)
    {
        if (bfactor(n->right) < 0)
            n->right = rotateright(n->right);
        return rotateleft(n);
    }
    if (bfactor(n) == -2)
    {
        if (bfactor(n->left) > 0)
            n->left = rotateleft(n->left);
        return rotateright(n);
    }
    return n;
}

template<class T>
node<T> *utils_tree<T>::findmin(node<T> *n)
{
    return n->left ? findmin(n->left) : n;
}

template<class T>
node<T> *utils_tree<T>::removemin(node<T> *n)
{
    if (!n->left) return n->right;
    n->left = removemin(n->left);
    return balance(n);
}

template<class T>
node<T> *utils_tree<T>::insertbalanced(node<T> *n, T data)
{
    if (!n) return new node<T>(data);
    if (data < n->data) n->left = insertbalanced(n->left, data);
    else n->right = insertbalanced(n->right, data);
    return balance(n);
}

template<class T>
node<T> *utils_tree<T>::removebalanced(node<T> *n, T data)
{
    if (!n) return nullptr;
    if (data < n->data)
    {
        n->left = removebalanced(n->left,data);
    }
    else if (data > n->data)
    {
        n->right = removebalanced(n->right,data);
    }
    else
    {
        node<T>* l = n->left;
        node<T>* r = n->right;
        delete n;
        if (!r) return l;
        node<T>* min = findmin(r);
        min->right = removemin(r);
        min->left = l;
        return balance(min);
    }
    return balance(n);
```

```
}


#endif // UTILS_TREE_H
```

Файл mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    tree(nullptr),
    state(0),
    locked(false),
    lockedUpd(false),
    stepped_mode(empty),
    ui(new Ui::MainWindow),
    mainGraphicsScene(new QGraphicsScene())
{
    ui->setupUi(this);
    ui->graphicsView->setScene(mainGraphicsScene);
    QMainWindow::showMaximized();
    QColor color = QColor(203, 119, 47);

    pen.setColor(color);
    brush.setColor(color);
    font.setFamily("Roboto");
    pen.setWidth(3);

    srand(static_cast<unsigned long>(time(nullptr)));
    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(updateTreeColor()));
    timer->start(100);
    stack = utils_linked<node<int>*>();
    ui->horizontalSlider->setValue(2);
    ui->graphicsView->resetTransform();
    ui->graphicsView->scale(1.0 / 2, 1.0 / 2);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::UpdateGraphics()
{
    if (lockedUpd) return;
    lockedUpd = true;
    mainGraphicsScene->clear();
    if (!tree) return;
    DrawNode(tree->root, tree->max_depth());
    lockedUpd = false;
}

void MainWindow::on_butFile_clicked()
{
    std::string inputStr;
    QString fileName = QFileDialog::getOpenFileName(this, "Open TXT File",
QDir::homePath(), "TXT text (*.txt);;All Files (*)");
    if (fileName == nullptr)
    {
```

```
        QMessageBox::warning(this, "Warning", "File name is empty");
        return;
    }
    QFile file(fileName);
    if(file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QTextStream stream(&file);
        foreach (QString i ,QString(stream.readAll()).split(QRegExp("[ \t]"),
QString::SkipEmptyParts))
        inputStr.append(i.toUtf8().constData());
    }
    if(inputStr.empty())
        return;
    file.close();
    ui->input->setText(QString::fromUtf8(inputStr.c_str()));
}

void MainWindow::on_butGenerate_clicked()
{
    std::string readingStr;
    if (ui->input->text().isEmpty())
    {
        QMessageBox::warning(this, "Warining!", "Input is empty. Applying test:
13(8(6(3(1)(4))(7))(10(9)(11)))(14))");
        ui->input->setText("13(8(6(3(1)(4))(7))(10(9)(11)))(14))");
    }
    QString tempInp = ui->input->text();
    QTextStream stream(&tempInp);
    foreach (QString i, QString(stream.readAll()).split(QRegExp("[ \t]"),
QString::SkipEmptyParts))
        readingStr.append(i.toUtf8().constData());
    tree = new utils_tree<int>(readingStr);
    is_bst = tree->is_bst_stepped(bst_stepped, tree->root, INT_MIN, INT_MAX);
    is_pyramid = tree->is_pyramid_stepped(pyramid_stepped, tree->root, INT_MAX);
    ui->balancedLabel->setText("Binary tree");
    UpdateGraphics();
}

void MainWindow::on_butGenerateOptimal_clicked()
{
    if (!tree) on_butGenerate_clicked();
    if (!tree || tree->max_depth() <= 2) return;
    tree->to_optimal();
    ui->balancedLabel->setText("AVL tree");
    UpdateGraphics();
}

void MainWindow::on_butRun_clicked()
{
    if (locked || !tree) return;
    locked = true;
    if (is_bst) QMessageBox::information(this, "BST?", "    YES    ");
    else QMessageBox::warning(this, "BST?", "    NO    ");
    if (is_pyramid) QMessageBox::information(this, "Pyramid?", "    YES    ");
    else QMessageBox::warning(this, "Pyramid?", "    NO    ");
    UpdateGraphics();
    locked = false;
}

void MainWindow::DrawNode(node<int> *n, int maxdepth, int depth, int x, int y)
{
    if (n == nullptr) return;
    int offset = pow(2, maxdepth + 3) / pow(2, depth);
```

```
    if (n->left) mainGraphicsScene->addLine(x + 32, y + 32, x - offset + 32, y +
64 + 32, pen);
    if (n->right) mainGraphicsScene->addLine(x + 32, y + 32, x + offset + 32, y
+ 64 + 32, pen);
    QBrush brush(n->color);
    color.setRgb(0, 255 * (depth/(float)maxdepth), 255 * ((maxdepth - depth)/
(float)maxdepth));
    QPen pen(color, 3);
    mainGraphicsScene->addEllipse(x, y, 64, 64, pen, brush);
    QGraphicsTextItem *numb = new QGraphicsTextItem();
    numb->setPlainText(QString::number(n->data));
    numb->setDefaultTextColor(Qt::white);
    numb->setScale(2);
    numb->setPos(x + 16, y + 8);
    mainGraphicsScene->addItem(numb);
    DrawNode(n->left, maxdepth, depth + 1, x - offset, y + 64);
    DrawNode(n->right, maxdepth, depth + 1, x + offset, y + 64);
}

void MainWindow::SetActiveButtons(bool mode)
{
    ui->butAdd->setEnabled(mode);
    ui->butStepAdd->setEnabled(mode);
    ui->butRemove->setEnabled(mode);
    ui->butStepRemove->setEnabled(mode);
    ui->butFile->setEnabled(mode);
    ui->butStepBst->setEnabled(mode);
    ui->butStepPyramid->setEnabled(mode);
    ui->butRun->setEnabled(mode);
    ui->butGenerate->setEnabled(mode);
    ui->butGenerateOptimal->setEnabled(mode);

    ui->inputElement->setEnabled(mode);
    ui->input->setEnabled(mode);
}

bool MainWindow::ReadElement()
{
    if (!tree) return false;
    if (ui->inputElement->text().isEmpty())
    {
        QMessageBox::warning(this, "Warining!", "Input elemet is empty. Applying
random data");
        ui->inputElement->setText(QString::number(rand() % 20));
    }
    bool ok;
    int res = ui->inputElement->text().toInt(&ok);
    if (!ok)
    {
        QMessageBox::warning(this, "Warining!", "Input elemet isn't a number!");
        ui->inputElement->setText("");
        return false;
    }
    current = res;
    return true;
}


void MainWindow::on_horizontalSlider_sliderMoved(int position)
{
    ui->graphicsView->resetTransform();
    ui->graphicsView->scale(1.0 / position, 1.0 / position);
```

```
}

void MainWindow::on_butStepBst_clicked()
{
    if (stepped_mode == empty || stepped_mode == bst)
    {
        stepped_mode = bst;
        SetActiveButtons(false);
        ui->butStepBst->setEnabled(true);
        if (bst_stepped.empty())
        {
            if (is_bst) QMessageBox::information(this, "BST?", "    YES    ");
            else QMessageBox::warning(this, "BST?", "    NO    ");
            stepped_mode = empty;
            SetActiveButtons(true);
        }
        else
        {
            node<int>* n = bst_stepped.pop_front();
            n->color.setRgb(0, 255, 0);
        }
        UpdateGraphics();
    }
}

void MainWindow::on_butStepPyramid_clicked()
{
    if (stepped_mode == empty || stepped_mode == pyramid)
    {
        stepped_mode = pyramid;
        SetActiveButtons(false);
        ui->butStepPyramid->setEnabled(true);
        if (pyramid_stepped.empty())
        {
            if (is_pyramid) QMessageBox::information(this, "Pyramid?", "    YES
");
            else QMessageBox::warning(this, "Pyramid?", "    NO    ");
            stepped_mode = empty;
            SetActiveButtons(true);
        }
        else
        {
            node<int>* n = pyramid_stepped.pop_front();
            n->color.setRgb(0, 0, 255);
        }
        UpdateGraphics();
    }
}

void MainWindow::on_butAdd_clicked()
{
    if (!ReadElement()) return;
    if (tree->search(current))
    {
        QMessageBox::information(this, "Insertion", "Duplicate");
    }
    else
    {
        tree->insertbalanced(current);
    }
    ui->inputElement->setText("");
    UpdateGraphics();
```

```
}

void MainWindow::on_butStepAdd_clicked()
{
    if (!ReadElement()) return;
    if (locked || !tree) return;
    locked = true;
    if (stepped_mode == empty || stepped_mode == step_add)
    {
        stepped_mode = step_add;
        SetActiveButtons(false);
        ui->butStepAdd->setEnabled(true);
        node<int>* n;
        switch (state)
        {
            case 0: // Init
                stack.clean();
                state_node = tree->root;
                state = 1;
                break;
            case 1: // Down
                stack.push_back(state_node);
                state_node->color.setRgb(0, 255, 0);
                if (current < state_node->data)
                {
                    state_node = state_node->left;
                    if (!state_node)
                    {
                        stack.back()->left = new node<int>(current);
                        state = 2;
                    }
                }
                else if (current > state_node->data)
                {
                    state_node = state_node->right;
                    if (!state_node)
                    {
                        stack.back()->right = new node<int>(current);
                        state = 2;
                    }
                }
                else
                {
                    QMessageBox::information(this, "Insertion", "Duplicate");
                    stepped_mode = empty;
                    SetActiveButtons(true);
                    state = 0;
                }
                break;
            case 2: // Up
                if (stack.size() <= 1)
                {
                    QMessageBox::information(this, "Insertion", "Completed");
                    stepped_mode = empty;
                    SetActiveButtons(true);
                    state = 0;
                    break;
                }
                n = stack.pop_back();
                n->color.setRgb(0, 0, 255);
                tree->fixheight(n);
```

```
                if (tree->bfactor(n) == 2)
                {
                    if (tree->bfactor(n->right) < 0) n->right = tree->rota-
teright(n->right);
                    if (n == stack.back()->left) stack.back()->left = tree->ro-
tateleft(n);
                    else stack.back()->right = tree->rotateleft(n);
                }
                if (tree->bfactor(n) == -2)
                {
                    if (tree->bfactor(n->left) > 0) n->left = tree-
>rotateleft(n->left);
                    if (!stack.back())
                    {
                        QMessageBox::critical(this, "Insertion", "NULL");
                    }
                    if (n == stack.back()->left) stack.back()->left = tree->ro-
tateright(n);
                    else stack.back()->right = tree->rotateright(n);
                }
                break;
        }
        UpdateGraphics();
    }
    locked = false;
    //
//    if (!n) return new node<T>(data);
//    if (data < n->data) n->left = insertbalanced(n->left, data);
//    else n->right = insertbalanced(n->right, data);
//    fixheight(n);
//    if (bfactor(n) == 2)
//    {
//        if (bfactor(n->right) < 0)
//            n->right = rotateright(n->right);
//        return rotateleft(n);
//    }
//    if (bfactor(n) == -2)
//    {
//        if (bfactor(n->left) > 0)
//            n->left = rotateleft(n->left);
//        return rotateright(n);
//    }
//    return n;
    //
}

void MainWindow::on_butRemove_clicked()
{
    if (!ReadElement()) return;
    if (tree->search(current))
    {
        tree->removebalanced(current);
        ui->inputElement->setText("");
        UpdateGraphics();
    }
    else
    {
        QMessageBox::information(this, "Removing", "There is no " +
QString::number(current) + " element.");
    }
}
```

```
// FIX IT
void MainWindow::on_butStepRemove_clicked()
{
    if (!ReadElement()) return;
    if (stepped_mode == empty || stepped_mode == step_remove)
    {
        stepped_mode = step_remove;
        SetActiveButtons(false);
        ui->butStepRemove->setEnabled(true);
        node<int>* l;
        node<int>* r;
        node<int>* min;
        switch (state)
        {
            case 0: // Init
                stack.clean();
                state_node = tree->root;
                state = 1;
                break;
            case 1: // Down
                if (!state_node)
                {
                    QMessageBox::information(this, "Removing", "No element");
                    stepped_mode = empty;
                    SetActiveButtons(true);
                    state = 0;
                }
                else
                {
                    if (current < state_node->data)
                    {
                        stack.push_back(state_node);
                        state_node->color.setRgb(0, 255, 0);
                        state_node = state_node->left;
                    }
                    else if (current > state_node->data)
                    {
                        stack.push_back(state_node);
                        state_node->color.setRgb(0, 255, 0);
                        state_node = state_node->right;
                    }
                    else
                    {
                        l = state_node->left;
                        r = state_node->right;
                        if (!r)
                        {
                            if (state_node == tree->root) tree->root = l;
                            else if (state_node == stack.back()->left) stack-
.back()->left = l;
                            else stack.back()->right = l;
                        }
                        else
                        {
                            min = tree->findmin(r);
                            min->right = tree->removemin(r);
                            min->left = l;
                            if (state_node == tree->root) tree->root = tree-
>balance(min);
                            else if (state_node == stack.back()->left) stack-
.back()->left = tree->balance(min);
                            else stack.back()->right = tree->balance(min);
```

```
                        }
                        delete state_node;
                        state = 2;
                    }
                }
                break;
            case 2: // Up
                if (stack.size() <= 0)
                {
                    QMessageBox::information(this, "Removing", "Completed");
                    stepped_mode = empty;
                    SetActiveButtons(true);
                    state = 0;
                }
                else
                {
                    l = stack.pop_back();
                    l->color.setRgb(0, 0, 255);
                    if (l == tree->root) tree->root = tree->balance(l);
                    else if (l == stack.back()->left) stack.back()->left = tree-
>balance(l);
                    else stack.back()->right = tree->balance(l);
                }
                break;
        }
    }
    UpdateGraphics();
//    if (!n) return nullptr;
//    if (data < n->data)
//    {
//        n->left = removebalanced(n->left,data);
//    }
//    else if (data > n->data)
//    {
//        n->right = removebalanced(n->right,data);
//    }
//    else
//    {
//        node<T>* l = n->left;
//        node<T>* r = n->right;
//        delete n;
//        if (!r) return l;
//        node<T>* min = findmin(r);
//        min->right = removemin(r);
//        min->left = l;
//        return balance(min);
//    }
//    return balance(n);
}

void MainWindow::updateTreeColor()
{
    if (!tree || locked || lockedUpd) return;
    tree->go_darker();
    UpdateGraphics();
}
```