Министерство образования и науки Российской Федерации  
Федеральное‌ ‌государственное‌ ‌бюджетное‌ ‌образовательное‌ ‌учреждение‌

высшего‌ ‌образования‌

**«Пермский национальный исследовательский  
политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**О Т Ч Ё Т**

**по лабораторной работе №21**

Дисциплина: «информатика»

Тема: бинарные деревья

Вариант 18

Выполнил работу

студент группы РИС-20-1б

Тараканов Д.М.

Проверила

Доцент кафедры ИТАС

Полякова О.А.

Пермь, 2021

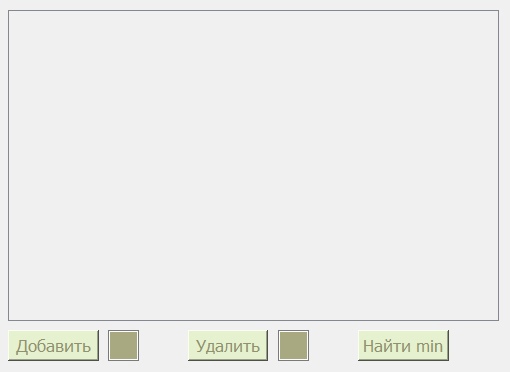
**Постановка задачи**

1. Сформировать идеально сбалансированное бинарное дерево.
2. Распечатать полученное дерево.
3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.
4. Преобразовать идеально сбалансированное дерево в дерево поиска.
5. Распечатать полученное дерево.
6. Тип информационного поля int. Найти минимальный элемент в дереве.

**Анализ задачи**

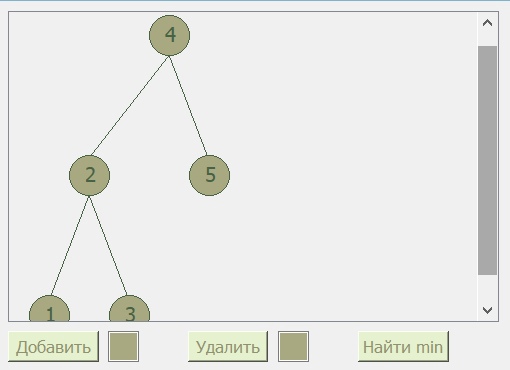
1. Для решения задачи необходимо…

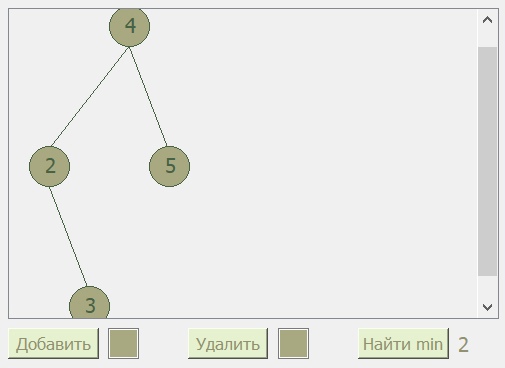
* Проработать лаконичный интерфейс для пользователя, ограничивающийся всего тремя кнопками, необходимыми для решения задачи;



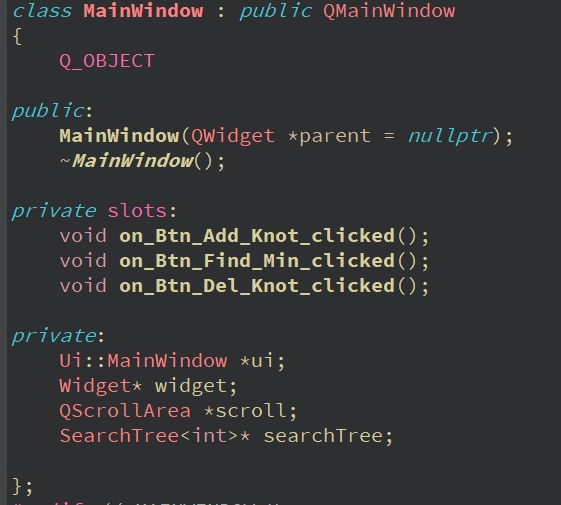
Кнопка «добавить» после введения какого—либо числа в окошко справа добавляет дочерний узел бинарного дерева с этим числом, либо же, если до этого значения узлов не вводились, то введенное число назначается коренным узлом дерева;

Если введенный элемент уже существует, то в консоли выводится ошибка о том, что данный элемент уже существует;

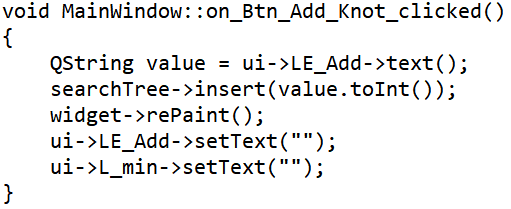




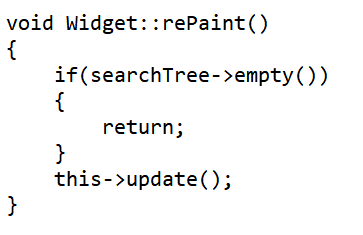
* Создать класс MainWindow, в котором будут указаны все параметры, указывающие на окно программы;



* Создать функцию Mainwindow::on\_Btn\_Add\_Knot\_clicked() типа void, которая отвечает за нажатия на кнопки внутри программы;



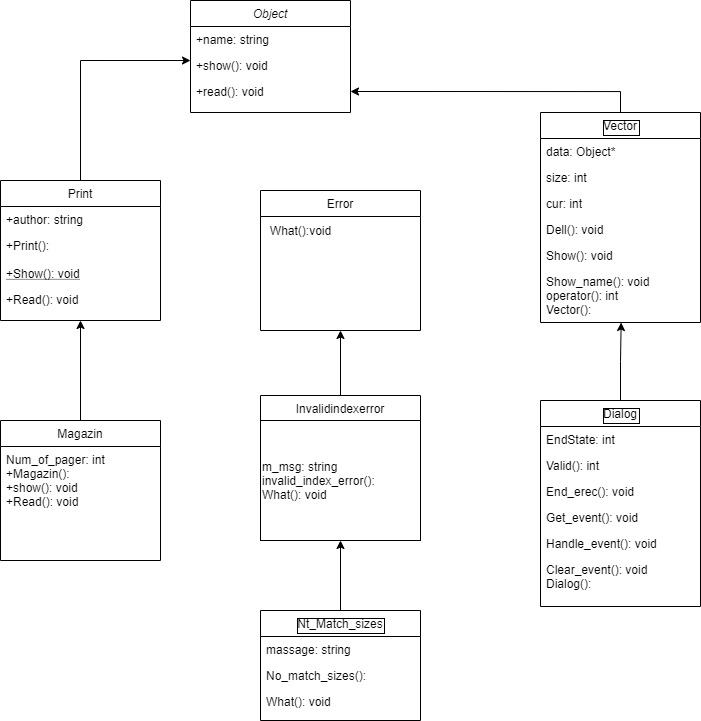
* Создать функцию Wedget::repaint(), отвечающую за перерисовку бинарного дерева, когда удаляются элементы либо же добавляются новые;



* Написать функцию, возвращающую тип bool и отвечающую за удаление элемента;



**Диаграмма классов**



**Решение**

**Main.cpp**

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}

**mainwindow.cpp**

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include <QIntValidator>

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow)

{

ui->setupUi(this);

searchTree = new SearchTree<int>;

widget = new Widget(searchTree);

scroll = new QScrollArea;

scroll->setWidget(widget);

ui->verticalLayout->addWidget(scroll);

ui->LE\_Add->setValidator(new QIntValidator(1, 99, this));

ui->LE\_Del->setValidator(new QIntValidator(1, 99, this));

}

MainWindow::~MainWindow()

{

delete ui;

}

void MainWindow::on\_Btn\_Add\_Knot\_clicked()

{

QString value = ui->LE\_Add->text();

searchTree->insert(value.toInt());

widget->rePaint();

ui->LE\_Add->setText("");

ui->L\_min->setText("");

}

void MainWindow::on\_Btn\_Find\_Min\_clicked()

{

if (!searchTree->empty())

{

int value = searchTree->findMin();

ui->L\_min->setText(QString::number(value));

}

else

{

cout << "Empty binary search tree! " << endl;

}

}

void MainWindow::on\_Btn\_Del\_Knot\_clicked()

{

QString value = ui->LE\_Del->text();

searchTree->deleteKnot(value.toInt());

widget->rePaint();

ui->LE\_Del->setText("");

ui->L\_min->setText("");

}

**mainwindow.h**

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include <QWidget>

#include <QScrollArea>

#include "search.h"

#include "widget.h"

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

private slots:

void on\_Btn\_Add\_Knot\_clicked();

void on\_Btn\_Find\_Min\_clicked();

void on\_Btn\_Del\_Knot\_clicked();

private:

Ui::MainWindow \*ui;

Widget\* widget;

QScrollArea \*scroll;

SearchTree<int>\* searchTree;

};

#endif // MAINWINDOW\_H

**Searchtree.h**

#ifndef SEARCHTREE\_H

#define SEARCHTREE\_H

#include <iostream>

#include <QPainter>

using namespace std ;

template <class T>

class SearchTree;

template <class T>

class Node

{

friend class SearchTree<T>;

public:

Node(const T&);

private:

T data;

int x;

Node<T>\* leftChild;

Node<T>\* rightChild;

Node<T>\* parent;

};

template <class T>

Node<T>::Node(const T& info)

{

data=info;

x=0;

leftChild = nullptr;

rightChild = nullptr;

parent = nullptr;

}

template <class T>

class SearchTree

{

public:

SearchTree();

~SearchTree();

bool empty() const;

bool insert(const T&);

bool deleteKnot (T);

int findMin() const;

int getDepth() const;

int getTotalY() const;

int getTotalX() const;

//int maxOfTwo(int, int) const;

void draw(QPainter\* painter);

private:

void reDraw(Node<T>\*);

void delNodes(const Node<T>\*);

int getLevel(Node<T>\*);

void rePosNode(Node<T>\*);

int depthTree(const Node<T>\*) const;

Node<T>\* getLowerLeftTree(Node<T>\*) const;

int getXLeftTree(const Node<T>\*);

int getXParent(const Node<T>\*);

private:

QPainter\* painter;

Node<T>\* root;

int shift\_x;

int shift\_y;

int nodeRadius;

};

template <class T>

SearchTree<T>::SearchTree()

{

root = nullptr;

}

template <class T>

SearchTree<T>::~SearchTree()

{

delNodes(root);

root = nullptr;

}

template <class T>

bool SearchTree<T>::empty() const

{

return root == nullptr;

}

template <class T>

bool SearchTree<T>::insert(const T& info)

{

Node<T>\* newNode = new Node<T>(info);

if (empty())

{

root = newNode;

return true;

}

Node<T> \*currentNode = root;

Node<T> \*tempNode = root;

while (currentNode != nullptr)

{

if (currentNode->data < info)

{

tempNode = currentNode;

currentNode = currentNode->rightChild;

}

else if (currentNode->data > info)

{

tempNode = currentNode;

currentNode = currentNode->leftChild;

}

else {

cout << "Duplicate value: " << currentNode->data <<endl;

return false;

}

}

if (tempNode->data < info){

tempNode->rightChild = newNode;

}

else

{

tempNode->leftChild = newNode;

}

newNode->parent = tempNode;

return true;

}

template <class T>

bool SearchTree<T>::deleteKnot(T info)

{

if (empty())

{

cout << "Empty binary search tree! " << endl;

return false;

}

bool flag = false;

Node<T> \*currentNode = root;

Node<T> \*tempNode = root;

while(!flag && currentNode != nullptr)

{

if (currentNode->data == info)

{

flag=true;

}

else if (currentNode->data < info)

{

tempNode = currentNode;

currentNode = currentNode->rightChild;

}

else

{

tempNode = currentNode;

currentNode = currentNode->leftChild;

}

}

if (!flag)

{

cout << "Not found value: " << info << endl;

return flag;

}

if (currentNode->leftChild == nullptr && currentNode->rightChild == nullptr)

{

if (currentNode == root)

{

delete root;

root = nullptr;

}

else if (tempNode->data < info)

{

delete tempNode->rightChild;

tempNode->rightChild = nullptr;

}

else

{

delete tempNode->leftChild;

tempNode->leftChild = nullptr;

}

return flag;

}

if (currentNode->leftChild == nullptr && currentNode->rightChild != nullptr)

{

if (currentNode == root)

{

Node<T>\* delNode = root;

root->rightChild->parent = nullptr;

root = root->rightChild;

delete delNode;

return flag;

}

if (tempNode->data < info)

{

Node<T>\* delNode = tempNode->rightChild;

tempNode->rightChild = currentNode->rightChild;

currentNode->rightChild->parent = tempNode;

delete delNode;

}

else

{

Node<T>\* delNode = tempNode->leftChild;

tempNode->leftChild = currentNode->rightChild;

currentNode->rightChild->parent = tempNode;

delete delNode;

}

return flag;

}

if (currentNode->leftChild != nullptr && currentNode->rightChild == nullptr)

{

if (currentNode == root)

{

Node<T>\* delNode = root;

root->leftChild->parent = nullptr;

root = root->leftChild;

delete delNode;

return flag;

}

if (tempNode->data < info)

{

Node<T>\* delNode = tempNode->rightChild;

tempNode->rightChild = currentNode->leftChild;

currentNode->leftChild->parent = tempNode;

delete delNode;

}

else

{

Node<T>\* delNode = tempNode->leftChild;

tempNode->leftChild = currentNode->leftChild;

currentNode->leftChild->parent = tempNode;

delete delNode;

}

return flag;

}

if (currentNode->leftChild != nullptr && currentNode->rightChild != nullptr)

{

Node<T>\* delNode = currentNode;

tempNode = currentNode;

delNode = delNode->leftChild;

if(delNode->rightChild == nullptr)

{

if(delNode->leftChild == nullptr)

{

currentNode->data = delNode->data;

currentNode->leftChild=nullptr;

delete delNode;

}

else

{

currentNode->data = delNode->data;

currentNode->leftChild = delNode->leftChild;

delNode->leftChild->parent = currentNode;

delete delNode;

}

}

else

{

while (delNode->rightChild != nullptr)

{

delNode = delNode->rightChild;

}

tempNode = delNode->parent;

tempNode->rightChild = nullptr;

currentNode->data = delNode->data;

delete delNode;

}

}

return flag;

}

template <class T>

int SearchTree<T>::findMin() const

{

Node<T>\* tempNode = getLowerLeftTree(root);

return tempNode->data;

}

template <class T>

void SearchTree<T>::delNodes(const Node<T>\* node)

{

if (node != nullptr)

{

delNodes(node->leftChild);

delNodes(node->rightChild);

delete node;

}

}

template <class T>

int SearchTree<T>::getDepth() const

{

if (empty())

{

return 0;

}

return depthTree(root);

}

template <class T>

int SearchTree<T>::depthTree(const Node<T>\* node) const

{

if (node == nullptr || (node->leftChild == nullptr && node->rightChild == nullptr))

{

return 0;

}

return 1 + max(depthTree(node->leftChild), depthTree(node->rightChild));

}

/\*template <class T>

int SearchTree<T>::maxOfTwo(int a, int b) const

{

return max(a, b);

}\*/

template <class T>

void SearchTree<T>::draw(QPainter\* painter)

{

if (root != nullptr)

{

this->painter = painter;

painter->setFont(QFont("Times", 12, QFont::Normal));

nodeRadius = 20;

shift\_x = nodeRadius;

shift\_y = nodeRadius \* 5;

rePosNode(root);

Node<T>\* lowerLeft = getLowerLeftTree(root);

lowerLeft->x = nodeRadius \* 2;

reDraw(root);

}

}

template <class T>

Node<T>\* SearchTree<T>::getLowerLeftTree(Node<T>\* node) const

{

if (node->leftChild == nullptr)

{

return node;

}

return getLowerLeftTree(node->leftChild);

}

template <class T>

void SearchTree<T>::reDraw(Node<T>\* node)

{

if (node != nullptr)

{

reDraw(node->leftChild);

int level = getLevel(node);

int y = level \* nodeRadius \* 2 + shift\_y \* (level - 1);

if (node->leftChild != nullptr)

{

node->x = getXLeftTree(node->leftChild) + nodeRadius + shift\_x;

painter->drawLine(QPoint(node->x, y + nodeRadius), QPoint(node->leftChild->x + 2, ((level + 1) \* nodeRadius \* 2 + shift\_y \* level) - nodeRadius));

}

else if (node->x == 0)

{

node->x = getXParent(node) + nodeRadius + shift\_x;

}

painter->drawEllipse(QPoint(node->x, y), nodeRadius, nodeRadius);

int shift\_text;

if (node->data < 10)

{

shift\_text = 4;

}

else

{

shift\_text = 10;

}

painter->drawText(QPoint(node->x - shift\_text, y + 7), QString::number(node->data));

reDraw(node->rightChild);

if (node->rightChild != nullptr)

{

painter->drawLine(QPoint(node->x, y + nodeRadius), QPoint(node->rightChild->x - 2, ((level + 1)\* nodeRadius \* 2 + shift\_y \* level) - nodeRadius));

}

}

}

template <class T>

int SearchTree<T>::getLevel(Node<T>\* node)

{

int level = 1;

Node<T>\* currentNode = node;

while (currentNode->parent != nullptr)

{

currentNode = currentNode->parent;

level++;

}

return level;

}

template <class T>

void SearchTree<T>::rePosNode(Node<T>\* node)

{

if (node != nullptr)

{

rePosNode(node->leftChild);

node->x = 0;

rePosNode(node->rightChild);

}

}

template <class T>

int SearchTree<T>::getTotalY() const

{

int level = getDepth() + 1;

return (level \* nodeRadius \* 2 + shift\_y \* (level - 1)) + nodeRadius \* 2;

}

template <class T>

int SearchTree<T>::getTotalX() const

{

if (root == nullptr)

{

return nodeRadius\*3;

}

Node<T>\* currentNode = root;

while (currentNode->rightChild != nullptr)

currentNode = currentNode->rightChild;

return currentNode->x + nodeRadius \* 3;

}

template <class T>

int SearchTree<T>::getXLeftTree(const Node<T>\* node)

{

if (node->rightChild == nullptr)

{

return node->x;

}

return getXLeftTree(node->rightChild);

}

template <class T>

int SearchTree<T>::getXParent(const Node<T>\* node)

{

Node<T>\* currentNode = node->parent;

while (currentNode->x == 0)

{

currentNode = currentNode->parent;

}

return currentNode->x;

}

#endif // SEARCHTREE\_H

**widget.cpp**

#include "widget.h"

Widget::Widget(SearchTree<int>\* searchTree, QWidget\* parent) : QWidget(parent), searchTree()

{

this->searchTree = searchTree;

node = QColor(168, 168, 129);

background = QColor(235, 235, 235);

text = QColor(71, 98, 68);

}

void Widget::rePaint()

{

if(searchTree->empty())

{

return;

}

this->update();

}

QSize Widget::sizeHint() const

{

return QSize(50, 50);

}

QSize Widget::minimumSizeHint() const

{

return QSize(50, 50);

}

void Widget::paintEvent(QPaintEvent\* event)

{

if (!searchTree->empty())

{

QPainter painter(this);

QBrush brush;

brush.setColor(node);

brush.setStyle(Qt::SolidPattern);

QPen pen;

pen.setColor(text);

painter.setBrush(brush);

painter.setPen(pen);

searchTree->draw(&painter);

Size();

}

}

void Widget::Size() {

QSize size(searchTree->getTotalX(), searchTree->getTotalY());

setMinimumSize(size);

resize(size);

}

**widget.h**

#ifndef WIDGET\_H

#define WIDGET\_H

#include <QWidget>

#include <QPen>

#include <QBrush>

#include <QColor>

#include "SearchTree.h"

class Widget : public QWidget

{

Q\_OBJECT

public:

Widget(SearchTree<int>\* searchTree, QWidget \*parent = nullptr);

public:

QSize minimumSizeHint() const override;

QSize sizeHint() const override;

void Size();

void rePaint();

protected:

void paintEvent(QPaintEvent \*event) override;

private:

SearchTree<int> \*searchTree;

QPen pen;

QBrush brush;

QColor background;

QColor node;

QColor text;

};

#endif // WIDGET\_H

**Скриншоты**

