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 №20**

Дисциплина: основы алгоритмизации и программирования

Тема: Бинарные деревья поиска

Вариант 7

Выполнил работу

студент группы ИВТ-20-2б

Солдатов Алексей Павлович

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Пермь, 2021

**Цель задачи**

Реализовать динамическую структуру данных бинарное дерево

Реализовать возможности добавления и удаления элементов

Реализовать графический вывод дерева

Постановка задачи

 1. Сформировать идеально сбалансированное бинарное дерево, тип информационного поля указан в варианте.

  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.

Анализ задачи

1. Определить какие операции должны быть выполнены по заданию:

Создание класса бинарного дерева

*class* **Node**

{

*friend* *class* BinarySearchTree<T> ;

*public*:

**Node**(*const* T &);

*private*:

T data;

int x;

Node<T> \*leftChild;

Node<T> \*rightChild;

Node<T> \*parent;

};

Добавление элементов в дерево

bool **insert**(*const* T &);

Удаление элементов дерева

bool **deleteItem**(T);

Для решения задачи используются переменные:

Несколько целочисленных переменных для управления циклами, контроля размеров массивов и контроля длины строк

int len = 0;

int adress;

people x;

string user\_search;

int error = 0;

Указатели

people\* data\_base = new people[len];

string\* key = new string[len];

1. Для решения задачи будут использованы циклы. Например, цикл для ввода информации в структуры

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

{

cout << "Город: ";

cin >> city[i].name;

cout << "Численость населения: ";

cin >> city[i].population;

}

1. Для защиты от случайных ошибок используются циклы на корректный ввод информации

while (size < 1)

{

cout << "Введите кол-во городов: ";

cin >> size;

cout << endl;

}

while (menu != 1 && menu != 2)

{

cout << "Выбери метод сортировки: \n 1. Хоара \n 2. Шелла \n Ввод: ";

cin >> menu;

}

Вывод дерева осуществлен средствами QTOpenGL myPainter::**myPainter**()

Код

#ifndef BINARYSEARCHTREE\_H\_

#define BINARYSEARCHTREE\_H\_

#include <iostream>

#include <QPainter>

#include <QStack>

#include <QQueue>

*using* *namespace* std ;

*template*<*typename* T> *class* **BinarySearchTree**;

*template*<*typename* T>

*class* **Node**

{

*friend* *class* BinarySearchTree<T> ;

*public*:

**Node**(*const* T &);

*private*:

T data;

int x;

Node<T> \*leftChild;

Node<T> \*rightChild;

Node<T> \*parent;

};

*template*<*typename* T>

*class* **BinarySearchTree**

{

*public*:

**BinarySearchTree**();

~**BinarySearchTree**();

bool **isEmpty**() *const*;

bool **insert**(*const* T &);

QString **getBreadthFirstSearch**();

int **getNodeCount**() *const*;

int **getLeafNodeCount**() *const*;

int **getTreeHeight**() *const*;

bool **deleteItem**(T);

bool **find**(T) *const*;

void **draw**(QPainter \*painter, double &scale);

int **getTotalY**() *const*;

int **getTotalX**() *const*;

bool **deleteAtLocation**(int x, int y);

*private*:

QPainter \*painter;

Node<T> \*root;

int yspace;

int xspace;

int nodeRadius;

double scale;

int **max**(int a, int b) *const*;

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

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

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

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

void **recursiveDraw**(Node<T> \*node);

Node<T>\* **getLeftmostNode**(Node<T> \*node) *const*;

int **getNodeLevel**(Node<T> \*node);

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

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

void **resetNodePosition**(Node<T> \*node);

bool **recursiveDeleteAtLocation**(Node<T> \*node, int x, int y);

};

*template*<*typename* T>

Node<T>::**Node**(*const* T &info) :data(info), x(0), leftChild(0), rightChild(0), parent(0){}

*template*<*typename* T>

int BinarySearchTree<T>::**max**(int a, int b) *const*

{

*return* a > b ? a : b;

}

*template*<*typename* T>

BinarySearchTree<T>::**BinarySearchTree**() : root(0), scale(1){}

*template*<*typename* T>

BinarySearchTree<T>::~**BinarySearchTree**()

{

recursiveDeleteNodes(root);

*this*->root = 0;

}

*template*<*typename* T>

bool BinarySearchTree<T>::**recursiveDeleteAtLocation**(Node<T> \*node, int x, int y)

{

*if* (node == 0)

*return* *false*;

*if* (recursiveDeleteAtLocation(*node->*leftChild, x, y) == *true*)

*return* *true*;

int level = getNodeLevel(*node*);

int nodey = (level \* nodeRadius \* 2 + yspace \* (level-1)) - nodeRadius;

int nodex = node->x - nodeRadius;

*if* (nodex <= x && x <=nodex + nodeRadius\*2 )

{

*if* (nodey <= y && y <= nodey + nodeRadius\*2)

{

deleteItem(node->data);

*return* *true*;

}

}

*return* recursiveDeleteAtLocation(*node->*rightChild, x, y);

}

*template*<*typename* T>

bool BinarySearchTree<T>::**deleteAtLocation**(int x, int y)

{

*return* recursiveDeleteAtLocation(*root*, x, y);

}

*template*<*typename* T>

bool BinarySearchTree<T>::**isEmpty**() *const*

{

*return* root == 0;

}

*template*<*typename* T>

bool BinarySearchTree<T>::**insert**(*const* T &item)

{

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

*if* (*this*->isEmpty())

{

*this*->root = newNode;

*return* *true*;

}

Node<T> \*currentNode = root;

Node<T> \*trailCurrentNode = root;

*while* (currentNode != 0)

{

*if* (currentNode->data < item)

{

trailCurrentNode = currentNode;

currentNode = currentNode->rightChild;

}

*else* *if* (currentNode->data > item)

{

trailCurrentNode = currentNode;

currentNode = currentNode->leftChild;

}

*else*

{

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

*return* *false*;

}

}

*if* (trailCurrentNode->data < item)

trailCurrentNode->rightChild = newNode;

*else*

trailCurrentNode->leftChild = newNode;

newNode->parent = trailCurrentNode;

*return* *true*;

}

*template*<*typename* T>

int BinarySearchTree<T>::**getTreeHeight**() *const*

{

*if* (*this*->isEmpty())

*return* 0;

*return* recursiveComputeHeightOfTree(root);

}

*template*<*typename* T>

bool BinarySearchTree<T>::**deleteItem**(T item)

{

*if* (*this*->isEmpty())

*return* *false*;

bool found = *false*;

Node<T> \*currentNode = root;

Node<T> \*trailCurrentNode = root;

*while* (!found && currentNode != 0)

{

*if* (currentNode->data == item)

found = *true*;

*else* *if* (currentNode->data < item)

{

trailCurrentNode = currentNode;

currentNode = currentNode->rightChild;

}

*else*

{

trailCurrentNode = currentNode;

currentNode = currentNode->leftChild;

}

}

*if* (!found)

*return* found;

*if* (currentNode->leftChild == 0 && currentNode->rightChild == 0)

{

*if* (currentNode == root)

{

*delete* root;

root = 0;

}

*else* *if* (trailCurrentNode->data < item)

{

*delete* trailCurrentNode->rightChild;

trailCurrentNode->rightChild = 0;

}

*else*

{

*delete* trailCurrentNode->leftChild;

trailCurrentNode->leftChild = 0;

}

*return* found;

}

*if* (currentNode->leftChild == 0 && currentNode->rightChild != 0)

{

*if* (currentNode == root)

{

Node<T> \*tempPtr = root;

root->rightChild->parent = 0;

root = root->rightChild;

*delete* tempPtr;

*return* found;

}

*if* (trailCurrentNode->data < item)

{

Node<T> \*tempPtr = trailCurrentNode->rightChild;

trailCurrentNode->rightChild = currentNode->rightChild;

currentNode->rightChild->parent = trailCurrentNode;

*delete* tempPtr;

}

*else*

{

Node<T> \*tempPtr = trailCurrentNode->leftChild;

trailCurrentNode->leftChild = currentNode->rightChild;

currentNode->rightChild->parent = trailCurrentNode;

*delete* tempPtr;

}

*return* found;

}

*if* (currentNode->leftChild != 0 && currentNode->rightChild == 0)

{

*if* (currentNode == root)

{

Node<T> \*tempPtr = root;

root = root->leftChild;

root->parent = 0;

*delete* tempPtr;

}

*if* (trailCurrentNode->data < item)

{

Node<T> \*tempPtr = trailCurrentNode->rightChild;

trailCurrentNode->rightChild = currentNode->leftChild;

currentNode->leftChild->parent = trailCurrentNode;

*delete* tempPtr;

}

*else*

{

Node<T> \*tempPtr = trailCurrentNode->leftChild;

trailCurrentNode->leftChild = currentNode->leftChild;

currentNode->leftChild->parent = trailCurrentNode;

*delete* tempPtr;

}

*return* found;

}

*if* (currentNode->leftChild != 0 && currentNode->rightChild != 0)

{

Node<T> \*ptr = currentNode;

trailCurrentNode = currentNode;

ptr = ptr->leftChild;

*if*(ptr->rightChild==NULL)

{

*if*(ptr->leftChild==NULL)

{

currentNode->data=ptr->data;

currentNode->leftChild=NULL;

*delete* ptr;

}

*else*

{

currentNode->data=ptr->data;

currentNode->leftChild=ptr->leftChild;

ptr->leftChild->parent=currentNode;

*delete* ptr;

}

}

*else*

{

*while* (ptr->rightChild!=0)

ptr=ptr->rightChild;

trailCurrentNode=ptr->parent;

trailCurrentNode->rightChild=NULL;

currentNode->data=ptr->data;

*delete* ptr;

}

}

*return* found;

}

*template*<*typename* T>

bool BinarySearchTree<T>::**find**(T item) *const*

{

*if* (*this*->isEmpty())

*return* *false*;

Node<T> \*currentNode = root;

*while* (currentNode != 0)

{

*if* (currentNode->data == item)

*return* *true*;

*else* *if* (currentNode->data < item)

currentNode = currentNode->rightChild;

*else*

currentNode = currentNode->leftChild;

}

*return* *false*;

}

*template*<*typename* T>

QString BinarySearchTree<T>::**getBreadthFirstSearch**()

{

QString traversal("");

Node<T> \*traverse;

*if* (*this*->root == 0)

*return* traversal;

QQueue<Node<T>\*> ptrQueue;

ptrQueue.enqueue(*this*->root);

*while* (!ptrQueue.isEmpty()) {

traverse = ptrQueue.dequeue();

traversal.append(QString::number(traverse->data) + " ");

*if* (traverse->leftChild != 0)

ptrQueue.enqueue(traverse->leftChild);

*if* (traverse->rightChild != 0)

ptrQueue.enqueue(traverse->rightChild);

}

*return* traversal;

}

*template*<*typename* T>

int BinarySearchTree<T>::**recursiveCountNodes**(*const* Node<T> \*node) *const*

{

*if* (node == 0)

*return* 0;

*return* (1 + recursiveCountNodes(node->leftChild)+ recursiveCountNodes(node->rightChild));

}

*template*<*typename* T>

int BinarySearchTree<T>::**recursiveCountLeafNodes**(*const* Node<T> \*node) *const*

{

*if* (node == 0)

*return* 0;

*if* (node->leftChild == 0 && node->rightChild == 0)

*return* 1;

*return* (recursiveCountLeafNodes(node->leftChild) + recursiveCountLeafNodes(node->rightChild));

}

*template*<*typename* T>

int BinarySearchTree<T>::**recursiveComputeHeightOfTree**(*const* Node<T> \*node) *const*

{

*if* (node == 0 || (node->leftChild == 0 && node->rightChild == 0))

*return* 0;

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

}

*template*<*typename* T>

void BinarySearchTree<T>::**recursiveDeleteNodes**(*const* Node<T> \*node)

{

*if* (node == 0)

*return*;

recursiveDeleteNodes(node->leftChild);

recursiveDeleteNodes(node->rightChild);

*delete* node;

*return*;

}

*template*<*typename* T>

void BinarySearchTree<T>::**resetNodePosition**(Node<T> \*node)

{

*if* (node == 0)

*return*;

resetNodePosition(*node->*leftChild);

node->x = 0;

resetNodePosition(*node->*rightChild);

*return*;

}

*template*<*typename* T>

void BinarySearchTree<T>::**draw**(QPainter \*painter, double &scale)

{

*if*(*this*->root == 0)

*return*;

*this*->painter = painter;

*this*->painter->setFont(QFont("Times", 12 \* scale, QFont::*Normal*));

*this*->scale = scale;

*this*->nodeRadius = 20 \* scale;

*this*->xspace = nodeRadius;

*this*->yspace = nodeRadius \* 5;

resetNodePosition(*root*);

Node<T> \*leftmost = getLeftmostNode(*root*);

leftmost->x = nodeRadius \* 2;

*this*->recursiveDraw(root);

*return*;

}

*template*<*typename* T>

Node<T>\* BinarySearchTree<T>::**getLeftmostNode**(Node<T> \*node) *const*

{

*if* (node->leftChild == 0)

*return* node;

*return* getLeftmostNode(*node->*leftChild);

}

*template*<*typename* T>

int BinarySearchTree<T>::**getNodeLevel**(Node<T> \*node)

{

int level = 1;

Node<T> \*current = node;

*while*(current->parent != 0){

current = current->parent;

++level;

}

*return* level;

}

*template*<*typename* T>

int BinarySearchTree<T>::**getPxLocOfLeftTree**(*const* Node<T> \*node)

{

*if*(node->rightChild == 0){

*return* node->x;

}

*return* getPxLocOfLeftTree(node->rightChild);

}

*template*<*typename* T>

int BinarySearchTree<T>::**getPxLocOfAncestor**(*const* Node<T> \*node)

{

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

*while*(currentNode->x == 0)

currentNode = currentNode->parent;

*return* currentNode->x;

}

*template*<*typename* T>

int BinarySearchTree<T>::**getTotalY**() *const*

{

int level = getTreeHeight() + 1;

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

}

*template*<*typename* T>

int BinarySearchTree<T>::**getTotalX**() *const*

{

*if* (*this*->root == 0)

*return* nodeRadius\*3;

Node<T> \*current = root;

*while* (current->rightChild != 0)

current = current->rightChild;

*return* current->x + nodeRadius \* 3;

}

*template*<*typename* T>

void BinarySearchTree<T>::**recursiveDraw**(Node<T> \*node)

{

*if* (node == 0)

*return*;

*this*->recursiveDraw(node->leftChild);

int level = getNodeLevel(*node*);

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

*if* (node->leftChild != 0)

{

node->x = getPxLocOfLeftTree(node->leftChild) + nodeRadius + xspace;

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

}

*else* *if* (node->x == 0)

node->x = getPxLocOfAncestor(node) + nodeRadius + xspace;

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

int textAdjuster;

*if*(abs(node->data) < 10)

textAdjuster = 4;

*else* *if* (abs(node->data) < 100)

textAdjuster = 7;

*else* *if* (abs(node->data) < 1000)

textAdjuster = 12;

*else*

textAdjuster = 16;

painter->drawText(QPoint(node->x-(textAdjuster\*scale), y+(5\*scale)), QString::number(node->data));

*this*->recursiveDraw(node->rightChild);

*if* (node->rightChild != 0)

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

*return*;

}

#endif

Код печати

#include "mainwindow.h"

#include "bst\_about\_window.h"

#include <QVBoxLayout>

#include <QHBoxLayout>

#include <QCloseEvent>

#include <QMenuBar>

#include <QSpacerItem>

#include <qglobal.h>

#include <QTime>

#include <QFileDialog>

#include <QStandardPaths>

#include <QFileInfo>

#include <QFile>

#include <QTextStream>

#include <QStringList>

#include <QStringListIterator>

MainWindow::**MainWindow**(QWidget \*parent) :

QMainWindow(*parent*)

{

QString directory = QStandardPaths::writableLocation(QStandardPaths::*DocumentsLocation*) ;

*if* (!QDir(directory).exists())

QDir().mkdir(directory);

*this*->bst = *this*->getBST();

*this*->createMenu();

deleteButton = *new* QPushButton("&Удалить", *this*);

insertButton = *new* QPushButton("Добавить", *this*);

insertValueLineEdit = *new* QLineEdit;

deleteValueLineEdit = *new* QLineEdit;

statusLabel = *new* QLabel;

deleteButton->setSizePolicy(QSizePolicy::*Fixed*, QSizePolicy::*Fixed*);

insertButton->setSizePolicy(QSizePolicy::*Fixed*, QSizePolicy::*Fixed*);

insertValueLineEdit->setFixedWidth(100);

insertValueLineEdit->setToolTip("Enter single value or multiple values separated by space");

deleteValueLineEdit->setFixedWidth(100);

deleteValueLineEdit->setToolTip("Enter value to delete");

connect(deleteButton, SIGNAL(clicked()), *this*, SLOT(deleteClicked()));

connect(insertButton, SIGNAL(clicked()), *this*, SLOT(insertClicked()));

connect(insertValueLineEdit, SIGNAL(returnPressed()), *this*, SLOT(insertClicked()));

connect(deleteValueLineEdit, SIGNAL(returnPressed()), *this*, SLOT(deleteClicked()));

QHBoxLayout \*buttonLayout = *new* QHBoxLayout;

buttonLayout->addWidget(*deleteButton*);

buttonLayout->addWidget(*deleteValueLineEdit*);

buttonLayout->addWidget(*insertButton*);

buttonLayout->addWidget(*insertValueLineEdit*);

buttonLayout->addSpacing(20);

buttonLayout->addWidget(*statusLabel*);

buttonLayout->addStretch(0);

renderArea = *new* RenderArea(*this->bst*);

treeScrollArea = *new* QScrollArea;

treeScrollArea->setWidget(*renderArea*);

treeScrollArea->installEventFilter(*renderArea*);

mainLayout = *new* QVBoxLayout;

mainLayout->addWidget(*treeScrollArea*);

mainLayout->addLayout(*buttonLayout*);

centralWidget = *new* QWidget(*this*);

centralWidget->setLayout(*mainLayout*);

*this*->setCentralWidget(*centralWidget*);

*this*->setMinimumHeight(400);

*this*->setWindowTitle("Binary Search Tree");

*this*->show();

}

void MainWindow::***resizeEvent***(QResizeEvent\* event)

{

QMainWindow::resizeEvent(*event*);

*this*->renderArea->callRepaint();

}

MainWindow::~***MainWindow***()

{

*delete* renderArea;

*delete* deleteButton;

*delete* insertButton;

*delete* treeScrollArea;

*delete* bst;

*delete* centralWidget;

}

void MainWindow::**createMenu**()

{

*this*->createActions();

*this*->menuBar()->addAction(*aboutAction*);

}

void MainWindow::**createActions**()

{

aboutAction = *new* QAction(tr(""), *this*);

}

void MainWindow::**deleteClicked**() *const* {

QString value = deleteValueLineEdit->text();

*if*(!*this*->bst->deleteItem(value.toInt()))

*this*->statusLabel->setText("");

*else*

*this*->statusLabel->setText("");

*this*->renderArea->repaint();

*this*->deleteValueLineEdit->setText("");

*return*;

}

void MainWindow::**insertClicked**() *const*

{

QString values = insertValueLineEdit->text();

QStringList valueList = values.split(QRegExp("\\s+"), QString::*SkipEmptyParts*);

QStringListIterator iterator(valueList);

*while* (iterator.hasNext())

{

*if*(!*this*->bst->insert(iterator.next().toInt()))

*this*->statusLabel->setText("");

*else*

*this*->statusLabel->setText("");

}

*this*->renderArea->repaint();

insertValueLineEdit->setText("");

*return*;

}

BinarySearchTree<int>\* MainWindow::**getBST**()

{

BinarySearchTree<int> \*bst = *new* BinarySearchTree<int>;

QString fileName = QStandardPaths::writableLocation(QStandardPaths::*DocumentsLocation*);

QString text;

QFile file(fileName);

*if* (!file.exists() || !file.*open*(QIODevice::*ReadOnly* | QIODevice::*Text*))

{

*return* bst;

}

QTextStream reader(*&file*);

*while* (!reader.atEnd())

{

reader >> text;

*if* (text != " " && text != "")

bst->insert(text.toInt());

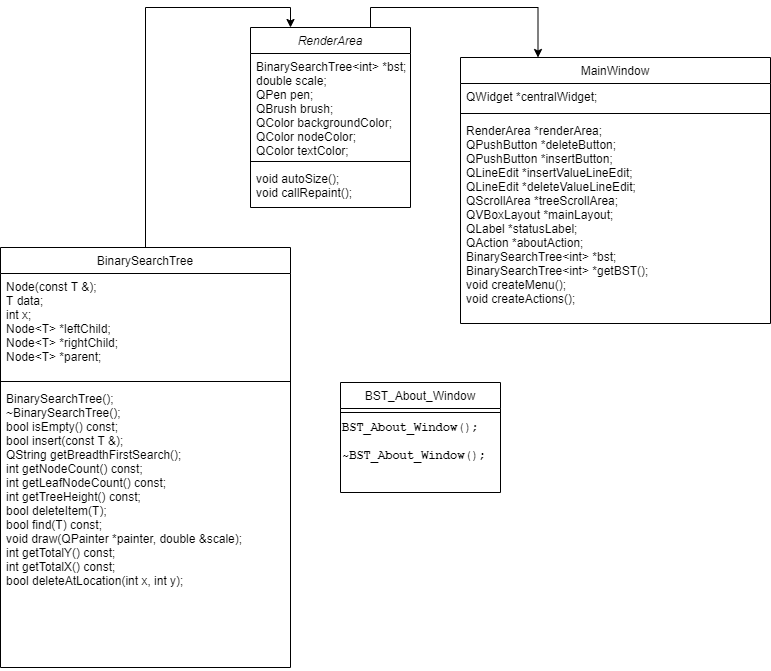
}

file.*close*();

*return* bst;

}

UML-диаграмма



Работа кода

