МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра математического обеспечения и применения ЭВМ

ОТЧЁТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 8381 _____ Почаев Н.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Изучить основные характеристики и реализовать структуру данных бинарное дерево (англ. *Binary tree*), а также такие его разновидности, как двоичное дерево поиска (англ. *binary search tree*, *BST*). Создать программу, выполняющую визуализацию заданного скобочной записью дерева, а также проверки его на принадлежность выше обозначенным подвидам.

Постановка задачи.

Для заданного бинарного дерева с числовым типом элементов определить, является ли оно бинарным деревом поиска и является ли оно пирамидой. Реализация дерева должна быть на базе вектора (массива).

Основные теоретические положения.

Бинарное дерево называется бинарным деревом поиска, если для каждого его узла справедливо: все элементы правого поддерева больше этого узла, а все элементы левого поддерева – меньше этого узла.

Бинарное дерево называется пирамидой, если для каждого его узла справедливо: значения всех потомков этого узла не больше, чем значение узла.

Выполнение работы.

Написание работы производилось на базе операционной системы Linux Manjaro в среде разработки Qt Creator с использование фреймворка Qt.

Для реализации графического интерфейса в стиле Material Design была использована сторонняя библиотека `laserpants/qt-material-widgets` по открытой лицензии с GitHub.

Для реализации пошагового режима алгоритмы проверки были реализованы в итеративном стиле, так как стандартный рекурсивный подход не позволяет выполнить ”петлю задержки” для ожидания следующего нажатия клави-

ши *NextStep*.

Алгоритм проверки бинарного дерева на BST. Корректное BST всегда следует правилу: left < root < right. Рекурсивным подходом является проверка того, что это правило не нарушено. Для перехода к итеративному решению данное правило может быть сопоставлено с обходом inorder. Таким образом, итерационный метод обхода по порядку двоичного дерева может быть использован для замены рекурсивного пути. В данном случае используется алгоритм на основе стека. Временная сложность O(n). Morris Traversal обход здесь не используется.

Алгоритм проверки бинарного дерева на Binary Heap. Идея обхода базируется на принципе, описанном в предыдущем пункте: рекурсивный обход в данном случае заменяется на level order, или уровневый обход. Основные пункты реализации:

1. Чтобы проверить структурное свойство, необходимо убедиться, что ни один непустой дочерний элемент не обнаружен ни для одного узла, для которого найден пустой дочерний элемент.
2. Чтобы проверить корректность кучи, необходимо убедиться, что левый и правый дочерние элементы больше родительского узла. В данном случае для этого используется тип данных очередь.

Тестирование

Таблица 1 – Результаты тестирования

Входная строка	Корректное построение?	Является BST?	Является Binary Heap?
1(2)(3)	да	нет	нет
4(2(3)(1))(6(5))	да	нет	нет
9(8(6(3(1)(4))(2))(5(3)(1)))(7))	да	нет	нет
16(11(10(1)(2)(5(4)))(9(6)(8)))	да	да	нет
8(3(1)(6(4)(7))(10(#)(14(13)(#))))	да	нет	да
7(3(2(1))(5(4)(6)))(9(8))	да	нет	да

Выводы.

В ходе выполнения лабораторной работы была разработана программа, осуществляющая считывания бинарного дерева, выбранным пользователем способом, и выводящая его графическое представление. Также в программе реализован функционал визуализации проверки полученного дерева на принадлежность к BST и Binary Heap.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: treeCheckingFunctions.cpp

```
#include ”basicheaders.h”
#include ”tree.h”
#include ”customvector.h”

namespace lrstruct {
    void Node::changeLoopStepSwitcherState() {
            stepLoopSwitcher = false;
    }

    void Node::setStepByStepFlag() {
        isStepByStepMode = true;
    }

    void Node::loopLatency() {
        qDebug() << ”Loop latency for step-by-step started” \
        << endl;
        for( ; ; ) {
            QApplication::processEvents();
            if(stepLoopSwitcher == false) break;
        }
        stepLoopSwitcher = true;
    }

    bool Node::isValidBst(Node* root) {
        if(isStepByStepMode) { stepLoopSwitcher = true; }
        qDebug() << ”Checking for valid BST started” << endl;
        if(!root) return true;

        std::stack<Node*> nodeStack;
        Node* current = root;

        int curr_val, pre_val = INT_MIN;
        int first_compare = 1;          // in first compare

        while(!nodeStack.empty() || current) {
            if(current) {
                qDebug() << ”Val: ” << current->_data.val << \
                ” Mode: ” << isStepByStepMode << endl;
```

```
        }
        if(current) {
            if(isStepByStepMode == true)
                emit root->drawCurrNode(current, 1);
            nodeStack.push(current);
            current = current->_left;
        } else {
            current = nodeStack.top();
            if(isStepByStepMode == true)
                emit root->drawCurrNode(current, 2);
            curr_val = current->_data.val;

            if(curr_val <= pre_val && first_compare == 0) {
                if(isStepByStepMode == true)
                    emit root->drawCurrNode(current, 3);
                root->changeLoopStepSwitcherState();
                return false;
            }

            first_compare = 0;      // after first compare
            pre_val = curr_val;
            nodeStack.pop();
            current = current->_right;
        }
        if(isStepByStepMode == true)
            loopLatency();
    }

    qDebug() << "End of checking for valid BST tree" << endl;
    root->changeLoopStepSwitcherState();
    return true;
}

bool Node::isValidBinHeap(Node* root) {
    // create an empty queue and enqueue root node
    std::queue<Node*> queue;
    queue.push(root);

    // take a boolean flag which becomes true when an empty left
    // or right child is seen for a node
    bool nullseen = false;
```

```
// run till queue is not empty
while (queue.size())
{
    // process front node in the queue
    Node* current = queue.front();
    queue.pop();
    if(isStepByStepMode == true)
        emit root->drawCurrNode(current, 1);

    // left child is non-empty
    if (current->_left)
    {
        if(isStepByStepMode == true)
            loopLatency();
        // if either heap-property is violated
        if (nullseen || current->_left->_data.val >= \
        current->_data.val) {
            if(isStepByStepMode == true)
                emit root->drawCurrNode(current->_left, 3);
            root->changeLoopStepSwitcherState();
            return false;
        }

        // enqueue left child
        queue.push(current->_left);
    }
    // left child is empty
    else {
        nullseen = true;
    }

    // right child is non-empty
    if (current->_right)
    {
        // if either heap-property is violated
        if (nullseen || current->_right->_data.val >= \
        current->_data.val) {
            if(isStepByStepMode == true)
                emit root->drawCurrNode(current->_right, 3);
            root->changeLoopStepSwitcherState();
            return false;
        }
```

```
                // enqueue left child
                queue.push(current->_right);
            }
            // right child is empty
            else {
                nullseen = true;
            }

            if(isStepByStepMode == true)
                loopLatency();
        }

        // we reach here only when the given binary tree is a min-heap
        root->changeLoopStepSwitcherState();
        return true;
    }
}
```