МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья — вставка и исключение

Студент гр. 7303 _____ Юсковец А.В.

Преподаватель _____ Балтрашевич Т.А.

Санкт-Петербург
2018

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Юсковец А.В.

Группа 7303

Тема работы:

АВЛ-деревья — вставка и исключение

Содержание пояснительной записки:

Введение

Содержание

Разработка класса вершины АВЛ-дерева

Разработка класса АВЛ-дерева

QT

Заключение

Приложение А. Пользовательский интерфейс

Приложение Б. Пример работы программы

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Юсковец А.В.

Преподаватель _____ Балтрашевич Т.А.

АННОТАЦИЯ

В данной работе разработан класс АВЛ-дерева на языке C++ в связке с классами и функциями фреймворка QT.

При разработке пользовательского интерфейса использовался фреймворк QT и поставляемый вместе со средой разработки QTCreator программа ElasticNodes.

Как и требовалось в задании, было реализовано:

- демонстрация
- вставка
- удаление

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В курсовой работе была поставлена задача разработать АВЛ-дерево (реализовать вставку, удаление) и каким-либо способом продемонстрировать проделанную работу с помощью законченного desktop-приложения со своим пользовательским интерфейсом.

1. РАЗРАБОТКА КЛАССА ВЕРШИНЫ АВЛ-ДЕРЕВА

Помимо реализованных методов класса Node было разработано несколько пользовательских, а именно:

- `void updateHeight()`
- `void removeEdge(Edge *edge)`
- `void removeAllEdges()`
- `Edge* findEdge(Node* dest)`
- `void display_tree(Node* parent_pos, double offset)`
- `void mousePressEvent(QGraphicsSceneMouseEvent* event)`

А также был добавлен один сигнал:

- `void rightClicked(Node* node)`

1.1. Метод `updateHeight`

```
void Node::updateHeight() {
    int lHeight = 0;
    int rHeight = 0;
    if (left != nullptr)
        lHeight = left->height;
    if (right != nullptr)
        rHeight = right->height;
    int max = (lHeight > rHeight) ? lHeight : rHeight;
    height = max + 1;
}
```

Данный метод нужен для пересчета высоты поддерева с корнем в данной вершине.

1.2. Метод `removeEdge`

```
void Node::removeEdge(Edge *edge) {
    if (edge) {
        graph->_scene->removeItem(edge);
        edgeList.removeOne(edge);
    }
}
```

Данный метод удаляет переданное ребро у вершины, а также стирает его с графической сцены.

1.3. Метод `removeAllEdges`

```
void Node::removeAllEdges() {
    for (Edge* edge : edgeList)
        removeEdge(edge);
    if (left)  left->removeAllEdges();
    if (right) right->removeAllEdges();
}
```

Данный метод удаляет все ребра, которые имеет данная вершина.

1.4. Метод `findEdge`

```
Edge* Node::findEdge(Node *dest) {
    for (Edge* edge : edgeList) {
        if (edge->destNode() == dest || edge->sourceNode() == dest ||
                edge->destNode() == this || edge->sourceNode() == this)
            return edge;
    }

    return nullptr;
}
```

Данный метод находит ребро, которое соединяет данную вершину с переданной, в случае неудачи возвращается нулевой указатель.

1.5. Метод `display_tree`

```
void Node::display_tree(Node* parent, double offset) {
    if (parent) {
        QPointF parent_pos = parent->pos();
        if (value < parent->value)
            setPos(parent_pos.x() - offset, parent_pos.y() + DIAMETR*3);
        else
            setPos(parent_pos.x() + offset, parent_pos.y() + DIAMETR*3);
    }
    else
        setPos(300, DIAMETR + DIAMETR/3);

    update();

    if (left) {
        graph->_scene->addItem(new Edge(this, left));
        left->display_tree(this, offset/2);
    }
    if (right) {
        graph->_scene->addItem(new Edge(this, right));
        right->display_tree(this, offset/2);
    }
}
```

Данный метод отображает поддерево на графической сцене опираясь на координаты родительской вершины и отступ по оси абсцисс. С каждым рекурсивным вызовом отступ по оси абсцисс делится на два, а по оси ординат отступ константный.

1.6. Метод *mousePressEvent*

```
void Node::mousePressEvent(QGraphicsSceneMouseEvent* event) {
    if(event->button() == Qt::RightButton) {
        emit rightClicked(this);
    }
}
```

Данный метод используется при удалении конкретной вершины на графической сцене. При нажатии правой кнопкой мыши по вершине — испускается сигнал `rightClicked`.

2. РАЗРАБОТКА КЛАССА АВЛ-ДЕРЕВА

Было разработано несколько методов для поддержания корректного состояния и удобной работы с деревом:

публичные методы:

- `void insert(int val)`
- `void remove(int val)`
- `void rebalance(Node *&node)`

приватные методы:

- `void insert(int d, Node *&node)`
- `void remove(int val, Node*& node)`
- `int min(Node*& node)`
- `void destroy(Node *&node)`
- `int getDiff(Node *node)`
- `void updateAllHeights(Node*& node)`

а также несколько методов АВЛ-дерева:

- `Node* rotateLeft(Node *&node)`
- `Node* rotateRight(Node *&node)`
- `Node* rotateRightLeft(Node *&node)`
- `Node* rotateLeftRight(Node *&node)`

2.1. Метод *insert(int val)*

```
void insert(int val) {
    insert(val, root);
    updateAllHeights(root);
    root->removeAllEdges();
    root->display_tree(nullptr, 300);
}
```

Данная функция просто вызывает приватный метод вставки, в котором заключена вся логика, затем перерисовывается дерево.

2.2. Метод *remove*

```
void remove(int val) {
    remove(val, root);
    updateAllHeights(root);
    if (root) {
        root->removeAllEdges();
        root->display_tree(nullptr, 300);
    }
}
```

Аналогично методу insert.

2.3. Метод *rebalance*

```
void rebalance(Node *&node) {
    int hDiff = getDiff(node);
    if (hDiff > 1){
        if (getDiff(node->left) > 0) node = rotateRight(node);
        else node = rotateLeftRight(node);
    } else if(hDiff < -1) {
        if (getDiff(node->right) < 0) node = rotateLeft(node);
        else node = rotateRightLeft(node);
    }
}
```

Метод, отвечающий за балансирование дерева. Метод `getDiff` будет рассмотрен далее.

2.4. Метод *insert(int d, Node *&node)*

```
void insert(int d, Node *&node){
    if (node == nullptr){
        node = new Node(d, gw);
        gw->_scene->addItem(node);
        node->updateHeight();
    }
    else {
        if (d < node->value){
            insert(d, node->left);
            node->updateHeight();
            rebalance(node);
        }
        else if (d > node->value){
            insert(d, node->right);
            node->updateHeight();
            rebalance(node);
        }
    }
}
```

Метод принимает значение для новой вершины и корень поддерева, в которое эту вершину нужно вставить. В случае, если мы дошли до листа, создается новый экземпляр Node и отрисовывается на сцене. В соответсвии со свойством двоичного дерева поиска вершина вставляется в левое или правое поддерево.

2.5. Метод *void remove(int val, Node*& node)*

```
void remove(int val, Node*& node) {
    if (node) {
        if (val < node->value) {
            remove(val, node->left);
            node->updateHeight();
            rebalance(node);
        }
        else if (val > node->value) {
            remove(val, node->right);
```

```
            node->updateHeight();
            rebalance(node);
        }
        else {
            if (node->left && node->right) {
                Node* current_node = node;
                int min_node_value = min(node->right);

                remove(min_node_value, node);
                current_node->value = min_node_value;

                current_node->updateHeight();
                rebalance(current_node);
            }
            else if (node->left || node->right) {
                Node* node_to_remove = node;
                if (node->left) {
                    node = node_to_remove->left;
                    node_to_remove->left = nullptr;
                }
                else if (node->right) {
                    node = node_to_remove->right;
                    node_to_remove->right = nullptr;
                }

                node->updateHeight();
                rebalance(node);
                destroy(node_to_remove);
            }
            else {
                destroy(node);
            }
        }
    }
}
```

Удаление происходит в соответсвии со свойством довичного дерева поиска. В случае, если нашлась вершина, алгоритм ветвится на три части:

1. Если удаляем лист — достаточно просто удалить лист методом `destroy`.
2. Если у удаляемой вершины 1 ребенок — удаляется лист и в эту вершину помещается значение ребенка.
3. Если у удаляемой вершины 2 ребенка — то значение этой вершини меняется со значение наименьшего листа в правом поддереве, затем удлаяется это лист и дерево начанает балансироваться.

2.6. Метод *min*

```
int min(Node*& node) {
    if (node)
        return node->left ? min(node->left) : node->value;
}
```

Возвращается самый левый лист в дереве.

2.7. Метод *destroy*

```
void destroy(Node *&node) {
    if (node != nullptr){
       destroy(node->left);
       destroy(node->right);

       gw->_scene->removeItem(node);
       delete node;
       node = nullptr;
    }
}
```

Данный метод рекурсивно удаляет все поддерево из динамической памяти, а также с графической сцены.

2.8. Метод *getDiff*

```
int getDiff(Node *node) {
    int lHeight = 0;
    int rHeight = 0;
    if (node->left != nullptr)
        lHeight = node->left->height;
    if (node->right != nullptr)
        rHeight = node->right->height;
    return lHeight - rHeight;
}
```

Данный метод возвращает разницу в высоте левого и правого поддерева.

2.9. Метод *updateAllHeights*

```
void updateAllHeights(Node*& node) {
    if (node) {
        updateAllHeights(node->left);
        updateAllHeights(node->right);
        node->updateHeight();
        rebalance(node);
    }
}
```

Данная функция обходит все дерево и балансирует его начиная с листьев.

2.10. Метод *rotateLeft*

```
Node* rotateLeft(Node *&node) {
     Node* temp = node->right;
     node->right = temp->left;
     temp->left = node;

     node->updateHeight();
     temp->updateHeight();

     return temp;
}
```

Данный метод представляет собой левое вращение.

2.11. Метод *rotateRight*

```
Node* rotateLeft(Node *&node) {
     Node* temp = node->left;
     node->left = temp->right;
     temp->right = node;

     node->updateHeight();
     temp->updateHeight();

     return temp;
}
```

Данный метод представляет собой правое вращение.

2.12. Метод *rotateRightLeft*

```
Node* rotateRightLeft(Node *&node) {
     Node* temp = node->right;
     node->right = rotateRight(temp);
     return rotateLeft(node);
}
```

Данный метод представляет собой правое-левое вращение.

2.13. Метод *rotateLeftRight*

```
Node* rotateLeftRight(Node *&node) {
     Node* temp = node->right;
     node->right = rotateRight(temp);
     return rotateLeft(node);
}
```

Данный метод представляет собой левое-правое вращение.

3. QT

Интерфейсы основного окна и дилогового осна для вставки новых элементов приведены в приложении А.

В исходном коде классов Node, Edge и GraphWidget были удалены функции, отвечающие за динамическую отрисовку. Причиной этому послужило плохая наглядность дерева, при движении вершин. Статическую картинку оказлось намного проще анализировать.

3.2. Разработка `InsertDialog`

Для реализации вставки новых элементов в дерево было реализовано диалоговое окно со слотом `void on_insertButton_clicked()` и сигналом `void insertSignal(int value)`.

Исходный код слота:

```
void InsertDialog::on_insertButton_clicked() {
    emit insertSignal(ui->spinBox->value());
}
```

При нажатии на кнопку *insert* испускается сигнал со значение введенным в поле для ввода значения очередной вершины. Данный сигнал затем обрабатывается в GraphWidget.

3.2. Вызов `InsertDialog` из `MainWindow`

```
void MainWindow::on_actionInsert_triggered() {
    InsertDialog* insertDialog = new InsertDialog(this);
    connect(insertDialog, SIGNAL(insertSignal(int)),
            graph, SLOT(insertNode(int)));

    insertDialog->show();
}
```

Создается экземпляр InsertDialog, затем сигнал `insertSignal` связывается со слотом GraphWidget `insertNode`.

3.3. Дополнения `GraphWidget`. Слот `insertNode`

```
void GraphWidget::insertNode(int value) {
    tree.insert(value);
}
```

Данный слот получает `value` из InsertDialog и просто вставляет элемент в дерево.

ЗАКЛЮЧЕНИЕ

Было проведено много работы по отладке разработанного класса АВЛ-дерева, собрано много материалов и учтено множество ошибок. Проделанная работа полностью соответсвтует поставленному заданию.

При разработке был получен опыт работы с готовым кодом (ElasticNodes). Также получен опыт интеграции собственного программного кода в уже готовую инфраструктуру.

ПРИЛОЖЕНИЕ А
ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

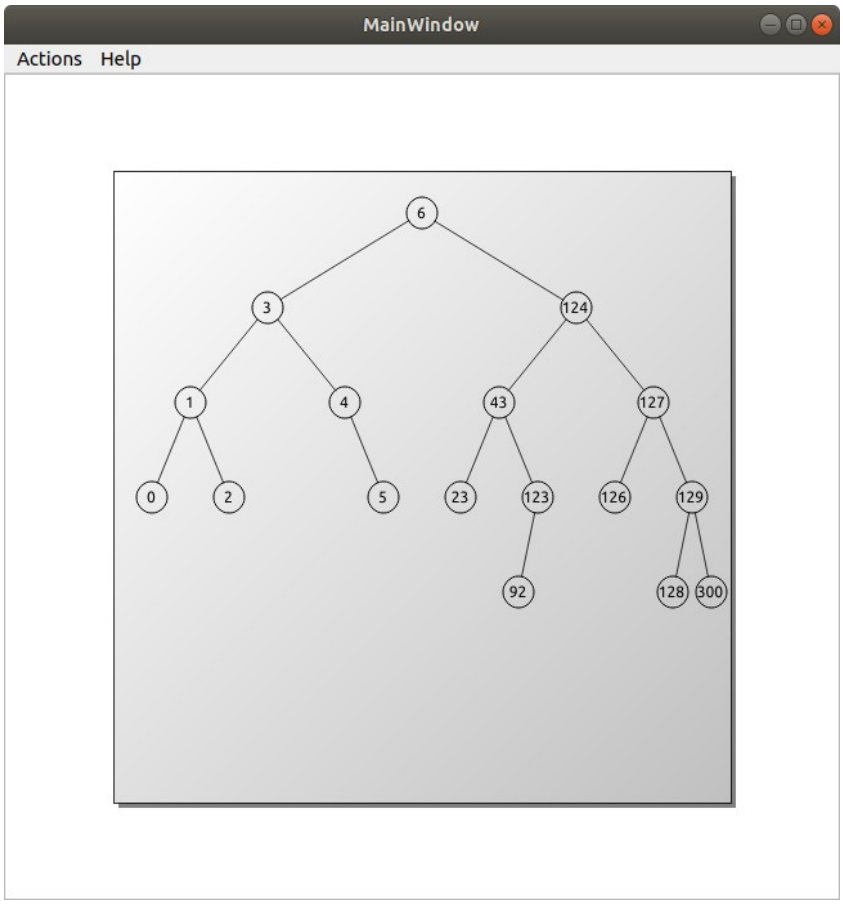


Рисунок 1 — Основное окно

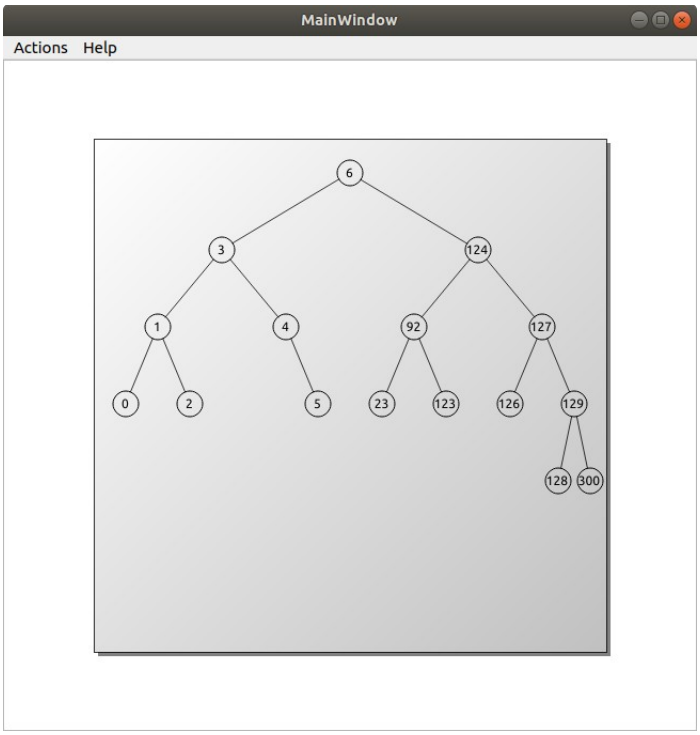
Рисунок 2 — Диалоговое окно вставки

ПРИЛОЖЕНИЕ Б
ПРИМЕР РАБОТЫ ПРОГРАММЫ

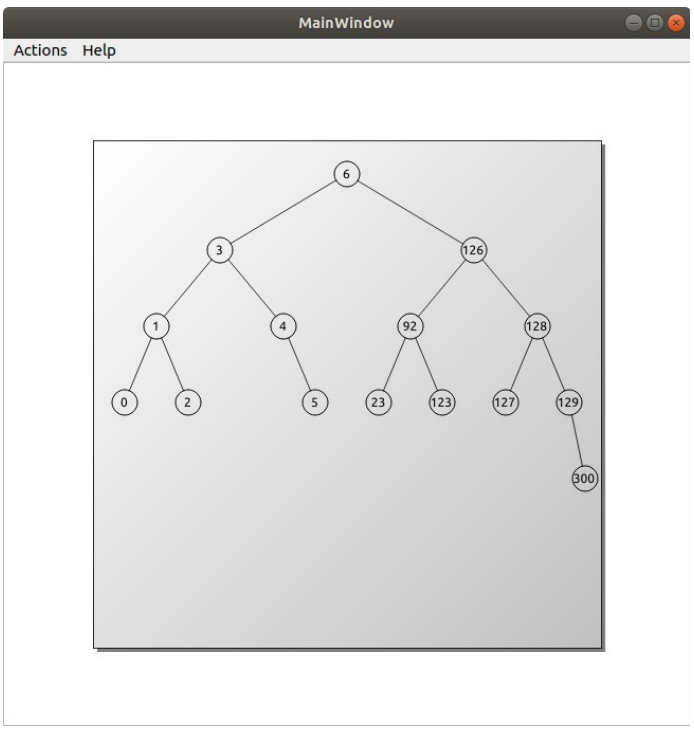
В данном приложении будет продемонстрировано вставка и удаление нескольких вершин.

В начале работы дерево выглядит как на рис. 1.

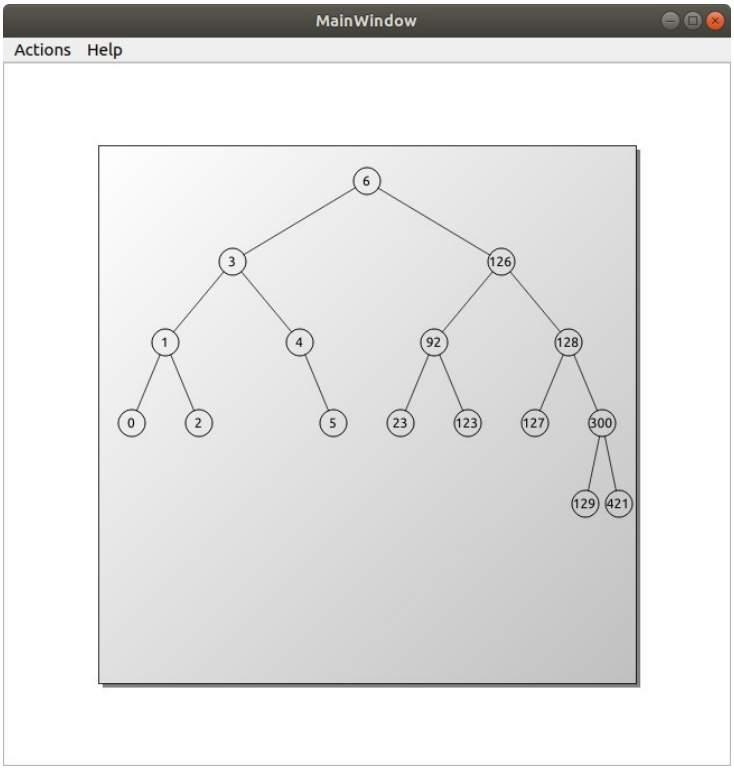
Попробуем удалить вершину *43*. Как и ожидалось лист *92* и вершина *43* поменялись местами.



После удаления вершины *124* дерево балансируется:



Теперь вставим вершину, причем подберем значение так, чтобы разбалансировать дерево, например *421*. Произойдет разбалансировка в правом поддереве.



И еще раз удалим вершину *127*, чтобы произошла разбалансировка.