```
int main( int argc, char ** argv )
{
    Q_INIT_RESOURCE(textedit);
    QApplication a( argc, argv );
    TextEdit mw;
    mw.resize( 700, 800 );
    mw.show();
    return a.exec();
}
```

`Q_INIT_RESOURCE(textedit);` – инициализируются ресурсы приложения через файл textedit.qrc

`Q_INIT_RESOURCE` гарантирует, что ресурсы связаны с приложением в случае статического связывания, или принудительно загружает библиотеку в случае динамического связывания

`QApplication a( argc, argv );`- создается экземпляр класса `QApplication`. `QApplication` управляет логикой приложения с графическим интерфейсом, содержит основной цикл обработки событий. Для любого приложения с графическим интерфейсом существует единственный экземпляр `QApplication`. `QApplication` наследуется от `QCoreApplication`. `QApplication` используется для получения событий, на которые приложение должно своевременно и правильно реагировать.

`TextEdit mw;` - создается экземпляр класса `TextEdit`.

`mw.resize( 700, 800 );` - устанавливается размер виджета

`mw.show();` - вывод виджета и дочерних виджетов

`return a.exec();`- запуск цикла обработки сообщений, который закончится, когда будет вызван метод `QCoreApplication::exit()`, либо окно будет закрыто.

Класс `TextEdit` наследуется от `QMainWindow`. `QMainWindow` – этот класс представляет главное окно приложения, имеет собственный компоновщик, в который можно добавлять виджеты, необходимые графическому приложению, - меню, панель инструментов, строки состояния. Наша работа основана на обработке сигналов и слотов, поэтому чтобы MOC(мета-компилятор) смог распознать такие классы, этот класс `TextEdit` должен содержать макрос `Q_OBJECT`:

```
/*textedit.h*/
class TextEdit : public QMainWindow
{
    Q_OBJECT
…
}
```

Рассмотрим конструктор TextEdit:

```
TextEdit::TextEdit(QWidget *parent):QMainWindow(parent)
```

```
{
…
    QString initialFile = ":/example.html";
    const QStringList args = QCoreApplication::arguments();
    if (args.count() == 2)
        initialFile = args.at(1);
    if (!load(initialFile))
        fileNew();
}
```

Внутри конструктора идет инициализация входных параметров: меню, панели инструментов, подключение слотов и сигналов.

Параметры из командной строки мы получаем с помощью статической функции `static QStringList arguments()`:

```
QStringList QCoreApplication::arguments()
{
    QStringList list;
    if (!self) {
        qWarning("QCoreApplication::arguments: Please instantiate the
QApplication object first");
        return list;
    }
    QString cmdline = QString::fromWCharArray(GetCommandLine());
…
    list = qWinCmdArgs(cmdline);
if (self->d_func()->application_type) { // GUI app? Skip known - see
qapplication.cpp
        QStringList stripped;
        for (int a = 0; a < list.count(); ++a) {
            QString arg = list.at(a);
            QByteArray l1arg = arg.toLatin1();
            if (l1arg == "-qdevel" ||
                l1arg == "-qdebug" ||
                l1arg == "-reverse" ||
                l1arg == "-stylesheet" ||
                l1arg == "-widgetcount")
                ;
            else if (l1arg.startsWith("-style=") ||
                     l1arg.startsWith("-qmljsdebugger="))
                ;
            else if (l1arg == "-style" ||
                     l1arg == "-session" ||
                     l1arg == "-graphicssystem" ||
                     l1arg == "-testability")
                ++a;
            else
                stripped += arg;
        }
        list = stripped;
    }
```

```
…
    return list;
}

static inline QStringList qWinCmdArgs(QString cmdLine) // not const-
ref: this might be modified
{
    QStringList args;
    int argc = 0;
    QVector<wchar_t*> argv = qWinCmdLine<wchar_t>((wchar_t
*)cmdLine.utf16(), cmdLine.length(), argc);
    for (int a = 0; a < argc; ++a) {
        args << QString::fromWCharArray(argv[a]);
    }
    return args;
}
```

С помощью конструктора `QCoreApplication::QCoreApplication(int &argc, char **argv): QObject(*new QCoreApplicationPrivate(argc, argv))` мы запоминаем `argc и argv` в статические поля класса `QCoreApplication`:

```
#ifdef QT_DEPRECATED
    QT_DEPRECATED static int argc();
    QT_DEPRECATED static char **argv();
#endif
```

После с помощью этой функции `QString cmdline = QString::fromWCharArray(GetCommandLine());` мы считываем командную строку в `cmdline`

Дальше идет проверка на существование файла, его открытие и считывание. Рассмотрим функцию `load(initialFile)`:

```
bool TextEdit::load(const QString &f)

{

    if (!QFile::exists(f))

        return false;

    QFile file(f);

    if (!file.open(QFile::ReadOnly))

        return false;

    QByteArray data = file.readAll();

    QTextCodec *codec = Qt::codecForHtml(data);

    QString str = codec->toUnicode(data);

    if (Qt::mightBeRichText(str)) {

        textEdit->setHtml(str);

    } else {

        str = QString::fromLocal8Bit(data);

        textEdit->setPlainText(str);

    }

    setCurrentFileName(f);
```

```
    return true;
}
```

В этой функции происходит проверка на существование файла. Потом идет открытие файла и считывание данных в массив байтов `QByteArray data`

Далее идет определение кодировки файла с помощью команды `codecForHtml(data)`:

```
QTextCodec *Qt::codecForHtml(const QByteArray &ba)
{
    return QTextCodec::codecForHtml(ba);
}
QTextCodec *QTextCodec::codecForHtml(const QByteArray &ba)
{
    return codecForHtml(ba, QTextCodec::codecForMib(/*Latin 1*/ 4));
}

QTextCodec *QTextCodec::codecForHtml(const QByteArray &ba, QTextCodec
*defaultCodec)
{
    // determine charset
    int pos;
    QTextCodec *c = 0;
    c = QTextCodec::codecForUtfText(ba, c);
    if (!c) {
        QByteArray header = ba.left(512).toLower();
        if ((pos = header.indexOf("http-equiv=")) != -1) {
            if ((pos = header.lastIndexOf("meta ", pos)) != -1) {
                pos = header.indexOf("charset=", pos) +
int(strlen("charset="));
                if (pos != -1) {
                    int pos2 = header.indexOf('\"', pos+1);
                    QByteArray cs = header.mid(pos, pos2-pos);
                    //            qDebug("found charset: %s",
cs.data());
                    c = QTextCodec::codecForName(cs);
                }
            }
        }
    }
    if (!c)
        c = defaultCodec;
    return c;
}
```

Вначале создается `QTextCodec *c = 0`, выполняется команда c = `QTextCodec::codecForUtfText(ba,c)`, где `ba` – это весь переданный файл в байтовом массиве. Внутри этой функции определяется кодировка с помощью BOM – специальная последовательность байт. Т.е. если наш файл UTF (он UTF-8), то возвращаемся в `codecForHtml`

Если это не UTF, то мы считываем первые 512 байт из байтового массива `QByteArray header = ba.left(512).toLower();` Потом в этих байтах ищется конструкции “http-equiv=” и “charset=”, по которой и определяем кодировку

Если не распознали никакую кодировку, то берем `defaultCodec`, который определяется с помощью функции `QTextCodec::codecForMib(/*Latin 1*/ 4);`

`QString str = codec->toUnicode(data);` - с помощью полученного указателя `QTextCodec *codec` наш файл переводится в формат Unicode и записывается в строку `str`

`if (Qt::mightBeRichText(str))` – проверка: является ли наш файл HTML-документом. Если да, то переходим к `textEdit->setHtml(str)`

Иначе выполняется:

```
    str = QString::fromLocal8Bit(data);

        textEdit->setPlainText(str);
```

У нас это условие выполняется, значит переходим к `textEdit->setHtml(str)`:

```
void QTextEdit::setHtml(const QString &text)
{
    Q_D(QTextEdit);
    d->control->setHtml(text);
    d->preferRichText = true;
}
```

Внутри `setHtml(str)` сначала выполняется макрос `Q_D(QTextEdit)`. С его помощью получаем указатель на приватного двойника передаваемого класса `QTextEdit`. После вызывается `d->control->setHtml(text)`:

```
void QTextControl::setHtml(const QString &text)
{
    Q_D(QTextControl);
    d->setContent(Qt::RichText, text);
}
```

Вызывается `d->setContent(Qt::RichText, text)`:

```
void QTextControlPrivate::setContent(Qt::TextFormat format, const
QString &text, QTextDocument *document)
{
    Q_Q(QTextControl);
    …
    if (!text.isEmpty()) {
        cursor = QTextCursor();
        if (format == Qt::PlainText) {
            …
        } else {
#ifndef QT_NO_TEXTHTMLPARSER
            doc->setHtml(text);
#else
            doc->setPlainText(text);
#endif
            doc->setUndoRedoEnabled(false);
        }
        cursor = QTextCursor(doc);
    } else if (clearDocument) {
        doc->clear();
    }
    …
}
```

В этой функции заполняются поля класса `QTextControl`. Потом переходим в `doc->setHtml(text)`, которая вызывается уже у класса `QTextDocument`:

```
void QTextDocument::setHtml(const QString &html)
{
    Q_D(QTextDocument);
    bool previousState = d->isUndoRedoEnabled();
    d->enableUndoRedo(false);
    d->beginEditBlock();
```

```
    d->clear();
    QTextHtmlImporter(this, html,
QTextHtmlImporter::ImportToDocument).import();
    d->endEditBlock();
    d->enableUndoRedo(previousState);
}
```

Внутри опять получаем указатель на приватного двойника класса `QTextDocument`

`QTextHtmlImporter(this, html, QTextHtmlImporter::ImportToDocument).import();` – в конструкторе класса `QTextHtmlImporter` будет создано DOM-дерево (Document Object Model), которое после преобразуется в два красно-черных дерева в методе `import()`

Построение DOM-дерева:

Любой документ известной структуры с помощью DOM может быть представлен в виде дерева узлов, каждый узел представляет собой объект(элемент, атрибут, текст). Узлы связаны между собой отношениями «родительский-дочерний».

Мы используем DOM-дерево, потому что с его помощью можем представить наш документы в виде иерархической структуры. Представление DOM состоит из структурированной группы узлов и объектов, которые имеют свойства и методы.

Основой HTML-документа являются теги, которые в свою очередь являются объектами. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. У текстового узла не может быть потомков, т.е. он находится всегда на самом нижнем уровне. Пробелы и переводы строки – это такие же символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM.

Зайдем в конструктор `QTextHtmlImporter`:

```
QTextHtmlImporter::QTextHtmlImporter(QTextDocument *_doc, const
QString &_html, ImportMode mode, const QTextDocument
*resourceProvider)
    : indent(0), compressNextWhitespace(PreserveWhiteSpace),
doc(_doc), importMode(mode)
{
    cursor = QTextCursor(doc);
    wsm = QTextHtmlParserNode::WhiteSpaceNormal;
    QString html = _html;
    const int startFragmentPos = html.indexOf(QLatin1String("<!--
StartFragment-->"));
    if (startFragmentPos != -1) {
        …
    }
    parse(html, resourceProvider ? resourceProvider : doc);
//        dumpHtml();
}
```

Здесь создается курсор `cursor = QTextCursor(doc);` Курсор – это интерфейс для доступа и изменения содержимого `QTextDocument`. После переданная строка записывается в другую строку: `QString html = _html;` Т.к. получается, что `startFragmentPos = -1`, то переходим в метод `parse(html, resourceProvider ? resourceProvider : doc)`, передавая строку `html` и объект `QTextDocument doc`

Переходим к классу `QTextHtmlParser`. В его protected доступе лежат поля:

```
class QTextHtmlParser
{
…
protected:
    QTextHtmlParserNode *newNode(int parent);
    QVector<QTextHtmlParserNode> nodes;
    QString txt;
    int pos, len;
    bool textEditMode;
    void parse();
        void parseTag();
…
    const QTextDocument *resourceProvider;
};
```

`nodes` лежит в `QVector`, вектор составлен из `QTextHtmlParserNode`.
`QTextHtmlParserNode` - это структура, в которой описано, как нужно представлять дерево

А мы как раз и хотим представить нашу строку в виде DOM-дерева так, чтобы в узлах дерева были записаны теги, а в листьях был записан конкретный текст.

Перейдем в метод `parse` класса `QTextHtmlParser`. Здесь заполняются поля класса:

```
void QTextHtmlParser::parse(const QString &text, const QTextDocument
*_resourceProvider)
{
    nodes.clear();
    nodes.resize(1);
    txt = text;
    pos = 0;
    len = txt.length();
    textEditMode = false;
    resourceProvider = _resourceProvider;
    parse();
    //dumpHtml();
}
```

Здесь создается первый узел в векторе `nodes`, поле `txt` теперь хранит полученный текст и устанавливается позиция `pos` на начало текста и его длина `len`. Рассмотрим метод `parse()`:

```
void QTextHtmlParser::parse()
{
    while (pos < len) {
        QChar c = txt.at(pos++);
        if (c == QLatin1Char('<')) {
            parseTag();
        } else if (c == QLatin1Char('&')) {
            nodes.last().text += parseEntity();
        } else {
            nodes.last().text += c;
        }
    }
}
```

Здесь бежим посимвольно по всей строке `txt` (исходный текст из файла) и сразу меняем позицию `pos`. Теперь, смотря какой символ получили, у нас выполнятся что-то из трех: либо разбор тега, либо разбор сущности, либо просто записываем текст.

Например, получили открывающий тег, тогда переходим в parseTag:

```
void QTextHtmlParser::parseTag()
```

```
{
    eatSpace();
    // handle comments and other exclamation mark declarations
    if (hasPrefix(QLatin1Char('!'))) {
        …
    }
    // if close tag just close
    if (hasPrefix(QLatin1Char('/'))) {
        …
       parseCloseTag();
    }
    …
    QTextHtmlParserNode *node = newNode(p);
    // parse tag name
          node->tag = parseWord().toLower();
          …
}
```

В `parseTag()` несколько вариантов: либо конструкция с «!», либо закрывающий тег, либо обычный

Например, обычный тег:

Создается новый узел, записывается имя тега:

```
QTextHtmlParserNode *node = newNode(p);

    // parse tag name

    node->tag = parseWord().toLower();
```

Потом если есть атрибуты, то они тоже записываются в соответствующие поля

Например, наш символ не является тегом и не является сущностью, получается это текст. Тогда в последний узел текст записывается посимвольно, пока снова не встретится тег

Теперь, например, тег оказался закрывающимся. Тогда вызывается `parseCloseTag()`

Здесь тег не записывается в узел, но узел создается

После того, как построили DOM-дерево, переходим в import(), т.е. начинаем стоить дерево фрагментов и дерево блоков(потому что хотим получить текст без тегов):

В поле класса `QTextDocumentPrivate` объявлены переменные

```
    BlockMap blocks;

    FragmentMap fragments;
```

Каждая, из которых является красно-черным деревом:

Они удобны, потому что являются самобалансирующимися из-за наличия цвета. Также такие операции как поиск, добавление, удаление выполняются достаточно быстро, а наши данные представлены в иерархической структуре

Свойства, которые должны всегда выполняться:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья — чёрные и не содержат данных.
4. Оба потомка каждого красного узла — чёрные.

5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

```
typedef QFragmentMap<QTextBlockData> BlockMap;

typedef QFragmentMap<QTextFragmentData> FragmentMap;
```

Блок – представление конкретной части документа, например, абзац текста.

Каждый текстовый блок расположен в определенной позиции в документе

Фрагмент – текст, в котором изменяется формат символа

Например, изменение стиля некоторого текста в середине предложения приведет к тому, что фрагмент будет разбит на три отдельных фрагмента: первый и третий с тем же форматом, что и раньше, а второй с новым стилем. Первый фрагмент будет содержать текст из начала предложения, второй будет содержать текст из середины, а третий - текст из конца предложения.

Также «/n» -это отдельный узел в дереве фрагментов

```
template <int N = 1> (Для блоков N = 3)
class QFragment
{
public:
    quint32 parent;
    quint32 left;
    quint32 right;
    quint32 color;
    quint32 size_left_array[N];
    quint32 size_array[N];
    enum {size_array_max = N };
};
```

Теперь , если выполнено условие `if (processBlockNode() == ContinueWithNextNode)`, то переходим в метод `appendBlock(block, charFmt);`

```
void QTextHtmlImporter::appendBlock(const QTextBlockFormat &format,
QTextCharFormat charFmt)

{

…

cursor.insertBlock(format, charFmt);

}
```

Внутри вызывается `cursor.insertBlock(format, charFmt);`

```
void QTextCursor:: (const QTextBlockFormat &format, const
QTextCharFormat &_charFormat)
{
…
d->insertBlock(format, charFormat);
}
```

Здесь вызывается метод `d->insertBlock(format, charFormat):`

```
void QTextCursorPrivate::insertBlock(const QTextBlockFormat &format,
const QTextCharFormat &charFormat)
{
…
```

```
priv->insertBlock(position, idx, formats->indexForFormat(charFormat));
}
```
Тут выполняются команды получения индексов форматов текста.

Затем выполняется команда `priv->insertBlock(position, idx, formats->indexForFormat(charFormat))`, где priv – это указатель на QTextDocumentPrivate, position – позиция курсора

Переходим в функцию `insertBlock` класса `QTextDocumentPrivate`:

```
int QTextDocumentPrivate::insertBlock(int pos, int blockFormat, int
charFormat, QTextUndoCommand::Operation op)
{
    return insertBlock(QChar::ParagraphSeparator, pos, blockFormat,
charFormat, op);
}
```
Идем дальше:
```
int QTextDocumentPrivate::insertBlock(const QChar &blockSeparator,
                                      int pos, int blockFormat, int
charFormat, QTextUndoCommand::Operation op)
{
…
int strPos = text.length();
    text.append(blockSeparator);
    int ob = blocks.findNode(pos);
…
const int fragment = insert_block(pos, strPos, charFormat,
blockFormat, op, QTextUndoCommand::BlockRemoved);
…
}
```
`strPos` - длина текста, а с помощью `findNode` находим узел красно-черного дерева блоков, соответствующий позиции pos – позиции курсора в документе. Если же такого блока нет, то переходим к созданию нового

Переходим к выполнению команды `const int fragment = insert_block(pos, strPos, charFormat, blockFormat, op, QTextUndoCommand::BlockRemoved)`:

```
int QTextDocumentPrivate::insert_block(int pos, uint strPos, int
format, int blockFormat, QTextUndoCommand::Operation op, int command)
{
    split(pos);
    uint x = fragments.insert_single(pos, 1);
    QTextFragmentData *X = fragments.fragment(x);
    X->format = format;
    X->stringPosition = strPos;
    // no need trying to unite, since paragraph separators are always
in a fragment of their own
    Q_ASSERT(isValidBlockSeparator(text.at(strPos)));
    Q_ASSERT(blocks.length()+1 == fragments.length());
    int block_pos = pos;
    if (blocks.length() && command == QTextUndoCommand::BlockRemoved)
        ++block_pos;
    int size = 1;
    int n = blocks.findNode(block_pos);
    int key = n ? blocks.position(n) : blocks.length();
    Q_ASSERT(n || (!n && block_pos == blocks.length()));
    if (key != block_pos) {
```

```
        Q_ASSERT(key < block_pos);
        int oldSize = blocks.size(n);
        blocks.setSize(n, block_pos-key);
        size += oldSize - (block_pos-key);
    }
    int b = blocks.insert_single(block_pos, size);
    QTextBlockData *B = blocks.fragment(b);
    B->format = blockFormat;
    Q_ASSERT(blocks.length() == fragments.length());
    QTextBlockGroup *group = qobject_cast<QTextBlockGroup
*>(objectForFormat(blockFormat));
    if (group)
        group->blockInserted(QTextBlock(this, b));
    QTextFrame *frame = qobject_cast<QTextFrame
*>(objectForFormat(formats.format(format)));
    if (frame) {
        frame->d_func()->fragmentAdded(text.at(strPos), x);
        framesDirty = true;
    }
    adjustDocumentChangesAndCursors(pos, 1, op);
    return x;
}
void QTextDocumentPrivate::insert_string(int pos, uint strPos, uint
length, int format, QTextUndoCommand::Operation op)
{
    // ##### optimize when only appending to the fragment!
    Q_ASSERT(noBlockInString(text.mid(strPos, length)));
    split(pos);
    uint x = fragments.insert_single(pos, length);
    QTextFragmentData *X = fragments.fragment(x);
    X->format = format;
    X->stringPosition = strPos;
    uint w = fragments.previous(x);
    if (w)
        unite(w);
    int b = blocks.findNode(pos);
    blocks.setSize(b, blocks.size(b)+length);
    Q_ASSERT(blocks.length() == fragments.length());
    QTextFrame *frame = qobject_cast<QTextFrame
*>(objectForFormat(format));
    if (frame) {
        frame->d_func()->fragmentAdded(text.at(strPos), x);
        framesDirty = true;
    }
    adjustDocumentChangesAndCursors(pos, length, op);
}
```

Здесь добавляем новый узел, пытаемся слить с другим.

В split() разбиваем узел дерева фрагментов на два, если это надо:

```
bool QTextDocumentPrivate::split(int pos)
{
    uint x = fragments.findNode(pos);
    if (x) {
        int k = fragments.position(x);
//          qDebug("found fragment with key %d, size_left=%d, size=%d
to split at %d",
//                 k, (*it)->size_left[0], (*it)->size_array[0], pos);
```

```
        if (k != pos)
            Q_ASSERT(k <= pos);
            // need to resize the first fragment and add a new one
            QTextFragmentData *X = fragments.fragment(x);
            int oldsize = X->size_array[0];
            fragments.setSize(x, pos-k);
            uint n = fragments.insert_single(pos, oldsize-(pos-k));
            X = fragments.fragment(x);
            QTextFragmentData *N = fragments.fragment(n);
            N->stringPosition = X->stringPosition + pos-k;
            N->format = X->format;
            return true;
        }
    }
    return false;
}
```

Здесь происходит добавление в дерево фрагментов - фрагментов и в дерево блоков – блоков. Весь текст хранится в поле `QString text` класса `QTextDocumentPrivate`, хранится он там в очищенном от тегов виде. В фрагментах лишь хранятся указатели на начало каждого сегмента, а добавляется это все в красно-черные деревья с помощью DOM – дерева.

В конце выполняется команда `adjustDocumentChangesAndCursors(pos, 1, op);` и мы возвращаемся к методу `appendNodeText()`. Здесь из узла, обрабатывающегося в данный момент, текст проверяется и попадает в переменную `textToInsert`, которая затем выводится на экран.

Дерево фрагментов `typedef QFragmentMap<QTextFragmentData> FragmentMap` содержит:

```
union {
     Header* head;
     Fragment* fragment;
};
class Header {
public:
     quint32 root;
     quint32 tag;
     quint32 freelist;
     quint32 node_count;//количество узлов
     quint32 allocated;
};
```

`template <class Fragment>, в нашем случае это будет QTextFragmentData`

Узел дерева фрагментов, т.е. экземпляр класса `QTextFragmentData` состоит из таких полей: класса `QFragment`(т.к. наследован от него), где у нас лежат родитель, дети, `size_array`-длина фрагмента, `size_left_array` – смещение от начала текста(куда вставили символ), также содержит поля `format` и `stringPosition`– смещение в строке text:

```
class QTextFragmentData : public QFragment<> {
public:
      …
      int format;
      int stringPosition;
};
template <int N = 1>
class QFragment
{
public:
      quint32 parent;
      quint32 left;
      quint32 right;
      quint32 color;
      quint32 size_left_array[N];
      quint32 size_array[N];
      enum {size_array_max = N };
};
```

Рассмотрим пример, когда в первый узел дерева фрагментов “hello “ добавляется на вторую позицию символ “u”

Заходим в метод `insert_string(pos, strPos, length, format, …)`, где у нас pos = 2, strPos = 29, length = 1, format = 0;

Тогда наш узел фрагмента “hello “ разбивается на 2 узла в методе `split()`:

```
uint x = fragments.findNode(pos);

int k = fragments.position(x);

int oldsize = X->size_array[0];

fragments.setSize(x, pos-k);

uint n = fragments.insert_single(pos, oldsize-(pos-k));
```

узел с ключом 0 (“he”):

`X:`

```
      QFragment<1>
            parent = 3;
            left = 0;
            right = 9;
            color = 1;
            size_left_array = 0;
            size_array = 2;
      format = 0;
      stringPosition = 0;
```

узел с ключом 3 (“llo ”):

`N:` (правый ребенок “he”)

```
      QFragment<1>
            parent = 2;
            left = 0;
            right = 0;
```

```
            color = 0;
            size_left_array = 3;
            size_array = 4;
      format = 0;
      stringPosition = 3;
```

После создается новый узел с ключом 2 для символа “u”:

```
uint x = fragments.insert_single(pos, length);

QTextFragmentData* X = fragments.fragment(x);

X->format = format;

X->stringPosition = strPos;

X:

      QFragment<1>
            parent = 3;
            left = 2;
            right = 9;
            color = 1;
            size_left_array = 2;
            size_array = 1;
      format = 0;
      stringPosition = 29;
```

У него правый ребенок это “he”, а левый ребенок это “llo “

Далее пытаемся слить с узлом “he” в методе `unite(w)`:

```
uint w = fragments.previous(x);
```

```
bool QTextDocumentPrivate::unite(uint f)
{
    uint n = fragments.next(f);
    if (!n)
        return false;
    QTextFragmentData *ff = fragments.fragment(f);
    QTextFragmentData *nf = fragments.fragment(n);
    if (nf->format == ff->format && (ff->stringPosition + (int)ff-
>size_array[0] == nf->stringPosition)) {
        if (isValidBlockSeparator(text.at(ff->stringPosition))
            || isValidBlockSeparator(text.at(nf->stringPosition)))
            return false;
        fragments.setSize(f, ff->size_array[0] + nf->size_array[0]);
        fragments.erase_single(n);
        return true;
    }
    return false;
}
```

Не склеивается, потому что в text стоит не рядом
После добавляем в нужный блок длину символа:

```
int b = blocks.findNode(pos);
blocks.setSize(b, blocks.size(b)+length);
```

Пример построения деревьев:

<html>

<body>

hello

cat

text

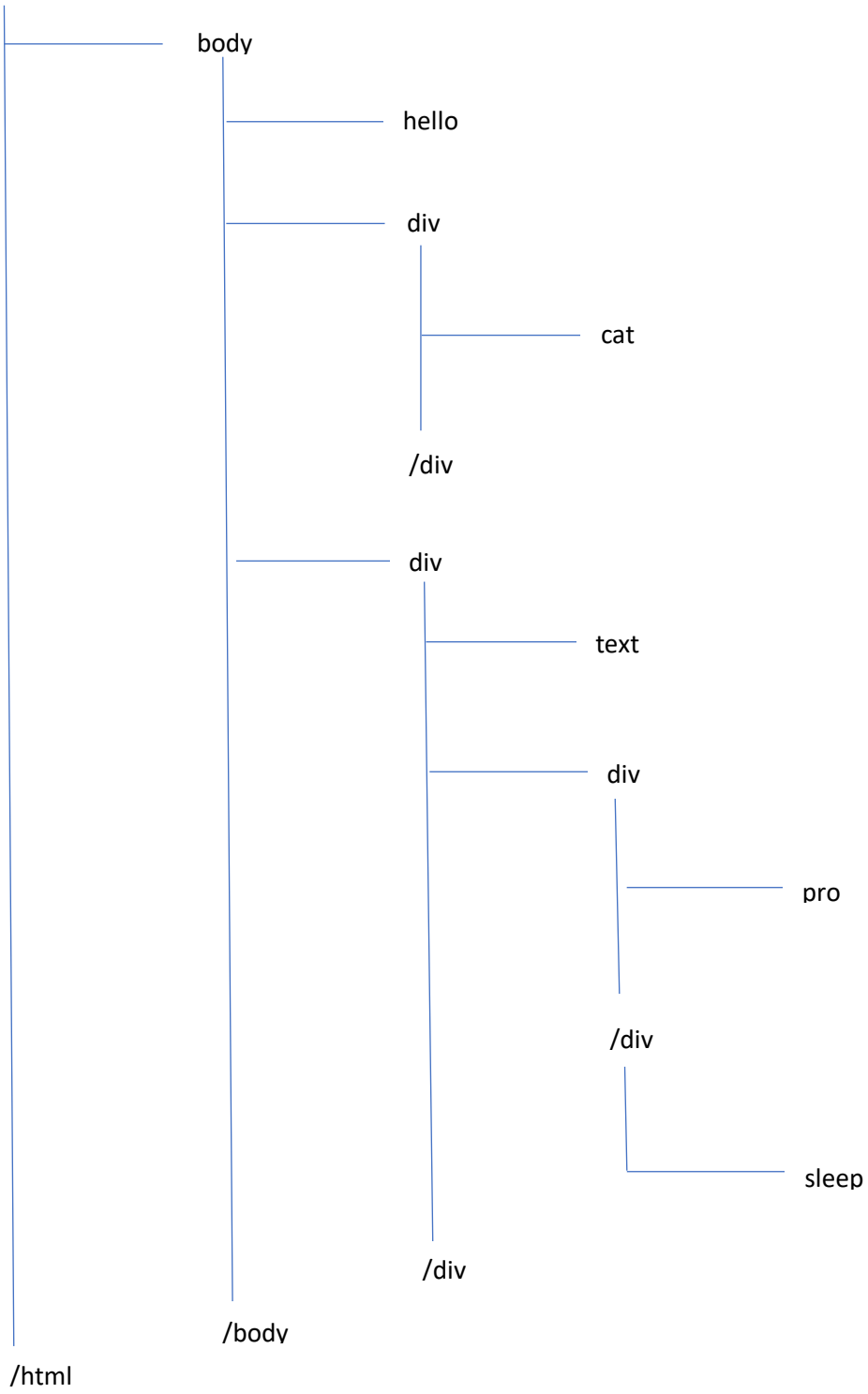
pro

sleep

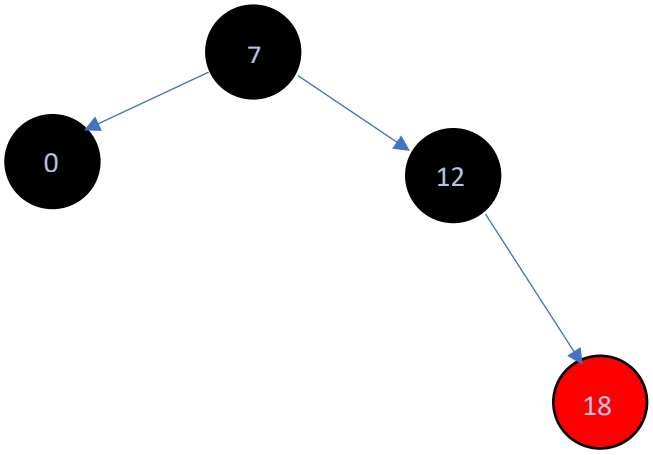
</body>

</html>

DOM-дерево:


html
body
hello
div
cat
/div
div
text
div
pro
/div
sleep
/div
/body
/html

Дерево блоков:



Дерево фрагментов: