ПРИЛОЖЕНИЕ В
(*обязательное*)

Листинг кода

```
//ArchivationThread.h

#ifndef ARCHIVATIONTHREAD_H
#define ARCHIVATIONTHREAD_H

#include <QThread>
#include <QFileInfo>

#include "filetranslator.h"
#include "filecollector.h"
#include "fileentry.h"

class ArchivationThread : public QThread
{
    Q_OBJECT
private:
    QString fileName;
    QFileInfo *activeFile{nullptr};
    QList<FileEntry> fcList;
signals:
    void exception_executed(QString e);
public:
    ArchivationThread() = default;

    void setFileName(QString fileName_){
        fileName=fileName_;
    }

    void setActiveFile(QFileInfo* activeFile_){
        activeFile = activeFile_;
    }

    void clear(){
        fcList.clear();
        fileName.clear();
        activeFile = nullptr;
    }

    QString getFileName(){
        return fileName;
    }

    QList<FileEntry> getFcList(){
        return fcList;
    }
```

```
    QString getArchiveName(){
        return fileName;
    }

    void run();
};

#endif // ARCHIVATIONTHREAD_H

//ArchivationThread.cpp

#include "archivationthread.h"

void ArchivationThread::run(){
    if(fileName.isNull() || !activeFile)
        emit exception_executed("ArchvationThread: fileName or
activeFile is empty");
    try{
        FileTranslator ft;
        ft.openFile(fileName);
        QList<FileEntry> fcList;
        if(activeFile->isDir()){
            FileCollector fc(activeFile->absoluteFilePath());
            fcList = fc.collectFiles();
        }
        else{
            FileEntry temp(activeFile->absoluteFilePath(), "",
activeFile->fileName());
            fcList.push_front(temp);
            FileEntry temp2("<", "", "");
            fcList.push_front(temp2);
        }
        if(QThread::currentThread()->isInterruptionRequested()){
            clear();
            return;
        }
        ft.setAllFiles(fcList);
        ft.translateFiles();
    }
    catch(runtime_error e){
        emit exception_executed(e.what());
    }
}

//bTree.h

#ifndef BTREE_H
#define BTREE_H

#include "node.h"
```

```
class bTree
{
private:
    Node<int, char>* root;
    QString endCode;
    QMap<char, QString> dictionary;
public:
    bTree() = delete;
    bTree(Node<int, char>* nd): root(nd) {}
    ~bTree(){ destroyTree(root); }

    void destroyTree(Node<int, char>*& node);
    void formCodes();
    void formCodesRec(Node<int, char>* node, QString tempCode);
    QString getEndCode();
    QMap<char, QString>& getDictionary();
};

#endif // BTREE_H

//bTree.cpp

#include "btree.h"

void bTree::destroyTree(Node<int, char>*& node){
    if(node){
        destroyTree(node->left);
        destroyTree(node->right);
        delete node;
        node = nullptr;
    }
}

void bTree::formCodes(){
    formCodesRec(root, "");
}

void bTree::formCodesRec(Node<int, char>* node, QString
tempCode){
    if(node){
        if(node->isEndNode())
            endCode = tempCode;
        if(node->hasValue()){
            node->code = tempCode;
            dictionary[node->value] = tempCode;
        }
        formCodesRec(node->left, tempCode+'1');
        formCodesRec(node->right, tempCode+'0');
    }
}

QMap<char, QString>& bTree::getDictionary(){
```

```
    return dictionary;
}

QString bTree::getEndCode(){
    return endCode;
}

//Catalog.h

#ifndef CATALOG_H
#define CATALOG_H

#include "pch.h"

class Catalog
{
private:
    QMap<char, int> catalog;
public:
    Catalog() = default;
    Catalog(QByteArray info);

    QMap<char, int>& add(QByteArray info);
    QMap<char, int>& getCatalog();
};

#endif // CATALOG_H

//Catalog.cpp

#include "catalog.h"

Catalog::Catalog(QByteArray info){
    add(info);
}

QMap<char, int>& Catalog::add(QByteArray info){
    for(char a : info){
        if(catalog.value(a, 0))
            ++catalog[a];
        else
            catalog[a]=1;
    }
    return catalog;
}

QMap<char, int>& Catalog::getCatalog(){
    return catalog;
}

//Coder.h
```

```
#ifndef CODER_H
#define CODER_H

#include "pch.h"

class Coder
{
private:
    QString endCode;
    char prev;
    bool leftPrev;
    short prevSize;
    int len{};
    QMap<char, QString> dictionary;

public:
    Coder():leftPrev(false), prevSize(0) {}
    bool hasPrev();
    char getPrev();
    void setDictionary(QMap<char, QString>& dict);
    QMap<char, QString> getDictionary();
    void setEndCode(QString code);
    QString getEndCode();
    QByteArray getEof();
    QByteArray encode(char sb);
    QByteArray getNextCodeBuffer(QByteArray source);
    void clear();

    static char formByte(QString code);
    static QByteArray formBytes(QString code);
};

#endif // CODER_H

//Coder.cpp

#include "coder.h"

bool Coder::hasPrev(){
    return leftPrev;
}

char Coder::getPrev(){
    return prev;
}

void Coder::setDictionary(QMap<char, QString>& dict){
    dictionary = dict;
}

QMap<char, QString> Coder::getDictionary(){
    return dictionary;
```

```
}

void Coder::setEndCode(QString code){
    endCode = code;
}
QString Coder::getEndCode(){
    return endCode;
}

QByteArray Coder::getEof(){
    char byte{0};
    int counter{8};
    QByteArray bf;
    if(hasPrev()){
        byte = prev;
        counter = 8-prevSize;
    }
    for(QChar bit: endCode){
        if(!counter){
            bf += byte;
            byte = 0;
            counter = 8;
        }
        else
            byte<<=1;
        if(bit=='1'){
            byte|=1;
        }
        --counter;
    }
    byte<<=counter;
    bf+=byte;
    prev = 0;
    leftPrev = false;
    prevSize = 0;
    return bf;
}

QByteArray Coder::encode(char sb){
    QByteArray bf;
    QString code;
    char byte{0};
    code = dictionary[sb];
    int counter = 8;
    if(leftPrev){
        leftPrev = false;
        byte = prev;
        counter = 8 - prevSize;
    }
    for(QChar bit: code){
        if(!counter){
            bf += byte;
```

```
            byte = 0;
            counter = 8;
        }
        else
            byte<<=1;
        if(bit=='1'){
            byte|=1;
        }
        --counter;

    }
    bf+=byte;
    return bf;
}

//Length of code must be less than 8, but greater than 0
char Coder::formByte(QString code){
    if(code.length() > 8|| code.length() < 1)
        throw std::runtime_error("Coder::formByte: argument
length must less than be 8, but greater than 0");
    char byte{0};

    for(QChar bit: code){
        byte<<=1;
        if(bit=='1')
            byte|=1;
    }
    byte<<=8-code.length();
    return byte;
}

//Returns string of bytes aligned by left: 111 represented as
1110000, not 00000111
QByteArray Coder::formBytes(QString code){
    if(!code.size())
        throw std::runtime_error("Coder::formBytes: code size
must be greater than 0");
    QByteArray bf;
    while(code.size()>8){
        bf+=formByte(code.left(8));
        code = code.right(code.length()-8);
    }
    bf+=formByte(code.left(8));
    return bf;
}

QByteArray Coder::getNextCodeBuffer(QByteArray source){
    QByteArray temp;
    QByteArray buf;
    for(char sb: source){
        temp = encode(sb);
        len += dictionary[sb].length();
```

```
        int i{};
        while(len>=8){
            buf+=temp[i++];
            len-=8;
        }
        if(len){
            prevSize = len;
            prev = temp[i];
            leftPrev = true;
        }
        else
            leftPrev = false;
    }
    //qDebug() << buf;
    return buf;
}

void Coder::clear(){
    dictionary.clear();
    endCode.clear();
    len = 0;
    leftPrev = false;
    prevSize = 0;
    prev = 0;
}

//DearchivationThread.h

#ifndef DEARCHIVATIONTHREAD_H
#define DEARCHIVATIONTHREAD_H

#include <QThread>
#include <QString>

#include "filedecoder.h"

class DearchivationThread: public QThread
{
    Q_OBJECT
private:
    QString fileName;
    QString filePath;
    QWidget *parent;
signals:
    void exception_executed(QString e);
public:
    DearchivationThread() = default;

    void setFileName(QString fileName_){
        fileName=fileName_;
    }
```

```
    void setFilePath(QString filePath_){
        filePath = filePath_;
    }

    void run();
};

#endif // DEARCHIVATIONTHREAD_H

//DearchivationThread.cpp

#include "dearchivationthread.h"

void DearchivationThread::run(){
    try{
        FileDecoder fd;
        fd.dearchive(filePath, fileName);
    }
    catch(std::runtime_error e){
        emit exception_executed(e.what());
    }
}

//DialogWindow.h

#include "dearchivationthread.h"

void DearchivationThread::run(){
    try{
        FileDecoder fd;
        fd.dearchive(filePath, fileName);
    }
    catch(std::runtime_error e){
        emit exception_executed(e.what());
    }
}

//DialogWindow.cpp

#include "dialogwindow.h"
#include "mainwindow.h"

DialogWindow::DialogWindow(QWidget* parent): QDialog(parent)
{
    message = new QLabel("&Enter archive name without
extension:");
    input = new QLineEdit;
    message->setBuddy(input);
    ok = new QPushButton("&Ok");
    ok->setDisabled(true);
    ok->setDefault(true);
    cancel = new QPushButton("&Cancel");
```

```
    QVBoxLayout *inputLayout = new QVBoxLayout;
    inputLayout->addWidget(message);
    inputLayout->addWidget(input);

    QVBoxLayout *buttonsLayout = new QVBoxLayout;
    buttonsLayout->addWidget(ok);
    buttonsLayout->addWidget(cancel);

    QHBoxLayout *windowLayout = new QHBoxLayout;
    windowLayout->addLayout(inputLayout);
    windowLayout->addLayout(buttonsLayout);

    setLayout(windowLayout);
    setWindowTitle("Archive name input");

    setWindowFlags( Qt::Window | Qt::WindowTitleHint |
Qt::WindowCloseButtonHint);


    connect(input, SIGNAL(textChanged(QString)), this,
SLOT(on_text_changed(QString)));
    connect(ok, SIGNAL(clicked()), this,
SLOT(on_ok_button_clicked()));
    connect(cancel, SIGNAL(clicked()), this,
SLOT(on_cancel_button_clicked()));
}

void DialogWindow::on_text_changed(QString str){
    ok->setEnabled(!str.isEmpty());
}

void DialogWindow::on_ok_button_clicked(){
    emit fileNameEntered(input->text());
    input->clear();
    emit close();
}

void DialogWindow::on_cancel_button_clicked(){
    input->clear();
    emit close();
}

//FileCollector.h

#ifndef FILECOLLECTOR_H
#define FILECOLLECTOR_H

#include <QDir>
#include <QFile>
#include <QString>
#include <QList>
```

```
#include <QThread>
#include "fileentry.h"

class FileCollector
{
private:
    QDir startDir;
    QList<FileEntry> allFiles;
    QStringList allDirs;
public:
    FileCollector(QDir dir){
        startDir = dir;
    }

    QList<FileEntry> collectFiles();
    void collect_files(QDir current, QString relativePath);
    static int dirSize(QString dirPath,  int size, const int
max);
    static bool isCorrectFileName(QString path);
};

#endif // FILECOLLECTOR_H

//FileCollector.cpp


#include "filecollector.h"
#include <QDebug>

QList<FileEntry> FileCollector::collectFiles(){
    collect_files(startDir, startDir.dirName());
    QString allDirsLine;
    for(QString dir: allDirs){
        if(QThread::currentThread()->isInterruptionRequested())
            return allFiles;
        allDirsLine.append(dir);
        allDirsLine.append('|');
    }
    allDirsLine.append('<');
    FileEntry temp(allDirsLine, "", "");
    allFiles.push_front(temp);
    return allFiles;
}

void FileCollector::collect_files(QDir current, QString
relativePath){
    if(QThread::currentThread()->isInterruptionRequested())
        return;
    QStringList files = current.entryList(QDir::Files |
QDir::NoDotAndDotDot);
    QStringList dirs = current.entryList(QDir::AllDirs |
QDir::NoDotAndDotDot);
```

```
    if(!relativePath.isEmpty() &&
!allDirs.contains(relativePath))
        allDirs.push_back(relativePath);
    for(QString file: files){
        if(QThread::currentThread()->isInterruptionRequested())
            return;
        qDebug() << file << "   " << relativePath;
        FileEntry
temp(current.absolutePath()+QDir::separator()+file,
relativePath, file);
        allFiles.append(temp);
    }
    for(QString dir: dirs){
        if(QThread::currentThread()->isInterruptionRequested())
            return;
        QDir temp(current.absolutePath()+QDir::separator()+dir);
        collect_files(temp, relativePath+"/"+dir);
    }
}

int FileCollector::dirSize(QString dirPath, int size, const int
max){
    QDir dir{dirPath};
    for(QString filePath: dir.entryList(QDir::Files |
QDir::System | QDir::Hidden)){
        ++size;
    }
    for(QString childDirPath: dir.entryList(QDir::Dirs |
QDir::NoDotAndDotDot | QDir::System | QDir::Hidden)){
        ++size;
        size=dirSize(dirPath+QDir::separator()+childDirPath,
size, max);
        if(size>50)
            return size;
    }
    return size;
}

bool FileCollector::isCorrectFileName(QString path){
    // Anything following the raw filename prefix should be
legal.
    if (path.left(4)=="\\\\?\\")
        return true;

    // Windows filenames are not case sensitive.
    path = path.toUpper();

    // Trim the drive letter off
    if (path[1]==':' && (path[0]>='A' && path[0]<='Z'))
        path = path.right(path.length()-2);

    QString illegal="\\/<>:\"|?*";
```

```
    foreach (const QChar& c, path)
    {
        // Check for control characters
         if (c.toLatin1() > 0 && c.toLatin1() < 32)
            return false;

        // Check for illegal characters
        if (illegal.contains(c))
            return false;
    }

    // Check for device names in filenames
    static QStringList devices;

    if (!devices.count())
        devices << "CON" << "PRN" << "AUX" << "NUL" << "COM0" <<
"COM1" << "COM2"
                << "COM3" << "COM4" << "COM5" << "COM6" <<
"COM7" << "COM8" << "COM9" << "LPT0"
                << "LPT1" << "LPT2" << "LPT3" << "LPT4" <<
"LPT5" << "LPT6" << "LPT7" << "LPT8"
                << "LPT9";

    const QFileInfo fi(path);
    const QString basename = fi.baseName();

    foreach (const QString& s, devices)
        if (basename == s)
            return false;

    // Check for trailing periods or spaces
    if (path.right(1)=="." || path.right(1)==" ")
        return false;

    // Check for pathnames that are too long (disregarding raw
pathnames)
    if (path.length()>260)
        return false;

    // Exclude raw device names
    if (path.left(4)=="\\\\.\\")
        return false;

    // Since we are checking for a filename, it mustn't be a
directory
    if (path.right(1)=="\\")
        return false;

    return true;
}
```

```
#ifndef FILEDECODER_H
#define FILEDECODER_H

#include "pch.h"
#include "readbuffer.h"

#include <QThread>
#include <QDir>
#include <QMessageBox>

class FileDecoder
{
private:
    ReadBuffer input;
    QMap<QString, char> dictionary{};
    QString dirName{};
    QString endCode{};
    QString outpPath{};
    QString mainDir{};
    int longestCodeSize{};
public:
    FileDecoder():input(5120) {}
    void dearchive(QString path, QString fileName);
    bool decodeDictionary();
    void decodeFile(QFile& outf);
    void readDireactoryTree();
    int getNum();
    QByteArray getPath();
    QString toCode(char c, int length);
};

#endif // FILEDECODER_H

#ifndef FILEDECODER_H
#define FILEDECODER_H

#include "pch.h"
#include "readbuffer.h"

#include <QThread>
#include <QDir>
#include <QMessageBox>

class FileDecoder
{
private:
    ReadBuffer input;
    QMap<QString, char> dictionary{};
    QString dirName{};
    QString endCode{};
    QString outpPath{};
    QString mainDir{};
```

```
    int longestCodeSize{};
public:
    FileDecoder():input(5120) {}
    void dearchive(QString path, QString fileName);
    bool decodeDictionary();
    void decodeFile(QFile& outf);
    void readDireactoryTree();
    int getNum();
    QByteArray getPath();
    QString toCode(char c, int length);
};

#endif // FILEDECODER_H

//FileDecoder.cpp

#include "filedecoder.h"

void FileDecoder::dearchive(QString path, QString fileName){
    input.openFile(path+"/"+fileName);
    outpPath = path;
    readDireactoryTree();
    while (!input.isEnd()){
        if(QThread::currentThread()->isInterruptionRequested())
            return;
        dictionary.clear();
        longestCodeSize = 0;
        QString currentPath = getPath();
        QString currentName = getPath();
        if(!outpPath.isEmpty()){
            currentPath.prepend('/');
            currentPath.prepend(outpPath);
        }
        if(!currentPath.isEmpty()){
            currentName.prepend('/');
            currentName.prepend(currentPath);
        }
        QFile outf(currentName);
        if(!outf.open(QFile::WriteOnly)){
            throw std::runtime_error("Can't dearchive");
        }
        if(decodeDictionary())
            decodeFile(outf);
        outf.close();
    }
}

bool FileDecoder::decodeDictionary(){
    int size{};
    QString code;

    size = getNum();
```

```
    if(!size){
        return false;
    }
    do{
        char bt = input.get();
        code+=toCode(bt, size);
        size-=8;
    }while(size>0);
    endCode = code;
    char byte;
    while(true){
        if(QThread::currentThread()->isInterruptionRequested())
            return false;
        code.clear();
        input.get();
        byte = input.get();
        if(dictionary.values().contains(byte))
            break;
        input.get();
        size = getNum();
        do{
            code+=toCode(input.get(), size);
            size-=8;
        }while(size>0);
        dictionary[code]=byte;
    }
    for(QString str: dictionary.keys()){
        if(QThread::currentThread()->isInterruptionRequested())
            return false;
        if (str.length()>longestCodeSize)
            longestCodeSize = str.length();
    }
    return true;
}

int FileDecoder::getNum(){
    QString num{};
    while(input.peek()!='|'){
        num.append(input.get());
    }
    if(num.isNull())
        throw std::runtime_error("Archive was corrupted");
    input.get();
    return num.toInt();
}

QByteArray FileDecoder::getPath(){
    QByteArray path{};
    while(input.peek()!='|'){
        path.append(input.get());
    }
    input.get();
```

```
    return path;
}

QString FileDecoder::toCode(char c, int length){
    QString code;
    int bit8 = 128;
    if (length>8)
        length = 8;
    while(length){
        if(c&bit8){
            code.append("1");
        }
        else
            code.append("0");
        --length;
        c<<=1;
    }
    return code;
}

void FileDecoder::decodeFile(QFile& outf){
    char byte{};
    int size{};
    int bit8 = 128;
    bool notEnd{true};
    QString code;
    QByteArray buffer;
    while(notEnd){
        if(QThread::currentThread()->isInterruptionRequested())
            return;
        if(!size){
            byte = input.get();
            size = 8;
        }
        while(size){
            if (byte&bit8)
                code.append("1");
            else
                code.append("0");

            if(code.length()>longestCodeSize)
                throw std::runtime_error("Archive was
corrupted");

            byte<<=1;
            --size;
            if(code==endCode){
                notEnd=false;
                break;
            }
            if(dictionary.contains(code)){
                buffer.append(dictionary[code]);
```

```
                code.clear();
                if(buffer.size()>128){
                    outf.write(buffer);
                    buffer.clear();
                }
                break;
            }

        }
    }
    if(buffer.size())
        outf.write(buffer);
}

void FileDecoder::readDireactoryTree(){
    QString dir;
    QDir a(outpPath);
    while(input.peek()!='<'){
        if(QThread::currentThread()->isInterruptionRequested())
            return;
        while(input.peek()!='|'){
            dir.append(input.get());
        }
        if(input.peek()=='|'){
            if(!dir.isEmpty()){
                a.mkdir(dir);
            }
        }
        input.get();
        dir.clear();
    }
    input.get();
}

//FileEntry.h

#ifndef FILEENTRY_H
#define FILEENTRY_H

#include <QString>

class FileEntry
{
private:
    QString path;
    QString relativePath;
    QString fileName;
public:
    FileEntry(QString path_, QString relativePath_, QString
file_name){
        path = path_;
        relativePath = relativePath_;
```

```
        fileName = file_name;
    }

    void setPath(QString path_);
    QString getPath();
    void setRelativePath(QString relativePath_);
    QString getRelativePath();
    void setFileName(QString file_name);
    QString getFileName();
};

#endif // FILEENTRY_H

//FileEntry.cpp

#include "fileentry.h"

void FileEntry::setPath(QString path_){
    path = path_;
}

QString FileEntry::getPath(){
    return path;
}

void FileEntry::setRelativePath(QString relativePath_){
    relativePath = relativePath_;
}

QString FileEntry::getRelativePath(){
    return relativePath;
}

void FileEntry::setFileName(QString file_name){
    fileName = file_name;
}

QString FileEntry::getFileName(){
    return fileName;
}

//FileTranslator.h

#ifndef FILETRANSLATOR_H
#define FILETRANSLATOR_H

#include "coder.h"
#include "fileentry.h"

#include <QThread>

using namespace std;
```

```
class FileTranslator
{
private:
    QFile fout;
    QFile fin;
    QList<FileEntry> allFiles;
    Coder coder{};
public:
    FileTranslator() = default;
    FileTranslator(QString path): fout(path){
        fout.open(QFile::WriteOnly);
    }

    ~FileTranslator(){
        fout.close();
        fin.close();
    }

    void openFile(QString path_);
    void setAllFiles(QList<FileEntry> allFiles_);
    QList<FileEntry> getAllFiles();
    void translateFiles();
    void translateDictionary();
    void clear();
};

#endif // FILETRANSLATOR_H

//FileTranslator.cpp

#include "filetranslator.h"
#include "catalog.h"
#include "treeformer.h"
#include "btree.h"

void FileTranslator::openFile(QString path_){
    if(fout.isOpen())
        fout.close();
    fout.setFileName(path_);
    if(!fout.open(QFile::WriteOnly))
        throw std::runtime_error("Cant open output file");
}

void FileTranslator::translateFiles(){
    const int BUFFER_SIZE = 512;
    QByteArray buf{};

    if(allFiles.isEmpty())
        throw std::runtime_error("Internal error");

    buf.clear();
```

```
    buf.append(allFiles.front().getPath());
    fout.write(buf);
    allFiles.pop_front();
    while(allFiles.size()){
        if(QThread::currentThread()->isInterruptionRequested()){
            clear();
            return;
        }
        coder.clear();
        QString path = allFiles.front().getRelativePath();
        buf.clear();
        buf.append(path);
        buf.append("|");
        buf.append(allFiles.front().getFileName());
        buf.append("|");
        fout.write(buf);

        path = allFiles.front().getPath();
        QFile inf(path);
        if(!inf.open(QFile::ReadOnly)){
            throw runtime_error("Can't open input file");
        }
        Catalog cat;

        buf.clear();
        allFiles.pop_front();

        if(inf.size())
            do{
                if(QThread::currentThread()-
>isInterruptionRequested()){
                    clear();
                    return;
                }
                buf = inf.read(BUFFER_SIZE);
                if(buf.isNull() || !buf.size())
                    break;
                cat.add(buf);
            }while(!buf.isNull());
        else{
            buf.clear();
            buf.append("0|");
            fout.write(buf);
            buf.clear();
            continue;
        }

        TreeFormer trf(cat.getCatalog());
        bTree tree = trf.formBTree();
        tree.formCodes();
        coder.setDictionary(tree.getDictionary());
        coder.setEndCode(tree.getEndCode());
```

```
        translateDictionary();

        inf.seek(0);
        do{
            if(QThread::currentThread()-
>isInterruptionRequested()){
                clear();
                return;
            }
            buf = inf.read(BUFFER_SIZE);
            if(buf.isNull() || !buf.size())
                break;
            fout.write(coder.getNextCodeBuffer(buf));
            fout.flush();
        }while(!buf.isNull());
        fout.write(coder.getEof());
        fout.flush();
        inf.close();
    }
    fout.close();
}

void FileTranslator::translateDictionary(){
    QString temp;

fout.write(std::to_string(coder.getEndCode().size()).c_str());
    fout.write("|");
    temp = coder.getEndCode();
    fout.write(Coder::formBytes(temp));

    for(char key: coder.getDictionary().keys()){
        if(QThread::currentThread()->isInterruptionRequested()){
            clear();
            return;
        }
        QByteArray arr;
        arr.clear();
        arr.append("|");
        arr.append(key);
        arr.append("|");

arr.append(to_string(coder.getDictionary()[key].length()).c_str(
));
        arr.append("|");

arr.append(Coder::formBytes(coder.getDictionary()[key]));
        fout.write(arr);
    }
    fout.write("|");
    fout.putChar(coder.getDictionary().keys()[0]);
}
```

```
void FileTranslator::setAllFiles(QList<FileEntry> allFiles_){
    allFiles = allFiles_;
}

QList<FileEntry> FileTranslator::getAllFiles(){
    return allFiles;
}

void FileTranslator::clear(){
    if(fout.isOpen())
        fout.close();
    if(fin.isOpen())
        fin.close();
    allFiles.clear();
    coder.clear();
}

//Main.cpp

#include "pch.h"
#include "mainwindow.h"

#include <QApplication>


int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QFile styles(":style.css");
    if(!styles.open(QFile::ReadOnly))
        throw std::runtime_error("Can't open css file");
    a.setStyleSheet(styles.readAll());
    MainWindow w;
    w.show();

    return a.exec();
}

//MainWindow.h

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "filetranslator.h"
#include "filedecoder.h"
#include "filecollector.h"
#include "dialogwindow.h"
#include "waitbox.h"
#include "archivationthread.h"
#include "dearchivationthread.h"
```

```
#include <QFileSystemModel>
#include <QTreeView>
#include <QLabel>
#include <QMessageBox>
#include <QMainWindow>
#include <QFileInfo>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private:
    void disableButtons();
signals:
    void close_waitBox();
private slots:
    void on_treeView_clicked(const QModelIndex &index);

    void on_archiveButton_clicked();

    void on_dearchiveButton_clicked();

    void on_fileNameEntered(QString input);

    void when_archivation_complete();

    void when_archivation_canceled();

    void when_dearchivation_complete();

    void when_dearchivation_canceled();

    void on_deleteButton_clicked();

    void when_thread_exception_handled(QString e);

private:
    Ui::MainWindow *ui;
    QFileInfo *activeFile;
    QFileSystemModel *model;
    DialogWindow *dw;
    WaitBox *archiveWait;
    WaitBox *dearchiveWait;
    ArchivationThread *at;
```

```
    DearchivationThread *dt;
    QString fileName;
    const int MAX_DEPTH = 50;
    bool processing;
};
#endif // MAINWINDOW_H

//MainWindow.cpp

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    model = new QFileSystemModel(this);
    model->setRootPath(QDir::homePath());
    ui->treeView->setModel(model);
    ui->treeView->setColumnWidth(0, 350);

    this->setWindowTitle("Jacket");
    resize(size().width(), size().height()*1.2);
    setWindowIcon(QIcon("./images/jacket.ico"));

    activeFile = new QFileInfo();
    dw = new DialogWindow(this);
    archiveWait = new WaitBox(this);
    dearchiveWait = new WaitBox(this);
    at = new ArchivationThread();
    dt = new DearchivationThread();


    disableButtons();

    connect(dw, &DialogWindow::fileNameEntered, this,
&MainWindow::on_fileNameEntered);
    connect(at, &QThread::finished, this,
&MainWindow::when_archivation_complete);
    connect(archiveWait, &WaitBox::operation_canceled, this,
&MainWindow::when_archivation_canceled);
    connect(this, &MainWindow::close_waitBox, archiveWait,
&WaitBox::on_signal_to_close);
    connect(dt, &QThread::finished, this,
&MainWindow::when_dearchivation_complete);
    connect(dearchiveWait, &WaitBox::operation_canceled, this,
&MainWindow::when_dearchivation_canceled);
    connect(this, &MainWindow::close_waitBox, dearchiveWait,
&WaitBox::on_signal_to_close);
    connect(dt, &DearchivationThread::exception_executed, this,
&MainWindow::when_thread_exception_handled);
```

```
    connect(at, &ArchivationThread::exception_executed, this,
&MainWindow::when_thread_exception_handled);
}

MainWindow::~MainWindow(){
    delete ui;
}

void MainWindow::on_treeView_clicked(const QModelIndex &index)
{
    disableButtons();
    if(!processing){
        *activeFile = model->fileInfo(index);
        ui->infoLabel->setText(activeFile->fileName());

        if(activeFile->suffix() == "jacket")
            ui->dearchiveButton->setEnabled(true);
        ui->archiveButton->setEnabled(true);
        ui->deleteButton->setEnabled(true);
    }
}

void MainWindow::on_archiveButton_clicked()
{
    if(!activeFile->fileName().isEmpty()){
        dw->show();
    }
    else
        QMessageBox::warning(this, "Wrong input", "Please,
choose file or directory first");
}

void MainWindow::on_dearchiveButton_clicked()
{
    if(activeFile->suffix() == "jacket"){
        processing = true;
        ui->infoLabel->setText("");

        dt->setFileName(activeFile->fileName());
        dt->setFilePath(activeFile->absolutePath());
        dt->start();
        dearchiveWait->show();
    }
    else{
        QMessageBox::warning(this, "Wrong file", "Please, choose
file with '.jacket' extension");
    }
}

void MainWindow::on_fileNameEntered(QString input){
    if(!input.isEmpty() &&
FileCollector::isCorrectFileName(input)){
```

```
        processing = true;
        ui->infoLabel->setText("");

        fileName = input;
        dw->close();

        fileName+=".jacket";
        fileName.prepend('/');
        fileName.prepend(activeFile->absolutePath());

        at->setFileName(fileName);
        at->setActiveFile(activeFile);
        at->start();
        archiveWait->show();
    }
    else {
        fileName = "";
        QMessageBox::warning(this, "Wrong input", "Please, input
filename without special symbols");
    }
}

void MainWindow::when_archivation_complete(){
    emit close_waitBox();
    QString prev = model->rootPath();
    model->setRootPath(activeFile->absolutePath());
    model->setRootPath(prev);
    processing = false;
    disableButtons();
}

void MainWindow::when_archivation_canceled(){
    at->requestInterruption();
    at->wait();
    emit close_waitBox();
    QFile::remove(fileName);
    processing = false;
}


void MainWindow::when_dearchivation_complete(){
    emit close_waitBox();
    QString prev = model->rootPath();
    model->setRootPath(activeFile->absolutePath());
    model->setRootPath(prev);
    processing = false;
    disableButtons();
}

void MainWindow::when_dearchivation_canceled(){
    dt->requestInterruption();
    dt->wait();
```

```
    processing = false;
    emit close_waitBox();
}

void MainWindow::on_deleteButton_clicked()
{
    if(!activeFile->fileName().isEmpty()){
        QString what("file");
        if(activeFile->isDir()){
            if(FileCollector::dirSize(activeFile-
>absoluteFilePath(), 0, MAX_DEPTH)>MAX_DEPTH){
                QMessageBox::warning(this, "Bad directory",
"Choosen directory depth is to big");
                return;
            }
            what = "directory";
        }
        if(QMessageBox::Yes == QMessageBox::question(this, "",
"Are you sure you want to delete this " + what+ "?")){
            if(activeFile->isDir()){
                QDir temp(activeFile->absoluteFilePath());
                temp.removeRecursively();
            }
            else{
                QFile temp(activeFile->absoluteFilePath());
                temp.remove();
            }

        }
    }
    else
        QMessageBox::warning(this, "Wrong input", "Please,
choose file or directory first");
    processing = false;
    disableButtons();
}

void MainWindow::disableButtons(){
    ui->archiveButton->setEnabled(false);
    ui->dearchiveButton->setEnabled(false);
    ui->deleteButton->setEnabled(false);
}

void MainWindow::when_thread_exception_handled(QString e){
    QMessageBox::critical(this, "Error", e);
}

//Node.h

#ifndef NODE_H
#define NODE_H
```

```
#include "pch.h"

template <typename T, typename N>
class Node {
private:
    bool has_value{true};
public:
    QString code;
    T count{};
    N value{};
    Node<T, N>* left = nullptr, *right = nullptr;
    bool endNode{false};

    Node() = default;
    Node(T cnt): count(cnt), left(nullptr), right(nullptr),
has_value(false), endNode(true) {}
    Node(T cnt, N val): count(cnt), value(val), left(nullptr),
right(nullptr), has_value(true) {}
    Node(Node<T, N>* lef, Node<T, N>* rgt) : left(lef),
right(rgt), has_value(false) {
        count = left->count+right->count;
    }
    ~Node() {
        if(left)
            delete left;
        if(right)
            delete right;
    }

    T getCount(){
        return count;
    }

    N getValue(){
        return value;
    }

    QString getCode(){
        return code;
    }

    bool hasValue(){
        return has_value;
    }


    void setEndNode(bool t){
        endNode = t;
    }

    bool isEndNode(){
        return endNode;
```

```
    }

    bool operator>(Node<T, N> other){
        return count>other.getCount();
    }

    bool operator<(Node<T, N> other){
        return count<other.getCount();
    }

    bool operator==(Node<T, N> other){
        return count==other.getCount();
    }

    bool operator!=(Node<T, N> other){
        return !count==other.getCount();
    }
};



#endif // NODE_H

//NodeComparator.h

#ifndef NODECOMPARATOR_H
#define NODECOMPARATOR_H

#include "node.h"

struct NodeComparator
{
public:
    bool operator()(Node<int, char>* first, Node<int, char>*
second){
        return first->getCount()>second->getCount();
    }
};

#endif // NODECOMPARATOR_H

//pch.h

#ifndef PCH_H
#define PCH_H

#include <QFile>
#include <QString>
#include <QDebug>
#include <QMap>
#include <queue>
#include <iostream>
```

```
#include <string>
#include <vector>
#include <list>
#include <map>
#include <windows.h>
#include <locale>
#include <iomanip>
#include <sstream>
#include <fstream>


using std::cout;
using std::cin;
using std::endl;
using std::setw;
using std::string;
using std::wstring;
using std::list;
using std::vector;
using std::map;
using std::priority_queue;
using std::cout;
using std::pair;
using std::ifstream;
using std::ofstream;
using std::ios;

#endif // PCH_H

//ReadBuffer.h

#ifndef READBUFFER_H
#define READBUFFER_H

#include <QFile>

class ReadBuffer
{
private:
    QFile fin;
    char* buffer;
    const size_t inputSize;
    size_t index{};
    size_t readen{};
public:
    ReadBuffer(size_t inputSize_): inputSize(inputSize_){
        buffer = new char[inputSize];
        index = inputSize;
    }

    ~ReadBuffer(){
        delete[] buffer;
```

```
        fin.close();
    }

    void openFile(QString fileName){
        if(fin.isOpen())
            fin.close();
        fin.setFileName(fileName);
        fin.open(QFile::ReadOnly);
    }

    char get(){
        nextBuffer();
        return buffer[index++];
    }

    char peek(){
        nextBuffer();
        return buffer[index];
    }

    void nextBuffer(){
        if(!(index<inputSize)){
            memset(buffer, 0, inputSize);
            if(fin.atEnd()){
                throw
std::runtime_error("ReadBuffer::nextBuffer(): File was
corrupted");
            }
            readen = fin.read(buffer, inputSize);
            index=0;
        }
    }

    bool isEnd(){
        if(index<readen)
            return false;
        else
            return fin.atEnd();
    }

    void close(){
        fin.close();
    }
};

#endif // READBUFFER_H

//TreeFormer.h

#ifndef TREEFORMER_H
#define TREEFORMER_H
```

```
#include "pch.h"
#include "node.h"
#include "nodecomparator.h"

class TreeFormer
{
private:
    priority_queue<Node<int, char>*, vector<Node<int, char>*>,
NodeComparator> nodes;
public:
    TreeFormer() = default;
    TreeFormer(QMap<char, int> cat);

    void add(Node<int, char>* nw);
    Node<int, char>* take();
    Node<int, char>* formBTree();
};

#endif // TREEFORMER_H

//TreeFormer.cpp

#include "treeformer.h"

TreeFormer::TreeFormer(QMap<char, int> cat){
    for(char a: cat.keys()){
        Node<int, char> *temp = new Node<int, char>(cat[a], a);
        nodes.push(temp);
    }
}
void TreeFormer::add(Node<int, char>* nw){
    nodes.push(nw);
}

Node<int, char>* TreeFormer::take(){
    Node<int, char>* temp = nodes.top();
    nodes.pop();
    return temp;
}

Node<int, char>* TreeFormer::formBTree(){
    Node<int, char>* tree;
    if(nodes.empty())
        tree = nullptr;
    Node<int, char>* left, *right;
    Node<int, char> *temp = new Node<int, char>(0);
    nodes.push(temp);
    while(nodes.size()>1){
         left = take();
         right = take();
         Node<int, char>* temp = new Node<int, char>(left,
right);
```

```
            nodes.push(temp);
        }
        tree = take();
        return tree;
}

//WaitBox.h

#ifndef WAITBOX_H
#define WAITBOX_H

#include <QDialog>
#include <QThread>
#include <QLabel>
#include <QObject>
#include <QVBoxLayout>
#include <QPushButton>

class WaitBox: public QDialog
{
    Q_OBJECT
private:
    QLabel* msgLabel;
    QPushButton *cancel;
public:
    WaitBox(QWidget *parent = nullptr);
    void show();
signals:
    void operation_canceled();
public slots:
    void on_cancel_button_clicked();
    void on_signal_to_close();
};

#endif // WAITBOX_H

//WaitBox.cpp

#include "waitbox.h"

WaitBox::WaitBox(QWidget *parent)
    : QDialog(parent),
      msgLabel(new QLabel("Please wait until the end of
operation", this)),
      cancel(new QPushButton("Cancel")){
    QVBoxLayout* mainLayout = new QVBoxLayout;
    mainLayout->setContentsMargins(30, 30, 30, 30);
    mainLayout->addWidget(msgLabel);
    mainLayout->addWidget(cancel);
    setLayout(mainLayout);
    setWindowFlags(Qt::Dialog
        | Qt::WindowTitleHint
```

```
        | Qt::MSWindowsFixedSizeDialogHint);
    QObject::connect(cancel, &QPushButton::clicked, this,
&WaitBox::on_cancel_button_clicked);
}
void WaitBox::show(){
    QDialog::show();
}

void WaitBox::on_cancel_button_clicked(){
    emit operation_canceled();
}

void WaitBox::on_signal_to_close(){
    emit close();
}


//MainWindow.ui

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="windowModality">
   <enum>Qt::NonModal</enum>
  </property>
  <property name="enabled">
   <bool>true</bool>
  </property>
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>779</width>
    <height>617</height>
   </rect>
  </property>
  <property name="sizePolicy">
   <sizepolicy hsizetype="Preferred" vsizetype="Preferred">
    <horstretch>0</horstretch>
    <verstretch>0</verstretch>
   </sizepolicy>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <property name="styleSheet">
   <string notr="true"/>
  </property>
  <property name="inputMethodHints">
   <set>Qt::ImhNone</set>
  </property>
```

```
  <widget class="QWidget" name="centralwidget">
   <layout class="QVBoxLayout" name="verticalLayout">
    <property name="spacing">
     <number>0</number>
    </property>
    <item>
     <widget class="QTreeView" name="treeView">
      <property name="sortingEnabled">
       <bool>true</bool>
      </property>
      <property name="animated">
       <bool>false</bool>
      </property>
     </widget>
    </item>
    <item>
     <layout class="QHBoxLayout" name="horizontalLayout">
      <property name="spacing">
       <number>12</number>
      </property>
      <property name="topMargin">
       <number>20</number>
      </property>
      <item>
       <widget class="QPushButton" name="archiveButton">
        <property name="styleSheet">
         <string notr="true"/>
        </property>
        <property name="text">
         <string>Archive</string>
        </property>
       </widget>
      </item>
      <item>
       <widget class="QPushButton" name="dearchiveButton">
        <property name="styleSheet">
         <string notr="true">.myButton {
      box-shadow: 0px 7px 0px 0px #3cb1c9;
      background-color:#58c2e8;
      border-radius:8px;
      display:inline-block;
      cursor:pointer;
      color:#ffffff;
      font-family:Arial;
      font-size:22px;
      font-weight:bold;
      padding:13px 24px;
      text-decoration:none;
      text-shadow:0px 1px 0px #326e82;
}
.myButton:hover {
      background-color:#2daec2;
```

```
}
.myButton:active {
      position:relative;
      top:1px;
}

   </string>
        </property>
        <property name="text">
         <string>Dearchive</string>
        </property>
       </widget>
      </item>
      <item>
       <widget class="QPushButton" name="deleteButton">
        <property name="text">
         <string>Delete</string>
        </property>
       </widget>
      </item>
      <item>
       <widget class="QLabel" name="infoLabel">
        <property name="text">
         <string/>
        </property>
       </widget>
      </item>
      <item>
       <spacer name="horizontalSpacer">
        <property name="orientation">
         <enum>Qt::Horizontal</enum>
        </property>
        <property name="sizeHint" stdset="0">
         <size>
          <width>40</width>
          <height>20</height>
         </size>
        </property>
       </spacer>
      </item>
     </layout>
    </item>
   </layout>
  </widget>
  <widget class="QMenuBar" name="menubar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>779</width>
     <height>21</height>
    </rect>
```

```
   </property>
  </widget>
  <widget class="QStatusBar" name="statusbar"/>
 </widget>
 <resources/>
 <connections/>
</ui>

//Style.css

QPushButton {
    border-radius: 3px;
    background-color: #2196f3;
    color:#ffffff;
    font-size:12px;
    font-weight:bold;
    padding:5px 10px;
    min-width: 40px;
}

QPushButton:pressed {
    background-color: #6ec6ff;
}

QPushButton:hover {
    background-color: #0c88eb;
}

QPushButton:flat {
    border: none;
}

QPushButton:default {
    border-color: navy;
}

QPushButton#deleteButton {
    background-color: #ff5252;
}

QPushButton#deleteButton:hover  {
    background-color: #fc3a3a;
}


QPushButton#deleteButton:pressed  {
    background-color: #ff867f;
}

QPushButton:!enabled , QPushButton#deleteButton:!enabled  {
    background-color: #c7c7c7;
}
```