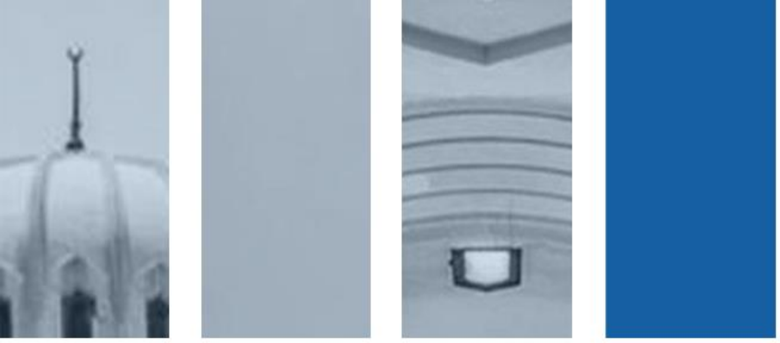


RESEARCH & PROJECT SUBMISSIONS

Program: Mainstream

Course Code: CSE 323

Course Name: Programming with Data Structures

Ain Shams University
Faculty of Engineering
Spring Semester – 2021

Students Personal Information

Students Names:	Students Codes:
مريم محمد رضا احمد محمد حسن	1701405
ايه الله عبد الرحيم محمد تامر	1700337
مريم فهمي صبحي زغلول	1701404
مريم محمد علاء الدين عصمت	1701406

Submission Contents

Github Repository Link: https://github.com/mariam199911/XML-project.git

Video Link: https://drive.google.com/file/d/15gXT3uqQmSjMK_5wR9LOb1S2JvGul3pw/view?usp=drivesdk

01: Background:

Trees:
The tree data structure is a data organization, management, and storage format that enables efficient access and modification. Trees simulate a hierarchical structure, with a root value and sub-trees of children with a parent node, represented as a set of linked nodes.

Stacks:
The stack is an abstract data type that uses the last in, first out (LIFO) structure/technique. In stacks, we can only access the element on the top.

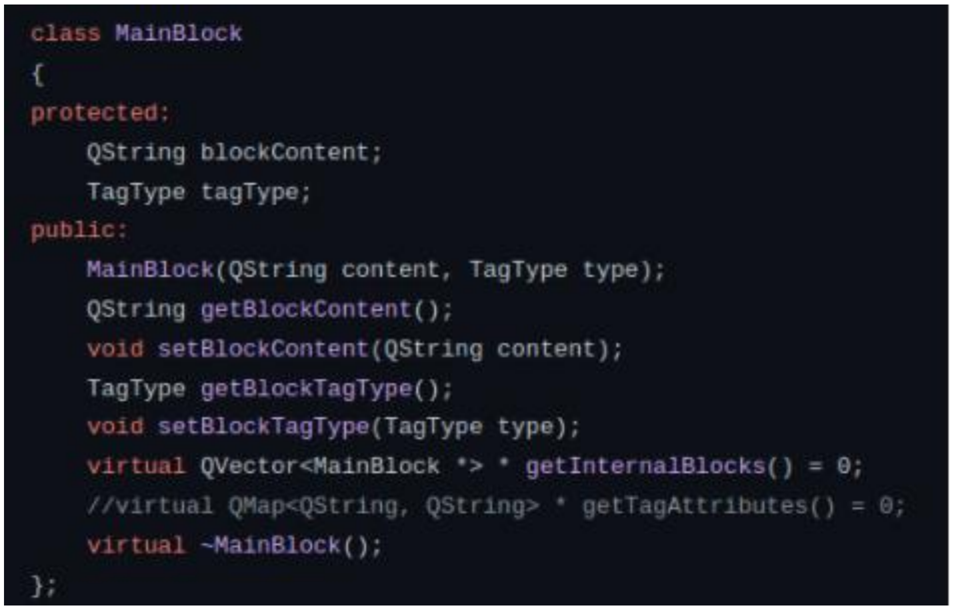
XML Files:
XML is a markup language created to define a syntax for encoding documents that both humans and machines could read using tags that define the structure of the document, as well as how the document should be stored and transported.

JSON Files:
A JSON file is a file that stores simple data structures and objects in JavaScript Object Notation format, which is a standard data interchange format. It’s primarily used for transmitting data between a web application and a server. JSON files are lightweight, text-based, human-readable, and can be edited using a text editor.

02: Implementation Details:

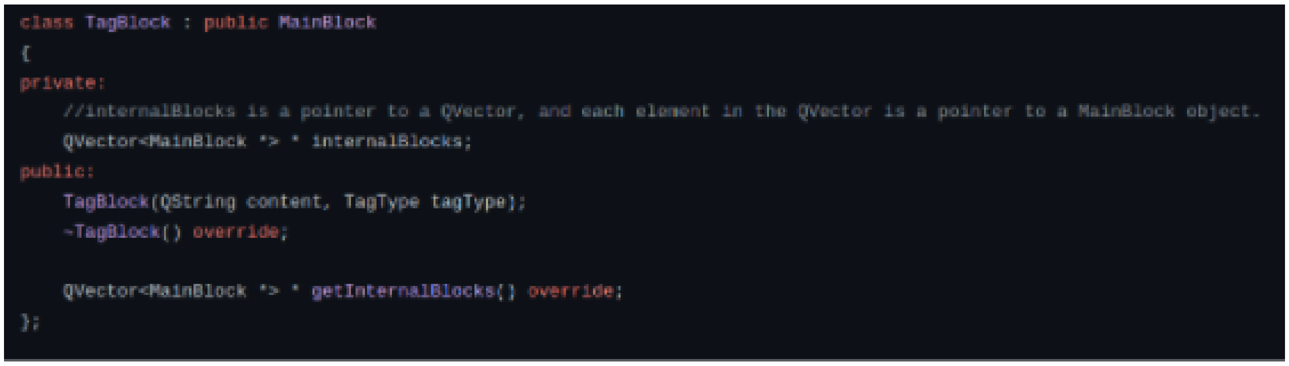
1. Main Block Class:

The base class for representing each tag stored in the XML file.



```
class MainBlock
{
protected:
    QString blockContent;
    TagType tagType;
public:
    MainBlock(QString content, TagType type);
    QString getBlockContent();
    void setBlockContent(QString content);
    TagType getBlockTagType();
    void setBlockTagType(TagType type);
    virtual QVector<MainBlock *> * getInternalBlocks() = 0;
    //virtual QMap<QString, QString> * getTagAttributes() = 0;
    virtual ~MainBlock();
};
```

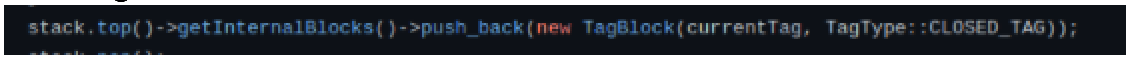
2. TagBlock Class:



```
class TagBlock : public MainBlock
{
private:
    //internalBlocks is a pointer to a QVector, and each element in the QVector is a pointer to a MainBlock object.
    QVector<MainBlock *> * internalBlocks;
public:
    TagBlock(QString content, TagType tagType);
    ~TagBlock() override;

    QVector<MainBlock *> * getInternalBlocks() override;
};
```

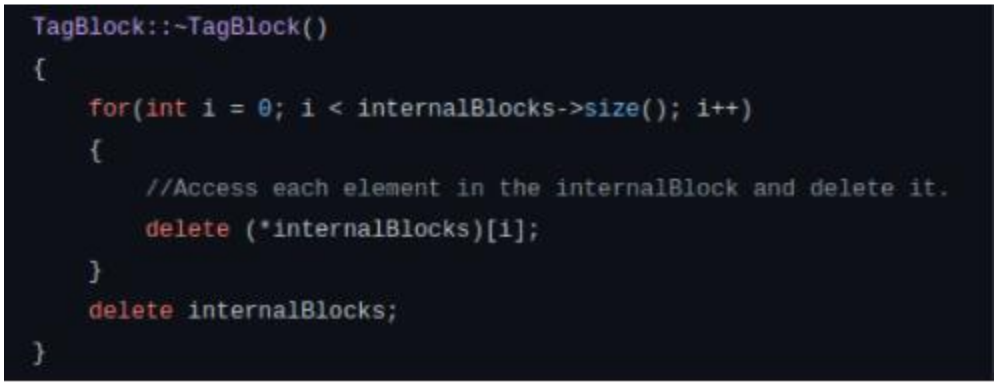
Inherits from **MainBlock** class, and overrides **getInternalBlocks()** (this method returns a pointer to **QVector** that each element in, pointes to a **MainBlock** object).

⇨ In the constructor, we create a new **QVector** (dynamically, which makes it easier to insert a new child tag to a parent tag). We insert a new child tag by declaring a new **QVector** of pointer to **MainBlock**, and we call **root->getInternalBlocks()**, which its return value will be stored in the new declared variable, and then we push back a new child tag.



```
stack.top()->getInternalBlocks()->push_back(new TagBlock(currentTag, TagType::CLOSED_TAG));
```

⇨ In the destructor, each **MainBlock** is deleted as well as **internalBlocks** private variable (**QVector<MainBlock*>**).



```
TagBlock::~TagBlock()
{
    for(int i = 0; i < internalBlocks->size(); i++)
    {
        //Access each element in the internalBlock and delete it.
        delete (*internalBlocks)[i];
    }
    delete internalBlocks;
}
```

3. XMLTree Class:

⇨ Can take the XML file as a **QString (fileText),** or as a **QFile (file),** and converts it into **QString** using **convertingXMLFileIntoText(),** so that it can be used by the other methods.

XMLTree methods:

1- convertingXMLFileIntoText

Used to convert **QFile** into **QString**.

```
void XMLTree::convertingXMLFileIntoText()
{
    file->open(QIODevice::ReadOnly);
    QTextStream fileInput(file);

    //Storing each line in the XML file in the QString private member.
    while(!fileInput.atEnd())
    {
        fileText.append(fileInput.readLine());
    }
    file->close();
}
```

```
void XMLTree::analyzeXMLFileText()
{
    QVector<QString> *separateTag = breakingFileTextIntoTags();
    buildingXMLTree(separateTag);
}
```

2- breakingFileTextIntoTags

```
QVector<QString> * XMLTree::breakingFileTextIntoTags() {...}
```

Used to divide the **fileText** into multiple tags stored in a **QVector**, so each tag is in a separated **QString**.
This is done by looping on each character of the **fileText,**we check that:

1- If this character is **"<",** then:
   - In case of having **"!"** as the next character, then it is a comment.
   - Otherwise, we check if the **QString** is empty or not.
     - In case of an open tag, the current tag is always an empty string, which means that we will add this character to empty **QString** as it is the start of a new open tag.
     - In case of text, the current tag has a value and it isn't empty so it is pushed in the vector of strings, then this **QString** is cleared and stores this character to it as it is the start of a new open tag.

2- If this character is **">"** and it isn't comment, then:
   - It indicates that the current tag ends, which leads to pushing back the **QString** into the **QVector**, and storing it,

3- Otherwise, we check that this character isn't a tab or a new line and if this condition is applied, then:
   - If this character is "- "and it is a comment, then it indicates the end of a comment which leads to incrementing the index by two, two skip the syntax of the end comment.
   - Otherwise, we store the character into the **QString**.

3- buildingXMLTree

```
void XMLTree::buildingXMLTree(QVector<QString> *separateTag) {...}
```

Used to build the whole XML Tree structure, it takes a pointer to a **QVector** of **QString,** and uses the private member ***xmlRoot** which indicates the root of the XML tree.
Steps:

1- Store the address of the first **QString** in the **QVector** in the **xmlRoot.**
2- Push xmlRoot into the **QStack**.
3- Loop on the **QVector** starting from the second element, then check on the **QString**:

- If it's the beginning of a closing tag then the whole tag ends with its internal blocks, so we:
  - Create a new **MainBlock** for it and get the top of the **QStack** then call **getInternalBlocks()** to this top element, which returns tag children and then we insert the newly created **MainBlock** to it.
- If it's **" < ",** it indicates a new open tag, so we:
  - Create a new **MainBlock** for it and get the top of the **QStack** then call **getInternalBlocks()** to this top element, which returns tag children and then we insert the newly created **MainBlock** to it.
  - Push this **MainBlock** to **QStack**, so that in case it contains children, then the next iteration will be inserting a new tag child for it in its **internalBlocks** variable.
- Otherwise, the scanned QString must be a normal text, so we:
  - Creating a new **MainBlock** for it and get the top element of the **QStack** which is a **MainBlock**, then call **getInternalBlocks()** to this top element, which returns tag children and then we insert the newly created **MainBlock** to it.

4- convertXMLFileIntoJSONFile

```
QString XMLTree::convertXMLFileIntoJSONFile() {...}
```

This function is mainly used to convert JSON file into XML file, it uses a private **QString** member which is **JSONFile**, that stored the JSON data, it is done by:

1- Inserting a new **"{"** into **JSONFile**, and inserting a new line to it.
2- Incrementing **indentCounter** by three, which is a private member used to insert the correct space number for the indentation in JSON file.
3- Then calling **generateJSONObject()**
4- Then decrementing **indentCounter** by 3, and inserting **"}"** and a new line to the **JSONFile**.

5- generateJSONObject

```
void XMLTree::generateJSONObject(MainBlock *currentBlock, bool isLastBlock) {...}
```

Used to insert each tag name, tag attributes, and text into its correct syntax taking the indentation into consideration, this function inserts each JSON object into **JSONFile** which is private member in the class, this function takes a pointer to the **MainBlock** which is the root, and it takes another boolean parameter which indicates if the **MainBlock** object is a closed tag, as inserting **"}"** depends on this passed value, it is done by:

- Getting the children of the passed MainBlock.
- If the size of the children is zero, it indicates that the current MainBlock is either a closed tag or a normal text, then it returns the JSONFile, so that it won't continue executing the rest of the function.
- Calling **breakingStartTagIntoParts()** , so that the start tag is divided into tag name which is stored in **QString**, and tag attributes which are stored in **QMap**.
- Calling **generateIndentationForJSON()**, which is responsible for adding the number of spaces to **JSONFile** equals to the **indentCounter.**

```
void XMLTree::generateIndentationForJSON() {...}
```

- Inserting the tag name between two double quotes.
- If the size of the children is two, the current block is an open tag, and has only its text tag, and its closed tag:

  1. Checking if the **QMap** is empty, if the condition is applied, it means that the open tag has no attributes, it leads to inserting the text tag into the **JSONfile** between two double quotes, then checking the passed boolean value and according to it, either a new **"}"** is inserted or not, at the end of the if condition a new line is inserted into the **JSONfile**.

  2. Otherwise, it means that the open tag has an attributes, a new **"}"** and **indentCounter** is incremented by 3, then an iterator is declared so that we can loop over the **QMap**, for each iteration, a new line is inserted, then calling **generateIndentationForJSON()**, then inserting **"@"**, then inserting the map key, then inserting **":"**, then inserting the map value, then inserting a newline into **JSONFile**, and the map iterator is incremented by one, after the iterations on **QMap** ends, the normal text is inserted into **JSONfile** by first calling **generateIndentationForJSON()**, then inserting **"@text"** into **JSONfile**, then inserting the normal text, then inserting a newline into the **JSONfile**, then

decrementing **indentCounter** by 3, and calling **generateIndentationForJSON(),** and a closed bracket **"}"** is inserted.

- Otherwise, it means the current block has a lot of children, then a new open bracket is inserted then a new line is inserted, and inserting its attributes in the same way as illustrated above, then we are looping upon all its children, and for each loop we check if this child is the last one, if this condition is applied then we call the function recursively and passing true as a boolean value, otherwise it isn't the last child, so we call the function recursively and passing false as a boolean value.
- Decrementing **indentCounter** by 3, then calling **generateIndentationForJSON(),** then a new closed bracket is inserted into **JSONFile.**

6- breakingStartTagIntoParts

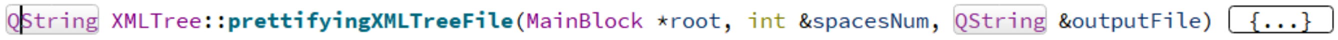
```
QStringList XMLTree::breakingStartTagIntoParts(QString startTag) {...}
```

Used to divide the open tag into its tag name, and its tag attributes (if exist), it's used in converting the XML file into JSON file, so that open tags that has attributes is correctly converted to its syntax in JSON, this function takes a **QString** as a parameter, and returns a **QStringList** which will contain the tag name and the tag attributes.

Steps:

- Looping on each character in the **QString**, and checking:
  1. If the scanned character is an empty character **' ',** in this case we may have an empty character after the tag name or an empty character between the tag attributes, so we declare a new boolean variable which tells us if it is an attribute or not, then the scanned character is inserted in the **QString**, otherwise it indicates the end of a tag name, which leads to inserting the **QString** into the **QStringList**, and then clearing this **QString**.
  2. If the scanned character is double quotes, we check if this double quotes indicates the end of an attribute value, or it indicates the start of an attribute value, It is known by checking the boolean attribute value, if is true, then it indicates the end of an attribute value which leads to inserting the **QString** into **QStringList** and clearing the **QString**, otherwise it indicates a start of an attribute value, and in this case we do nothing, at the end of the condition, the boolean value is toggled.
  3. If the scanned character is equal to **"=" ,** then it indicates the end of an attribute key, which leads to inserting the **QString** into **QStringList** and clearing the **QString**.
  4. If the scanned character is equal to **">",** and the **QString** isn't empty, then it indicated that there is a closed tag and there is text behind it that needs to be stored, this leads to inserting the **QString** into **QStringList**, and clearing the **QString**.
  5. Otherwise, inserting the scanned character into the **QString**.

7- prettifyingXMLTreeFile



```
QString XMLTree::prettifyingXMLTreeFile(MainBlock *root, int &spacesNum, QString &outputFile) {...}
```

Used to print the whole tree with the correct indentation, it’s a function which is called recursively, it takes the following parameters: a pointer to **MainBlock** which is the root, a variable integer for the number of spaces inserted, and a reference to **QString** variable which stores the XML tree structure after prettifying it. Steps:

- Content of the **MainBlock** is inserted in the **QString** with its correct indentation**.**
- Get the children of the root node.
- If the size of the children is zero, it means that the root has no children, in this case the root is either a normal text, or a closed tag, so we just return the **QString**.
- If the size of the children is two, it means that the root has only a normal text, and a closed tag, then we insert the text into the **QString**, and insert the closed tag into the **QString** and then return it.
- Otherwise, it indicates that the root is a parent, then we insert a new line to the **QString**, and increment the number of spaces by three, then loop on all its children, in the loop we check for the closed tag, if the loop reached to the last **MainBlock**, it means that this is the closed tag of the root, in this case the number of spaces is decremented by three, so that it produces a correct XML indentation, otherwise if it isn’t a closed tag then the function is called recursively.

8- minfyingXMLTreeFile

```
QString XMLTree::minfyingXMLTreeFile(MainBlock *root, QString &outputFile) {...}
```

Used to print the whole tree without new lines or indentation, it's called recursively, it takes the following parameters: a pointer to the root, and a reference to **QString** that stores the XML tree structure after minifying it.
Steps:

- Content of the **MainBlock** is inserted as it is in the **QString.**
- Getting the children of the root node.
- If the size of the children is zero, the root has no children, in this case the root is either a normal text, or a closed tag, so we just return the **QString**.
- Otherwise, root is a parent, so the function is called recursively until we reach a tag that has no children and then we return the **QString**.

9- error_checking

```
QString XMLTree::error_checking(QString &outputFile) {...}
```

Checks for errors in the input XML file and displays an error message with the type of the error at the position of this error. These errors can be: **missing closing tag, missing opening tag, or mismatching opening and closing tags**.
The function uses the vector of strings created by the **breakingFileTextIntoTags()** method where each string represents either a tag or text.
The function iterates on this vector and starts checking for errors depending on the type of each string (opening tag, closing tag, text):

- If it's **an opening tag** then it can only be preceded by another opening tag or a closing tag.
  So we check if stack isn't empty then there might be an opening tag or a text on top; if it's an opening tag, then no problem and the new opening tag is pushed into the stack, but if it's a text then its closing tag is missing. Also in case of the special opening tag "frame", we don't push it into the stack as it has no closing tag so we skip it.
- If it's **a closing tag** then it can only be preceded by another closing tag or a text.
  So we check if the stack is empty then opening tag is missing; otherwise, we check if the top of the stack is a text or an opening tag:
  - If it's a text then we get the next top of the stack which should be an opening tag and we compare both tags, but if the stack was empty and there is nothing before this text then opening tag is missing.

Whether the closing tag was preceded by an opening tag or by a text, both opening and closing tags obtained are compared; if they don't match then we check:

- If it was preceded by another closing tag then the function compares the obtained opening tag with all the tags that follow our closing tag; if a matching closing tag is found first then our closing tag has a missing opening tag, but if a matching opening tag is found first or no matching tags are found at all then the opening tag has a missing closing tag.
- If it was preceded by a text then we check the next string in the vector; if it's an opening tag then our closing tag doesn't match the obtained opening tag; However if the next string is another closing tag then the function compares it to the obtained opening tag, if they match then our closing tag has a missing opening tag; otherwise opening and closing tags don't match.

- If it's **a text** then it can only be preceded by an opening tag and followed by a closing tag.
  So we check if the stack is empty then there are no preceding opening tags, which means that this text's opening tag is missing.
  However, if the stack isn't empty then there's an opening tag, so we check that in the vector of the strings: the element before this text is an opening tag and the element after is a closing tag; if both conditions are true then this text is pushed into the stack and if either of them is false then there's a tag missing, either opening or closing.
  The case of a missing closing tag is handled before so in this part of the function, only the missing opening tag case is handled.

Note:
Only opening tags and texts are pushed into the stack while closing tags are only compared with the top of the stack.

10- error_correction

```
QString XMLTree::error_correction(QString &outputFile) {...}
```

Does the same as the previous function but instead of displaying an error message, it corrects the error by inserting the missing tag in its correct position or by making the tags match if they were mismatching.
Note:
The error checking and the error correction functions can work independently as the both detect errors and act on it depending on their functionality.

4. Compression and decompression:

1- Match_Pointer _largest_match:

```
Match_Pointer _largest_match(QByteArray::iterator window, QByteArray::iterator look_ahead_buffer){ ...}
```

Gets the longest match using LZ-77 Algorithm, by adding the characters in the input and searching for it backwardly.

2- QByteArray compression:

```
QByteArray compression(QString& file){ ...}
```

Compresses the file, by taking the input and checking for repetition. If the character is repeated, it stores the beginning of it (how many steps to go backward) and the length of the word (using the longest match function) and the next character. Ex: <3,1,'a'>. Therefore, reducing the number of bytes and reducing the size of file.

3- QByteArray decompression:

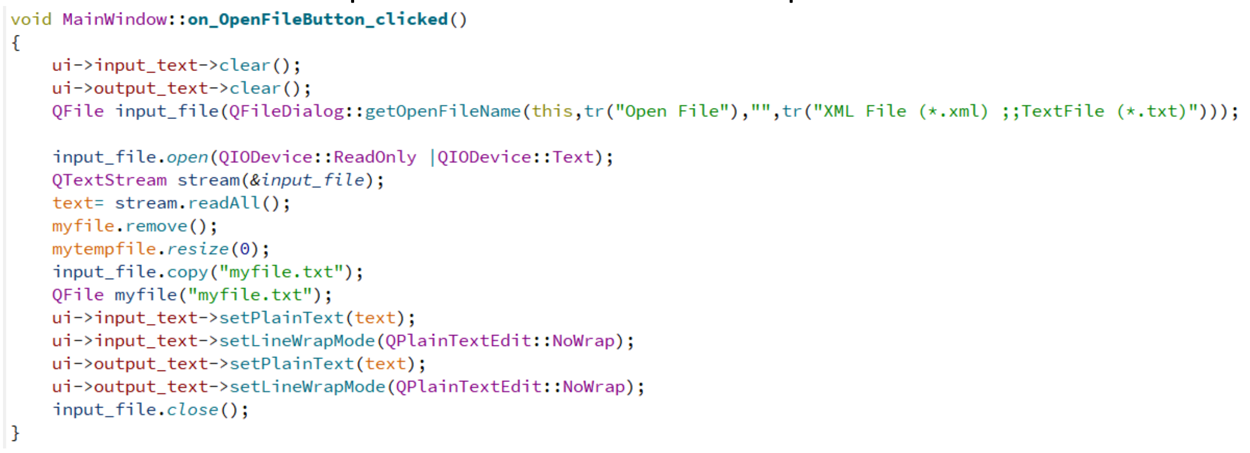
```
QString decompression(QByteArray& compressed){ ...}
```

Decompresses the file entered and returns it to its original form (the reverse of the previous function).

5. UI Slots:

1- on_OpenFileButton_clicked():

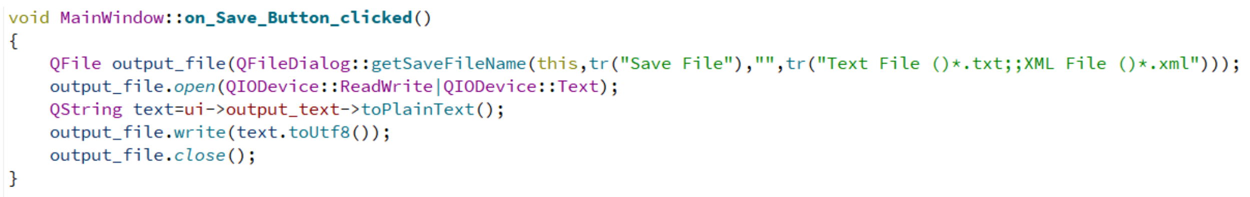
Called when the open file button is clicked to open the file.



```
void MainWindow::on_OpenFileButton_clicked()
{
    ui->input_text->clear();
    ui->output_text->clear();
    QFile input_file(QFileDialog::getOpenFileName(this,tr("Open File"),"",tr("XML File (*.xml) ;;TextFile (*.txt)")));

    input_file.open(QIODevice::ReadOnly |QIODevice::Text);
    QTextStream stream(&input_file);
    text= stream.readAll();
    myfile.remove();
    mytempfile.resize(0);
    input_file.copy("myfile.txt");
    QFile myfile("myfile.txt");
    ui->input_text->setPlainText(text);
    ui->input_text->setLineWrapMode(QPlainTextEdit::NoWrap);
    ui->output_text->setPlainText(text);
    ui->output_text->setLineWrapMode(QPlainTextEdit::NoWrap);
    input_file.close();
}
```

2- on_Save_Button_clicked():

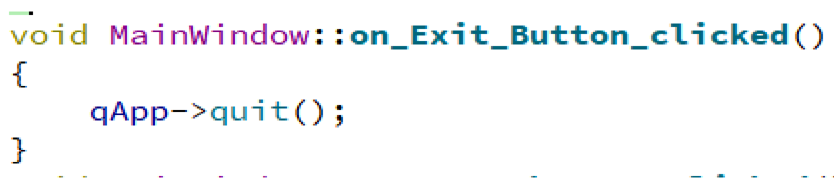
Called when the save file button is clicked to save the file.



```
void MainWindow::on_Save_Button_clicked()
{
    QFile output_file(QFileDialog::getSaveFileName(this,tr("Save File"),"",tr("Text File ()*.txt;;XML File ()*.xml")));
    output_file.open(QIODevice::ReadWrite|QIODevice::Text);
    QString text=ui->output_text->toPlainText();
    output_file.write(text.toUtf8());
    output_file.close();
}
```

3- on_Exit_Button_clicked():

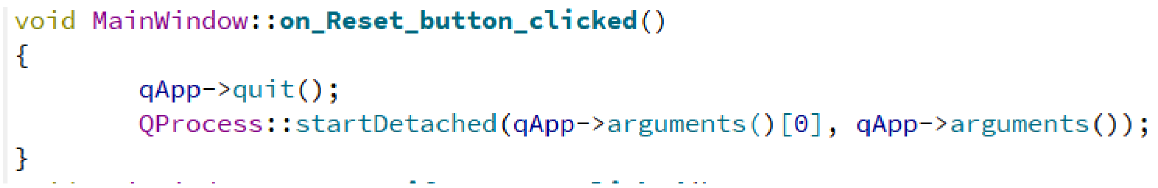
Called when the Exit button is clicked to exit the program.



```
void MainWindow::on_Exit_Button_clicked()
{
     qApp->quit();
}
```

4- on_Reset_button_clicked():

Called when the reset button is clicked to reset the program.



```
void MainWindow::on_Reset_button_clicked()
{
        qApp->quit();
        QProcess::startDetached(qApp->arguments()[0], qApp->arguments());
}
```

5- on_Prettify_Button_clicked():

Called when the prettify button is clicked to prettify the XML file.

```
void MainWindow::on_Prettify_Button_clicked()
{

    XMLTree* treeNode = new XMLTree(text);
    MainBlock* root = treeNode->getXMLFileRoot();
    QString output = "";
    int spacesNum = 0;
    QString out = treeNode->prettifyingXMLTreeFile(root, spacesNum, output);
    ui->output_text->setPlainText(out);
}
```

6- on_Remove_Spaces_clicked():

Called when the Remove spaces button is clicked to remove spaces from the XML file.

```
void MainWindow::on_Remove_Spaces_clicked()
{
    XMLTree* treeNode = new XMLTree(text);
    MainBlock* root = treeNode->getXMLFileRoot();
    QString output = "";
    QString out = treeNode->minfyingXMLTreeFile(root, output);
    ui->output_text->setPlainText(out);
}
```

7- on_JSON_Button_clicked():

Called when the TO JSON button is clicked to check to convert the XML file to a JSON file.

```
void MainWindow::on_JSON_Button_clicked()
{
    XMLTree* treeNode = new XMLTree(text);
    QString out = treeNode->convertXMLFileIntoJSONFile();
    ui->output_text->setPlainText(out);
}
```

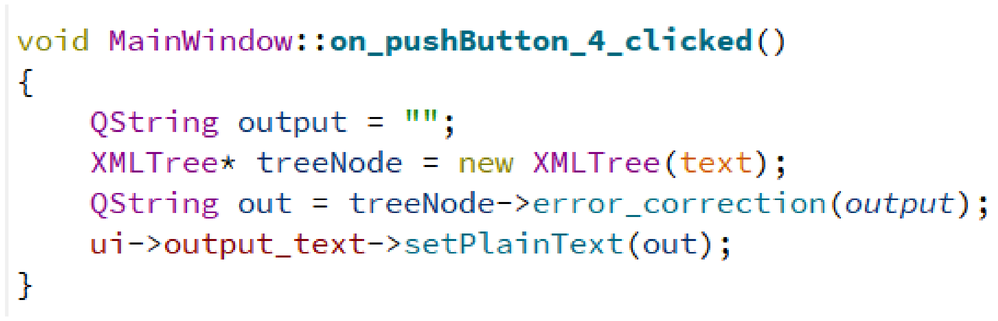
8- on_Check_Button_clicked():

Called when the check button is clicked to check the consistency of the XML file.

```
void MainWindow::on_Check_Button_clicked()
{
    QString output = "";
    XMLTree* treeNode = new XMLTree(text);
    QString out = treeNode->error_checking(output);
    ui->output_text->setPlainText(out);
}
```

9- on_pushButton_4_clicked ():

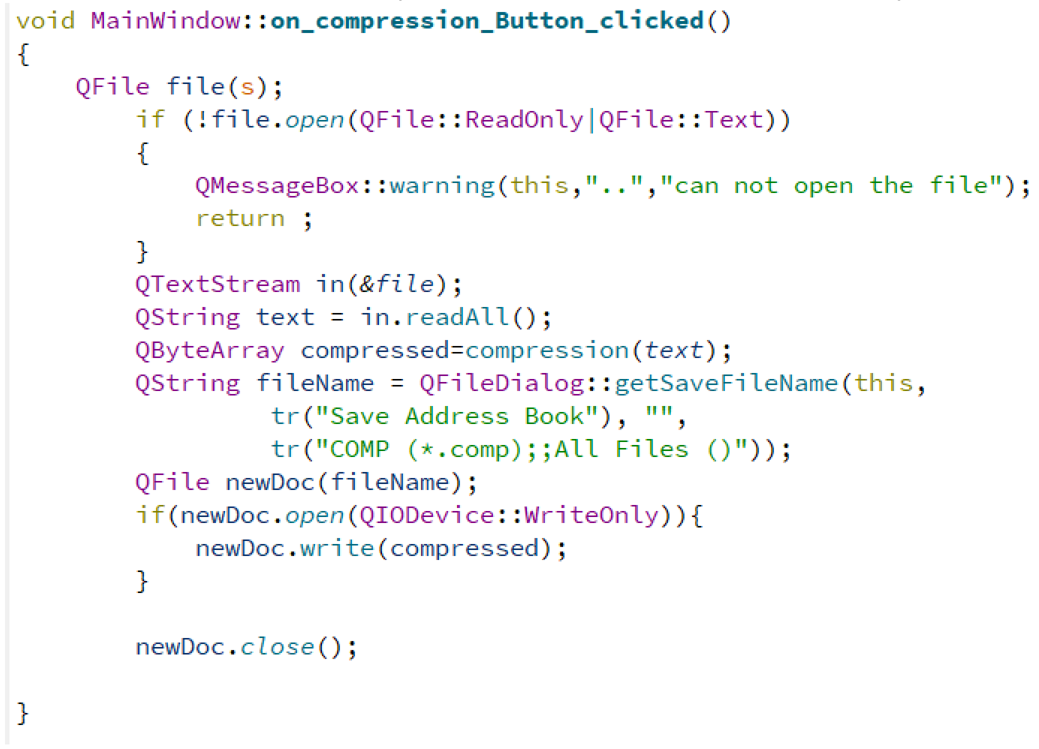
Called when the correct button is clicked to correct the mistakes that were found in the XML file.



```
void MainWindow::on_pushButton_4_clicked()
{
    QString output = "";
    XMLTree* treeNode = new XMLTree(text);
    QString out = treeNode->error_correction(output);
    ui->output_text->setPlainText(out);
}
```

10- on_compression_Button_clicked()

Called when the compression button is clicked to compress the file.



```
void MainWindow::on_compression_Button_clicked()
{
    QFile file(s);
        if (!file.open(QFile::ReadOnly|QFile::Text))
        {
            QMessageBox::warning(this,"..","can not open the file");
            return ;
        }
        QTextStream in(&file);
        QString text = in.readAll();
        QByteArray compressed=compression(text);
        QString fileName = QFileDialog::getSaveFileName(this,
                 tr("Save Address Book"), "",
                 tr("COMP (*.comp);;All Files ()"));
        QFile newDoc(fileName);
        if(newDoc.open(QIODevice::WriteOnly)){
            newDoc.write(compressed);
        }

        newDoc.close();

}
```

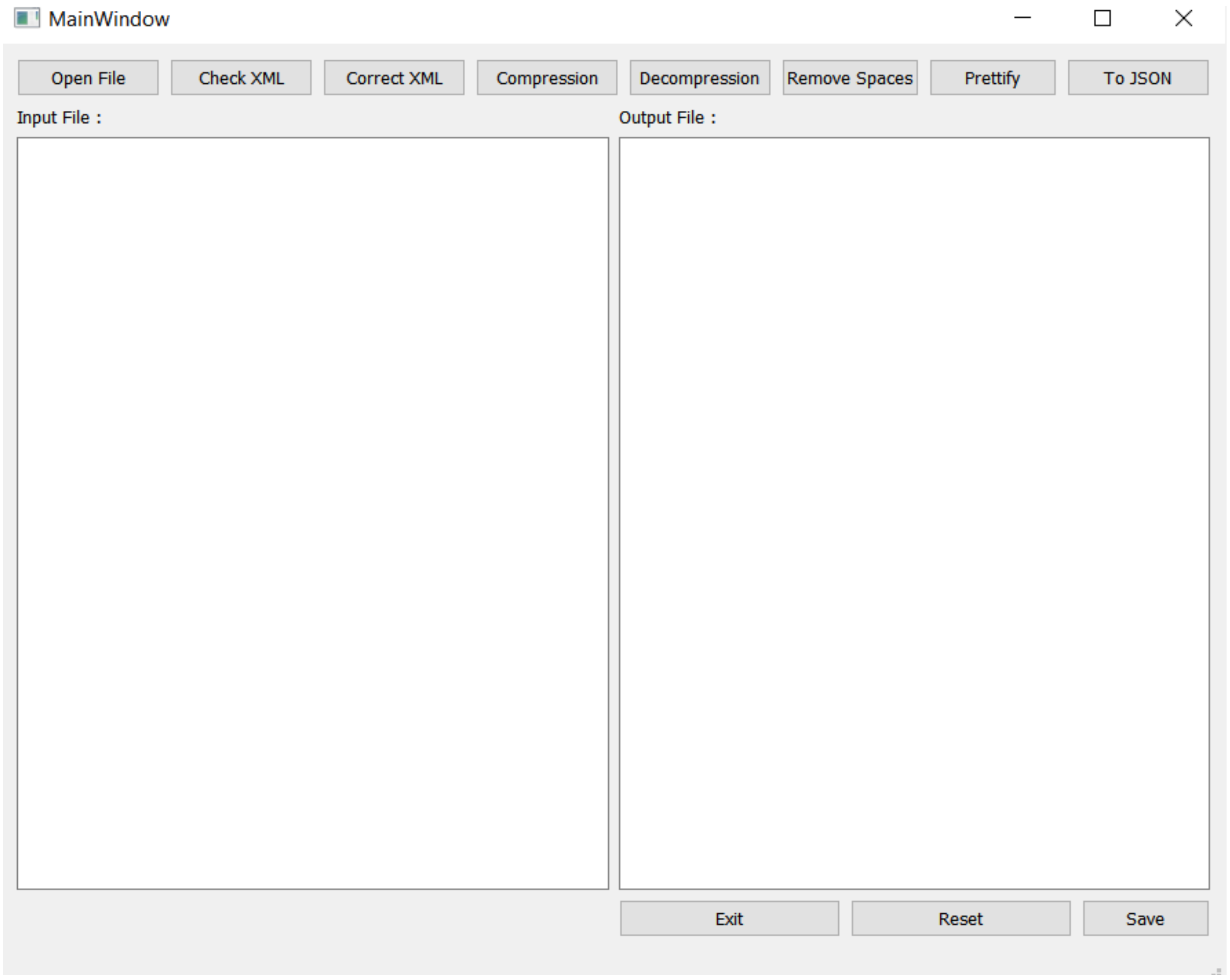
11- on_Decompression_Button_clicked()

Called when the Decompression button is clicked to Decompress the file.

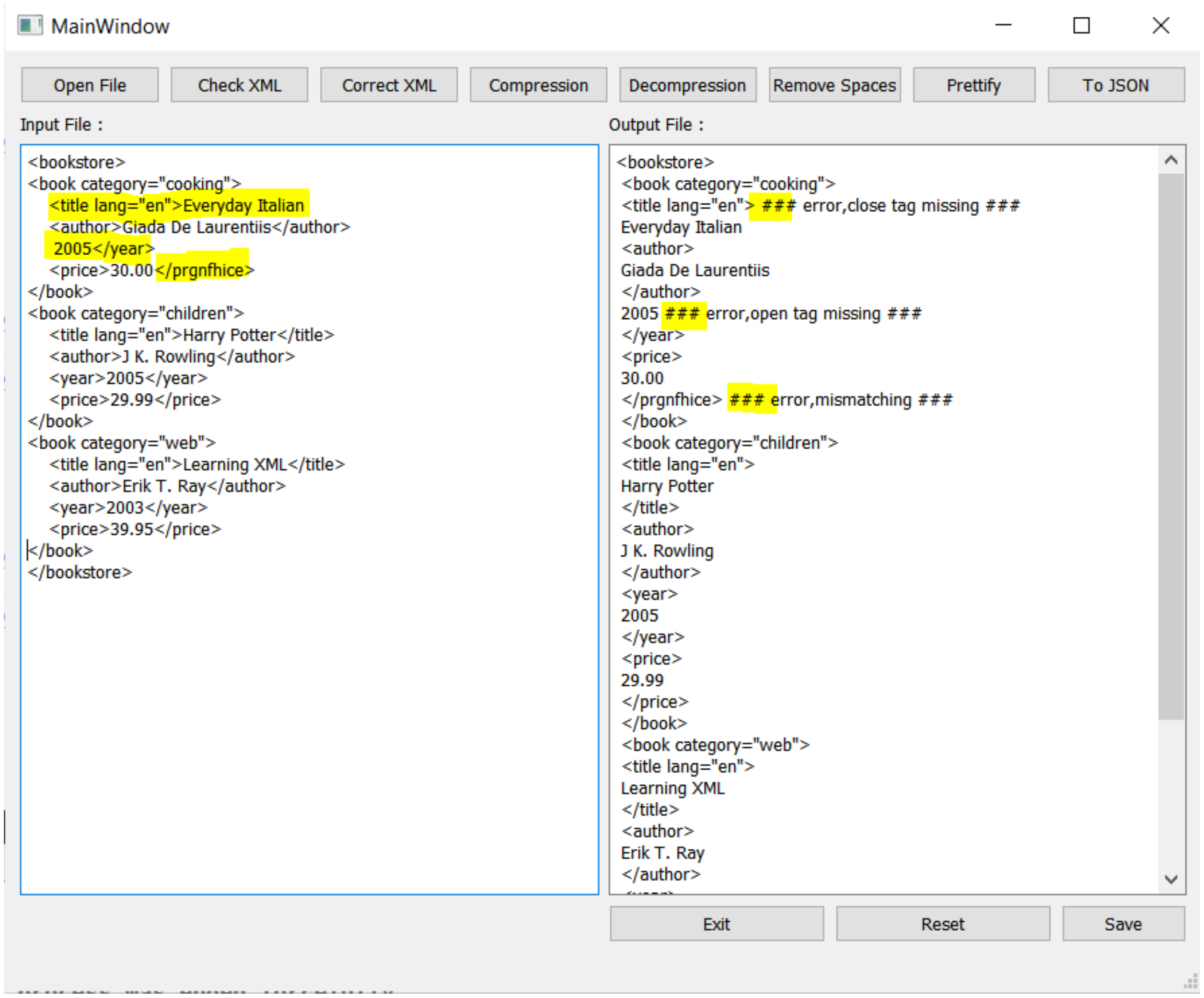
```
void MainWindow::on_Decompression_Button_clicked()
{
        QFile file(QFileDialog::getOpenFileName(this, tr("Open File"), QString(),tr("Text Files (*.comp)")));
                char file_data;
                QByteArray arr;
                if(!file.open(QFile::ReadOnly))
                {
                    QMessageBox::warning(this,"..","can not open the file");
                    return;
                }

                while(!file.atEnd())
                {
                  file.read(&file_data,sizeof(char));
                  arr.push_back(file_data);
                }
                file.close();
            QString txt = decompression(arr);
            ui->output_text->clear();

            ui->output_text->setPlainText(txt);
}
```

Graphical user interface:

1) General view:



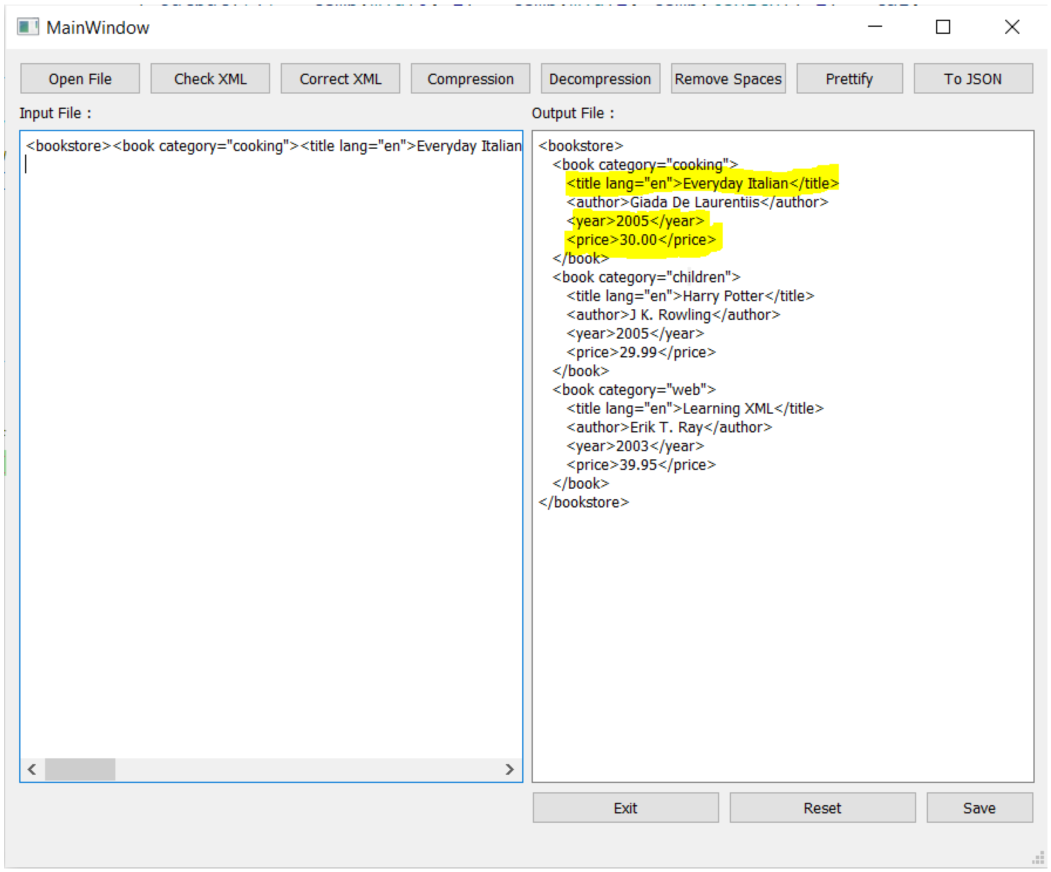
1) Check errors in XML files:



```
<bookstore>
<book category="cooking">
    <title lang="en">Everyday Italian
    <author>Giada De Laurentiis</author>
     2005</year>
    <price>30.00</prgnfhice>
</book>
<book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
</book>
<book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
</book>
</bookstore>
```

```
<bookstore>
 <book category="cooking">
 <title lang="en"> ### error,close tag missing ###
 Everyday Italian
 <author>
 Giada De Laurentiis
 </author>
 2005 ### error,open tag missing ###
 </year>
 <price>
 30.00
 </prgnfhice> ### error,mismatching ###
 </book>
 <book category="children">
 <title lang="en">
 Harry Potter
 </title>
 <author>
 J K. Rowling
 </author>
 <year>
 2005
 </year>
 <price>
 29.99
 </price>
 </book>
 <book category="web">
 <title lang="en">
 Learning XML
 </title>
 <author>
 Erik T. Ray
 </author>
```

2) Correct errors



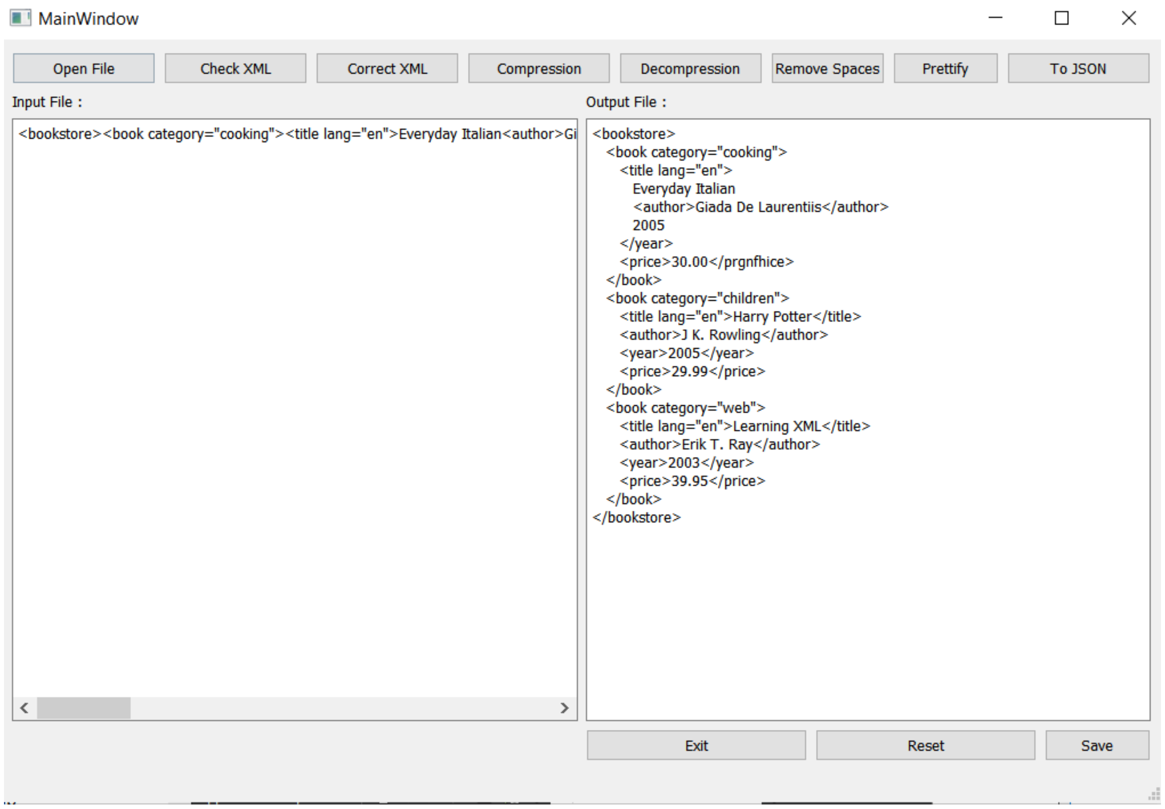
Input File :

```
<bookstore><book category="cooking"><title lang="en">Everyday Italian
```

Output File :

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

3) Prettify



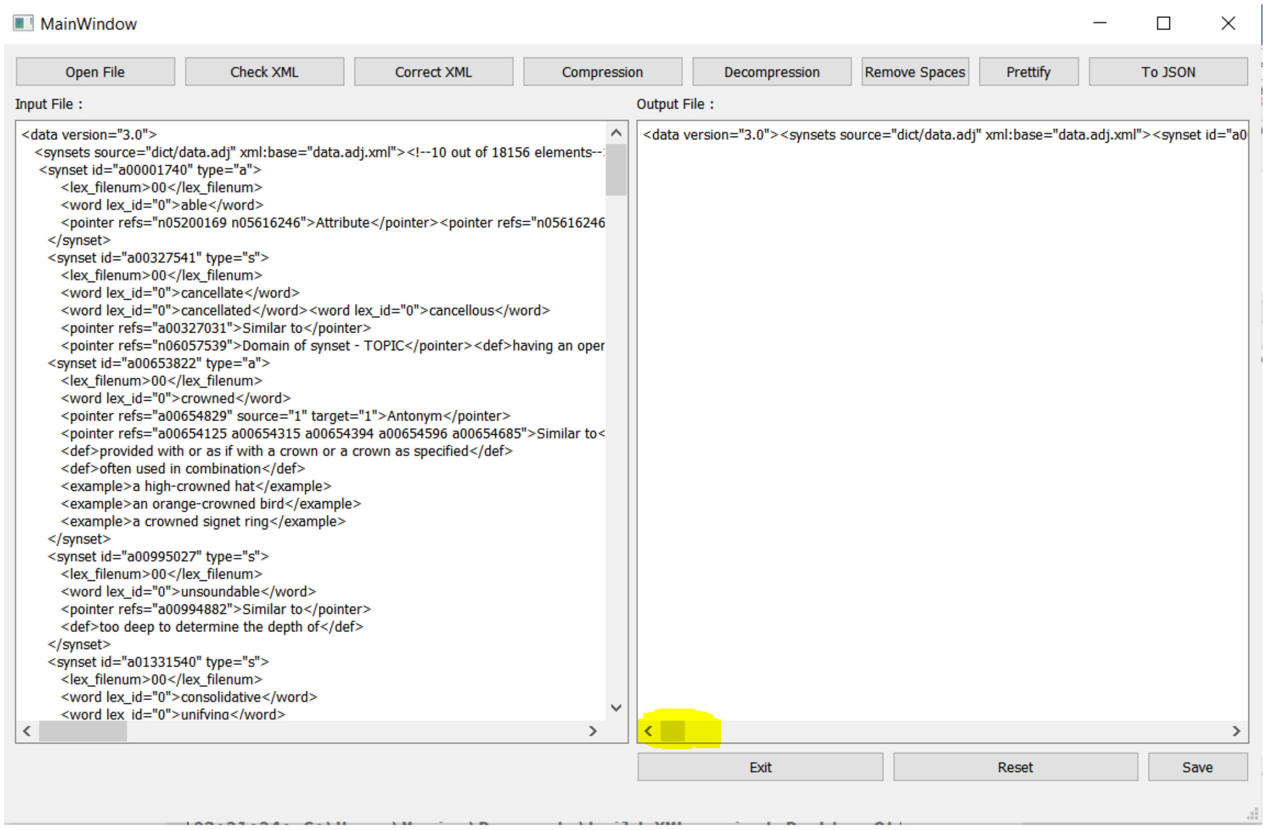
Input File :

```
<bookstore><book category="cooking"><title lang="en">Everyday Italian<author>Gi
```

Output File :

```
<bookstore>
  <book category="cooking">
    <title lang="en">
      Everyday Italian
      <author>Giada De Laurentiis</author>
      2005
    </year>
    <price>30.00</prgnfhice>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

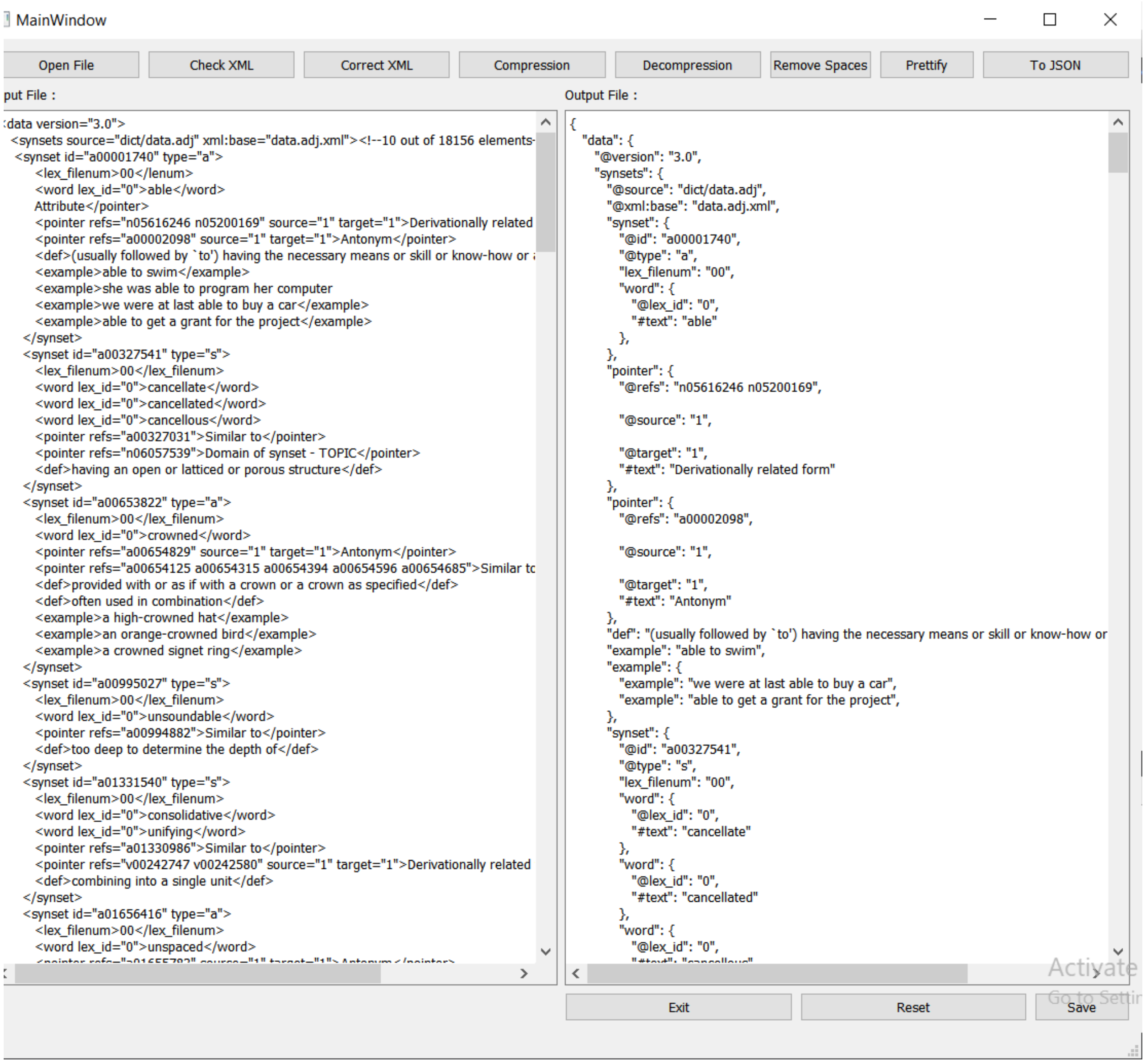
4) Remove Spaces:



```
<data version="3.0">
  <synsets source="dict/data.adj" xml:base="data.adj.xml"><!--10 out of 18156 elements--
   <synset id="a00001740" type="a">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">able</word>
      <pointer refs="n05200169 n05616246">Attribute</pointer><pointer refs="n05616246
    </synset>
    <synset id="a00327541" type="s">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">cancellate</word>
      <word lex_id="0">cancellated</word><word lex_id="0">cancellous</word>
      <pointer refs="a00327031">Similar to</pointer>
      <pointer refs="n06057539">Domain of synset - TOPIC</pointer><def>having an oper
    <synset id="a00653822" type="a">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">crowned</word>
      <pointer refs="a00654829" source="1" target="1">Antonym</pointer>
      <pointer refs="a00654125 a00654315 a00654394 a00654596 a00654685">Similar to<
      <def>provided with or as if with a crown or a crown as specified</def>
      <def>often used in combination</def>
      <example>a high-crowned hat</example>
      <example>an orange-crowned bird</example>
      <example>a crowned signet ring</example>
    </synset>
    <synset id="a00995027" type="s">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">unsoundable</word>
      <pointer refs="a00994882">Similar to</pointer>
      <def>too deep to determine the depth of</def>
    </synset>
    <synset id="a01331540" type="s">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">consolidative</word>
      <word lex_id="0">unifying</word>
```

```
<data version="3.0"><synsets source="dict/data.adj" xml:base="data.adj.xml"><synset id="a0
```

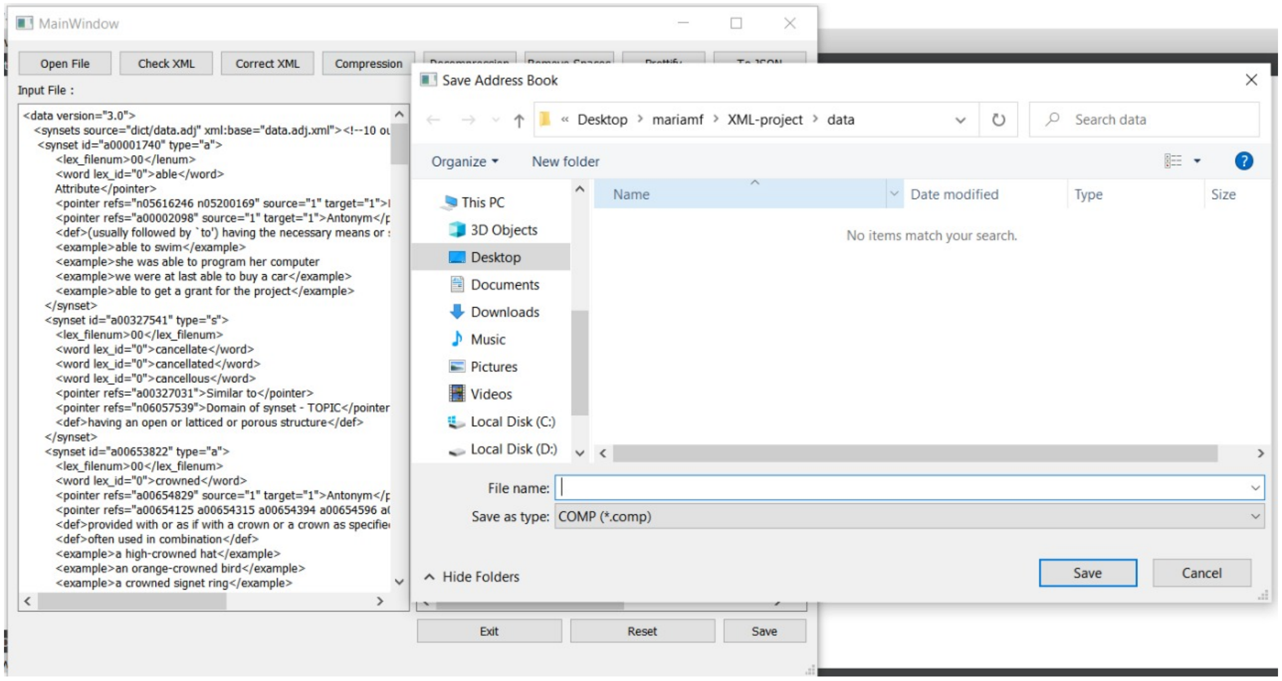
5) To JSON



```
:data version="3.0">
 <synsets source="dict/data.adj" xml:base="data.adj.xml"><!--10 out of 18156 elements-
  <synset id="a00001740" type="a">
     <lex_filenum>00</lenum>
     <word lex_id="0">able</word>
     Attribute</pointer>
     <pointer refs="n05616246 n05200169" source="1" target="1">Derivationally related
     <pointer refs="a00002098" source="1" target="1">Antonym</pointer>
     <def>(usually followed by `to') having the necessary means or skill or know-how or
     <example>able to swim</example>
     <example>she was able to program her computer
     <example>we were at last able to buy a car</example>
     <example>able to get a grant for the project</example>
   </synset>
   <synset id="a00327541" type="s">
     <lex_filenum>00</lex_filenum>
     <word lex_id="0">cancellate</word>
     <word lex_id="0">cancellated</word>
     <word lex_id="0">cancellous</word>
     <pointer refs="a00327031">Similar to</pointer>
     <pointer refs="n06057539">Domain of synset - TOPIC</pointer>
     <def>having an open or latticed or porous structure</def>
   </synset>
   <synset id="a00653822" type="a">
     <lex_filenum>00</lex_filenum>
     <word lex_id="0">crowned</word>
     <pointer refs="a00654829" source="1" target="1">Antonym</pointer>
     <pointer refs="a00654125 a00654315 a00654394 a00654596 a00654685">Similar tc
     <def>provided with or as if with a crown or a crown as specified</def>
     <def>often used in combination</def>
     <example>a high-crowned hat</example>
     <example>an orange-crowned bird</example>
     <example>a crowned signet ring</example>
   </synset>
   <synset id="a00995027" type="s">
     <lex_filenum>00</lex_filenum>
     <word lex_id="0">unsoundable</word>
     <pointer refs="a00994882">Similar to</pointer>
     <def>too deep to determine the depth of</def>
   </synset>
   <synset id="a01331540" type="s">
     <lex_filenum>00</lex_filenum>
     <word lex_id="0">consolidative</word>
     <word lex_id="0">unifying</word>
     <pointer refs="a01330986">Similar to</pointer>
     <pointer refs="v00242747 v00242580" source="1" target="1">Derivationally related
     <def>combining into a single unit</def>
   </synset>
   <synset id="a01656416" type="a">
     <lex_filenum>00</lex_filenum>
     <word lex_id="0">unspaced</word>
```

```
{
  "data": {
    "@version": "3.0",
    "synsets": {
      "@source": "dict/data.adj",
      "@xml:base": "data.adj.xml",
      "synset": {
        "@id": "a00001740",
        "@type": "a",
        "lex_filenum": "00",
        "word": {
          "@lex_id": "0",
          "#text": "able"
        },
      },
      "pointer": {
        "@refs": "n05616246 n05200169",

        "@source": "1",

        "@target": "1",
        "#text": "Derivationally related form"
      },
      "pointer": {
        "@refs": "a00002098",

        "@source": "1",

        "@target": "1",
        "#text": "Antonym"
      },
      "def": "(usually followed by `to') having the necessary means or skill or know-how or
      "example": "able to swim",
      "example": {
        "example": "we were at last able to buy a car",
        "example": "able to get a grant for the project",
      },
      "synset": {
        "@id": "a00327541",
        "@type": "s",
        "lex_filenum": "00",
        "word": {
          "@lex_id": "0",
          "#text": "cancellate"
        },
        "word": {
          "@lex_id": "0",
          "#text": "cancellated"
        },
        "word": {
          "@lex_id": "0",
```

6) Compression:



Input File :

```
<data version="3.0">
  <synsets source="dict/data.adj" xml:base="data.adj.xml"><!--10 ou
   <synset id="a00001740" type="a">
      <lex_filenum>00</lenum>
      <word lex_id="0">able</word>
      Attribute</pointer>
      <pointer refs="n05616246 n05200169" source="1" target="1">
      <pointer refs="a00002098" source="1" target="1">Antonym</p
      <def>(usually followed by `to') having the necessary means or :
      <example>able to swim</example>
      <example>she was able to program her computer
      <example>we were at last able to buy a car</example>
      <example>able to get a grant for the project</example>
    </synset>
    <synset id="a00327541" type="s">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">cancellate</word>
      <word lex_id="0">cancellated</word>
      <word lex_id="0">cancellous</word>
      <pointer refs="a00327031">Similar to</pointer>
      <pointer refs="n06057539">Domain of synset - TOPIC</pointer
      <def>having an open or latticed or porous structure</def>
    </synset>
    <synset id="a00653822" type="a">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">crowned</word>
      <pointer refs="a00654829" source="1" target="1">Antonym</p
      <pointer refs="a00654125 a00654315 a00654394 a00654596 a(
      <def>provided with or as if with a crown or a crown as specifie
      <def>often used in combination</def>
      <example>a high-crowned hat</example>
      <example>an orange-crowned bird</example>
      <example>a crowned signet ring</example>
```

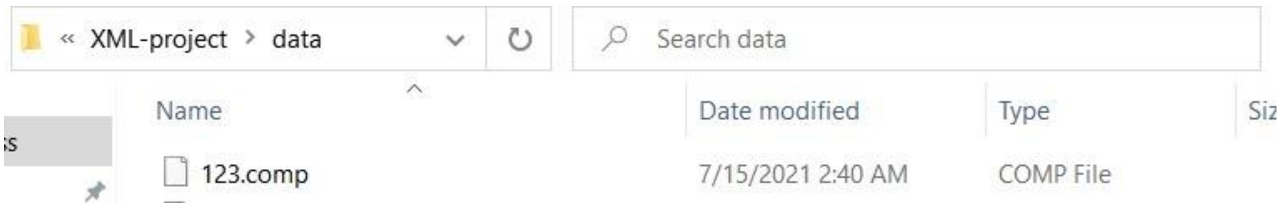
Save Address Book

Desktop › mariamf › XML-project › data

No items match your search.

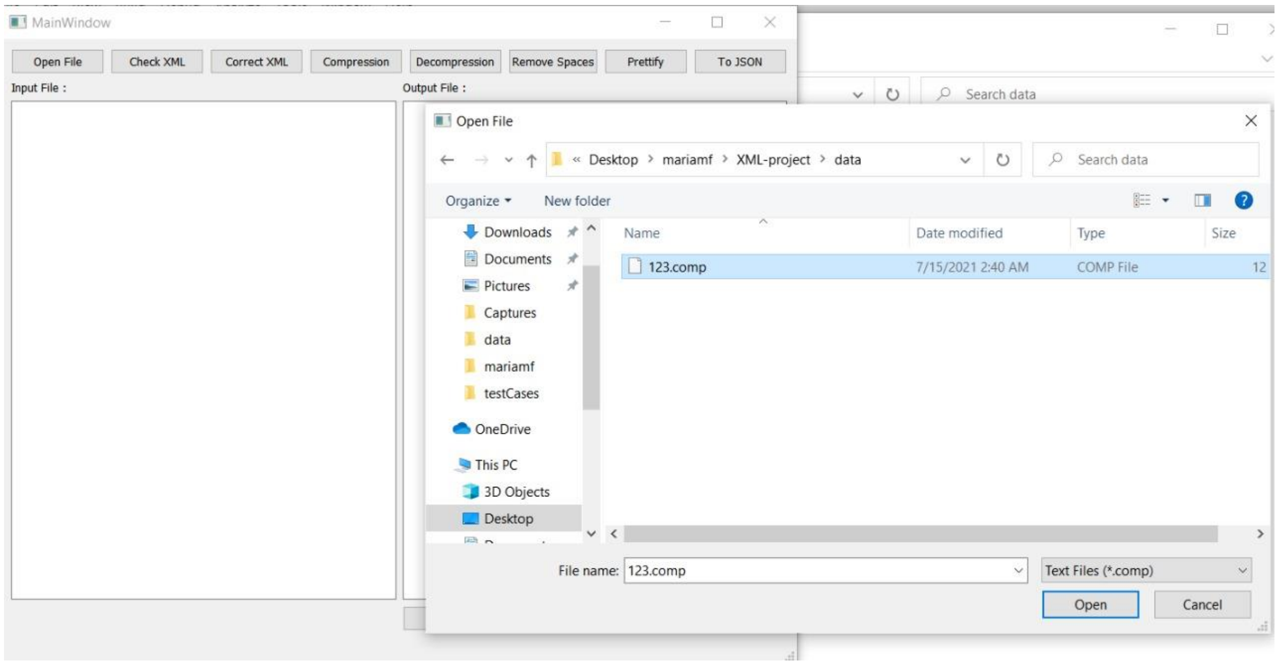
File name:

Save as type: COMP (*.comp)



Name	Date modified	Type
123.comp	7/15/2021 2:40 AM	COMP File

7) Decompression:



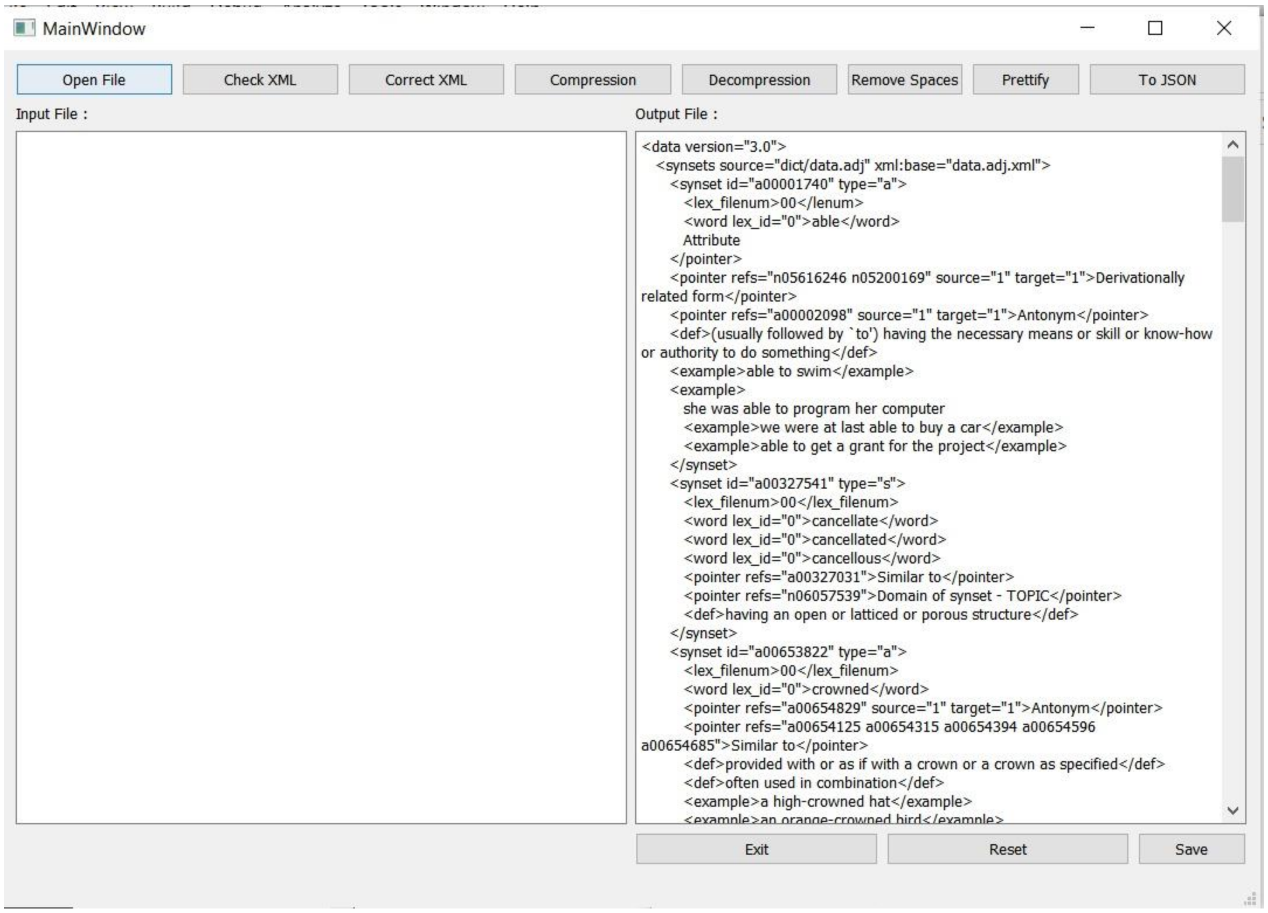
Open File

Desktop › mariamf › XML-project › data

Name	Date modified	Type	Size
123.comp	7/15/2021 2:40 AM	COMP File	12

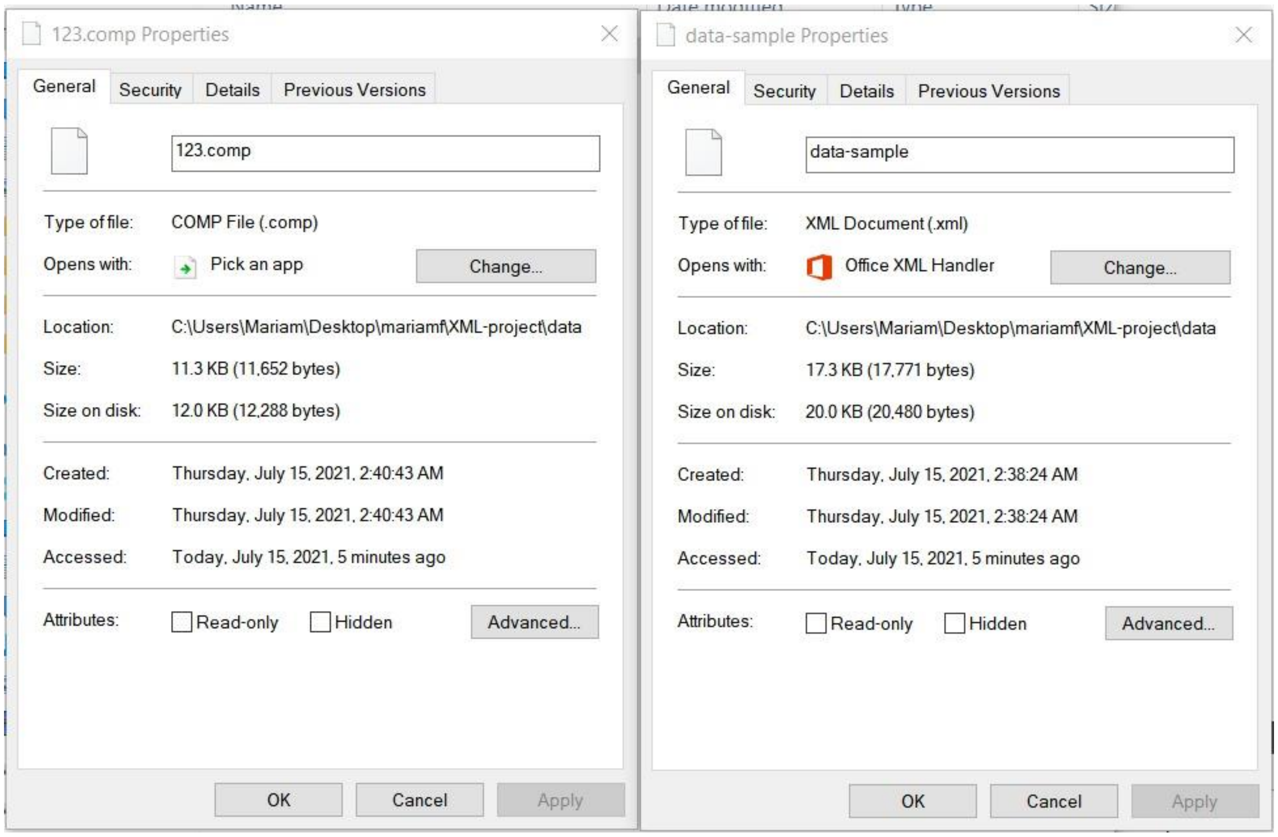
File name: 123.comp

Text Files (*.comp)



```
<data version="3.0">
  <synsets source="dict/data.adj" xml:base="data.adj.xml">
    <synset id="a00001740" type="a">
      <lex_filenum>00</lenum>
      <word lex_id="0">able</word>
      Attribute
    </pointer>
    <pointer refs="n05616246 n05200169" source="1" target="1">Derivationally
related form</pointer>
    <pointer refs="a00002098" source="1" target="1">Antonym</pointer>
    <def>(usually followed by `to') having the necessary means or skill or know-how
or authority to do something</def>
    <example>able to swim</example>
    <example>
      she was able to program her computer
      <example>we were at last able to buy a car</example>
      <example>able to get a grant for the project</example>
    </synset>
    <synset id="a00327541" type="s">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">cancellate</word>
      <word lex_id="0">cancellated</word>
      <word lex_id="0">cancellous</word>
      <pointer refs="a00327031">Similar to</pointer>
      <pointer refs="n06057539">Domain of synset - TOPIC</pointer>
      <def>having an open or latticed or porous structure</def>
    </synset>
    <synset id="a00653822" type="a">
      <lex_filenum>00</lex_filenum>
      <word lex_id="0">crowned</word>
      <pointer refs="a00654829" source="1" target="1">Antonym</pointer>
      <pointer refs="a00654125 a00654315 a00654394 a00654596
a00654685">Similar to</pointer>
      <def>provided with or as if with a crown or a crown as specified</def>
      <def>often used in combination</def>
      <example>a high-crowned hat</example>
      <example>an orange-crowned bird</example>
```

8) After vs Before:



123.comp

Type of file: COMP File (.comp)

Opens with: Pick an app

Location: C:\Users\Mariam\Desktop\mariamf\XML-project\data

Size: 11.3 KB (11,652 bytes)

Size on disk: 12.0 KB (12,288 bytes)

Created: Thursday, July 15, 2021, 2:40:43 AM

Modified: Thursday, July 15, 2021, 2:40:43 AM

Accessed: Today, July 15, 2021, 5 minutes ago

Attributes: ☐ Read-only ☐ Hidden

data-sample

Type of file: XML Document (.xml)

Opens with: Office XML Handler

Location: C:\Users\Mariam\Desktop\mariamf\XML-project\data

Size: 17.3 KB (17,771 bytes)

Size on disk: 20.0 KB (20,480 bytes)

Created: Thursday, July 15, 2021, 2:38:24 AM

Modified: Thursday, July 15, 2021, 2:38:24 AM

Accessed: Today, July 15, 2021, 5 minutes ago

Attributes: ☐ Read-only ☐ Hidden

03: Complexity of Operations

- **All constructors, destructors, setters, and getters used in the 3 classes are of complexity O(1)**
- **In XMLTree Class:**

  1- **convertingXMLFileIntoText & breakingFileTextIntoTags:**
  O(M); where M = number of lines in input file (not tags)

  2- **buildingXMLTree:**
  O(N); where N = number of tags + texts in input file

  ∴ **analyzeXMLFileText:** O(N+M)

  3- **generateJSONObject:**
  T(n) = a T(n/b) + n power d ➔ a = 1, b = 1, d =1
  a = b^d so complexity is O(n log n)

  4- **generateIndentationForJSON:**
  O(n); where n = indent counter

  5- **convertXMLFileIntoJSONFile:**
  O(n log n)

  6- **breakingStartTagIntoParts:**
  O(m); where m = number of characters in the whole input file

  7- **prettifyingXMLTreeFile & minifyingXMLTreeFile:**
  O(size); where size = number of internal blocks in XML tree

  8- **error_checking & error_correction:**
  average: O(2N)
  worst case: $O((N/2)^2/2) = O(N^2/8)$

- **In Compression and Decompression part:**

  1. **largest_match:**
  O(W); where W = size of window
  2. **minimize:**
  O(N); where N = size of file
  3. **compression:**
  O(N*W)
  4. **decompression:**
  O(Nfile); where Nfile = size of decompressed data

04: References:

[1] "Processing XML with E4X". Mozilla Developer Center. Mozilla Foundation.
[2] SOLOMON,D.: Data Compression, The Complete Reference., Springer, New York, 1998
[3] BELL, T. C., CLEARY, J. G., WITTEN, I. H.: Text Compression, Prentice Hall Advanced Reference Series, 1990