

**Team Members**

محمد موسى محمد موسي 1701297

محمد نبيل إبراهيم محمد راضي 1701298

محمد مدحت محمد إبراهيم 1701287

انطوان زكريا عدلى ابراهيم 1700317

**1.Background:**

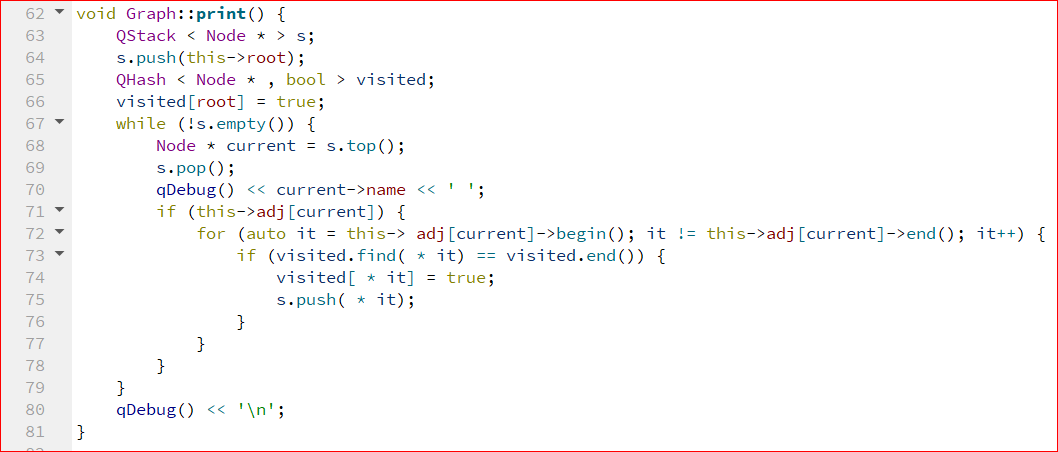
**2.Implementation details:**

**A-Backend functions:**

**1. Graph Implementation:**

****

****

****

Graph is a class that implements the XML tree that’s going to be built from the XML file. First we implement a Node class which describes each node in the tree, it has a tag name, value, attributes, a Boolean value that indicates whether it’s a self-closing tag and another Boolean value that indicates whether all the child nodes of this node has the same tag name, that value is useful in conversion to JSON.

Match pointer is a class that helps with compression functions, as it helps us return two values from “\_largest\_match” function, we will discuss later in compression functions.

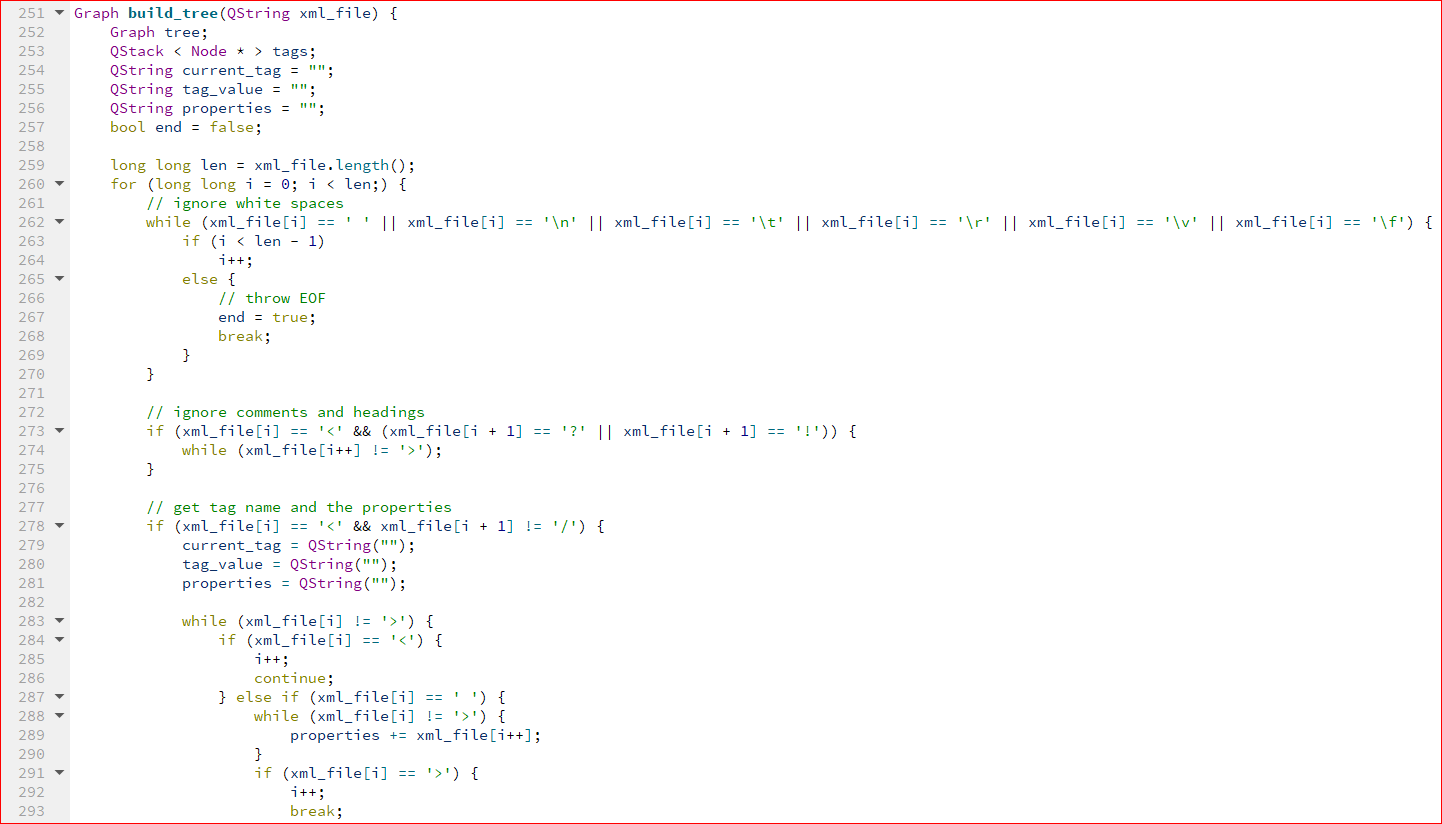
The Graph class itself has four properties, a pointer to the root of the tree, the adjacency list, number of edges and number of nodes. The adjacency list is implemented using a hash map that has Node pointers as keys and returns a pointer to a linked list of Node pointers, if the Node is a leaf node the pointer to the Linked List will be a null pointer.

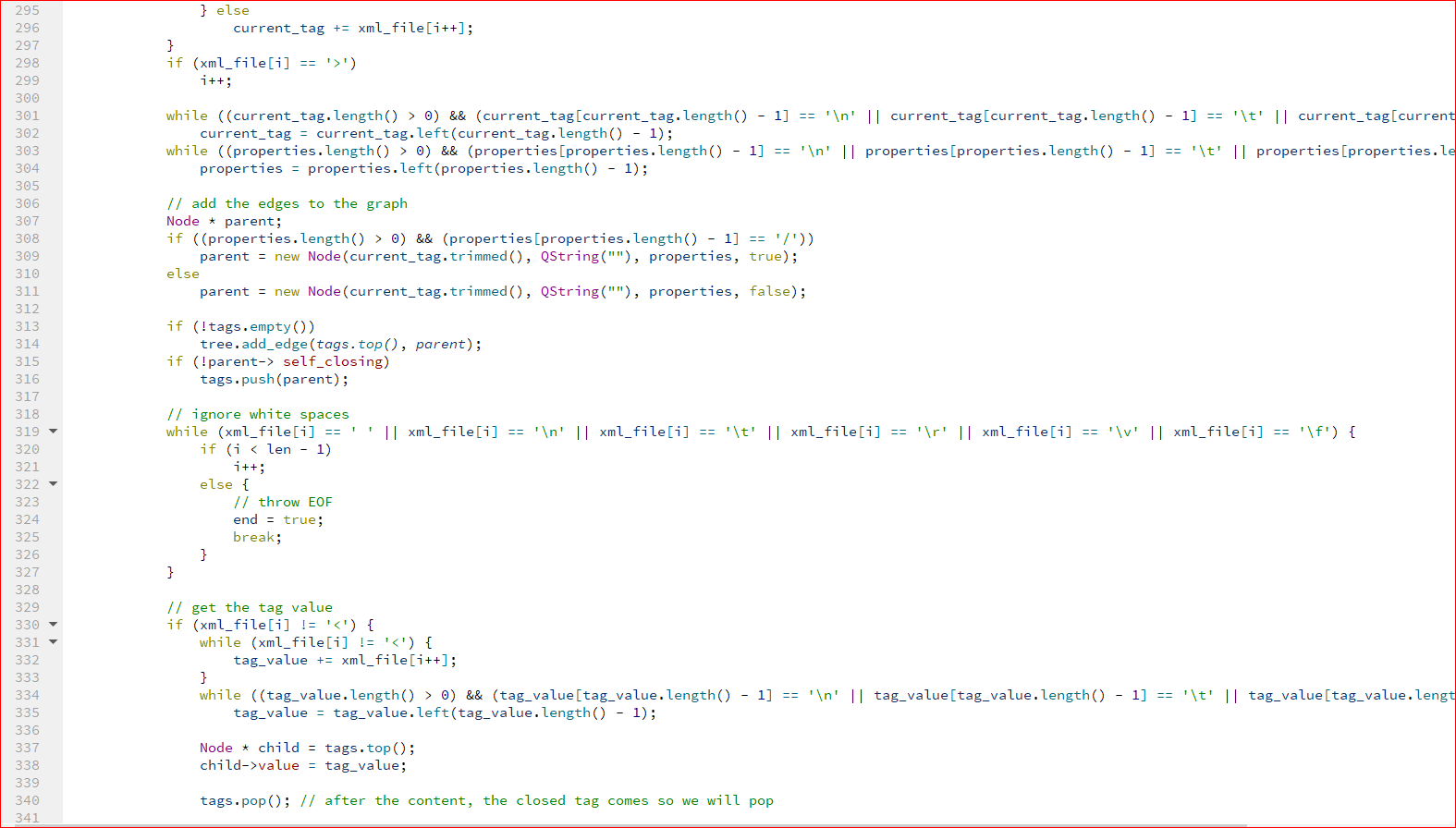
The constructor initialized number of nodes and edges to zero in O(1) time, the destructor deletes the allocated nodes and linked lists in O(N ) time, where N is the number of nodes.

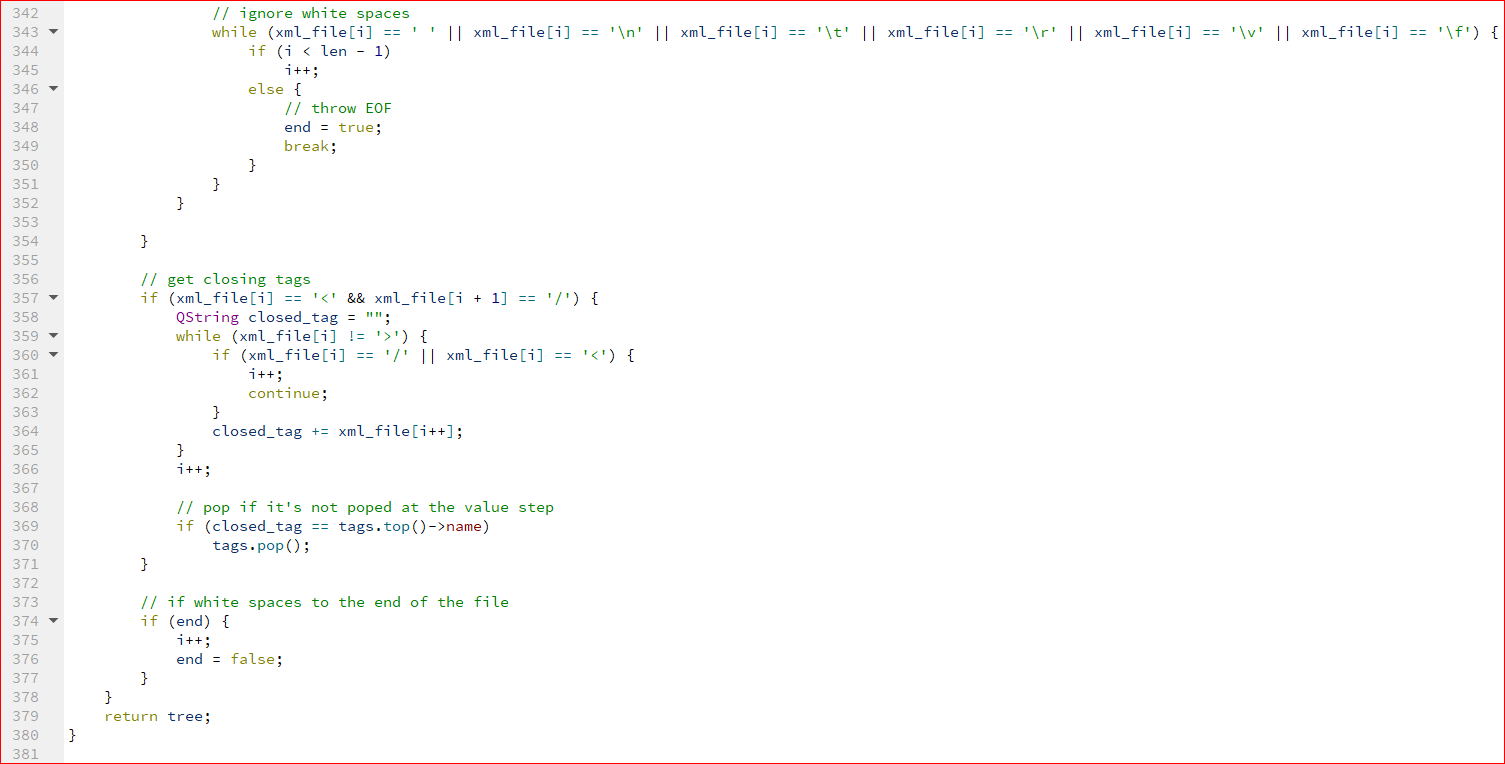
The function “add\_edge” takes two Node pointers, a “from” node pointer and a “to” node pointer. It adds the “from” Node pointer to adjacency hash map if it isn’t already in the hash map and appends the “to” pointer to the linked list associated with “from” pointer, it from pointer doesn’t have a linked list associated with it then a new one is created for it. The function takes O(1) time on average. It also assigns the first “from” pointer to be the pointer to the root of the tree and it increments the number of edges and nodes.

The function “print” uses iterative depth first search to output the Graph (or tree) in O(E+N), where E is the number of edges and N is the number of nodes.

**2. Build tree function implementation:**

****

****

****

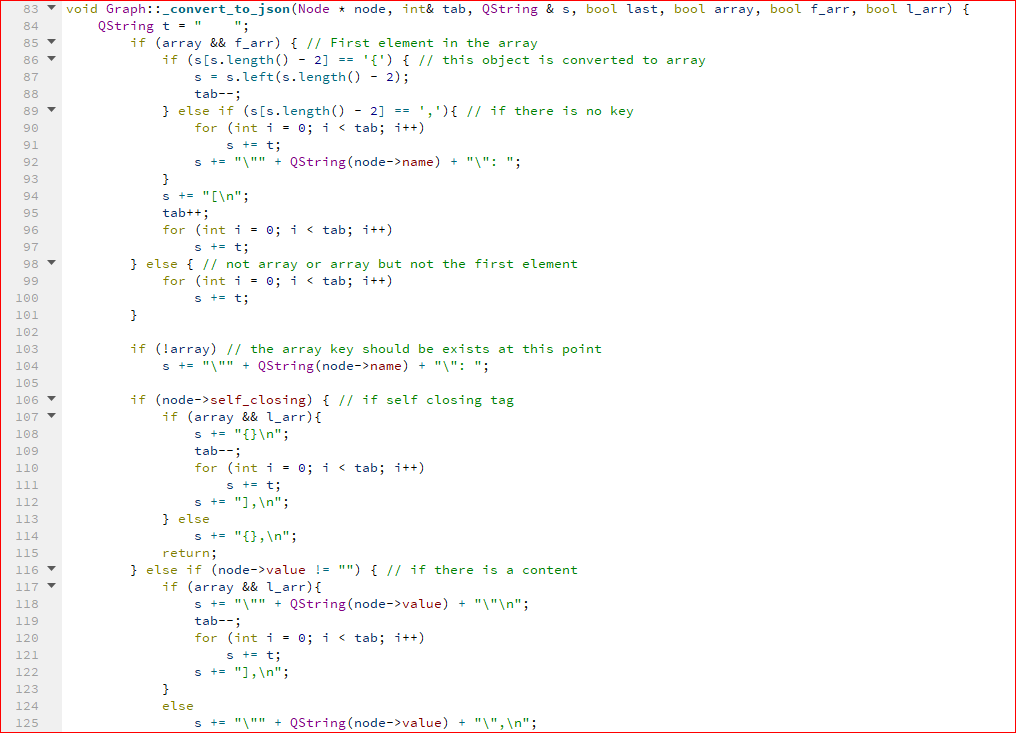
Build tree function is used to convert the xml string to a tree with a root and child nodes we take the xml as a parameter and loop over it, we ignore meta data and comment and not include it in our tree, we check the opening tags and get the tag name and the attributes and create a node then we check if that was a self-closing tag we don’t need to push that node to the stack, we make that node a child to the top node of the stack, and if the stack is empty we don’t add any thing and with next node add\_edge function will make the first node the root of our tree, then we check if there is a value we take it and edit the node value with that value and pop the node from the stack as xml tag with a value can’t also has child tags.

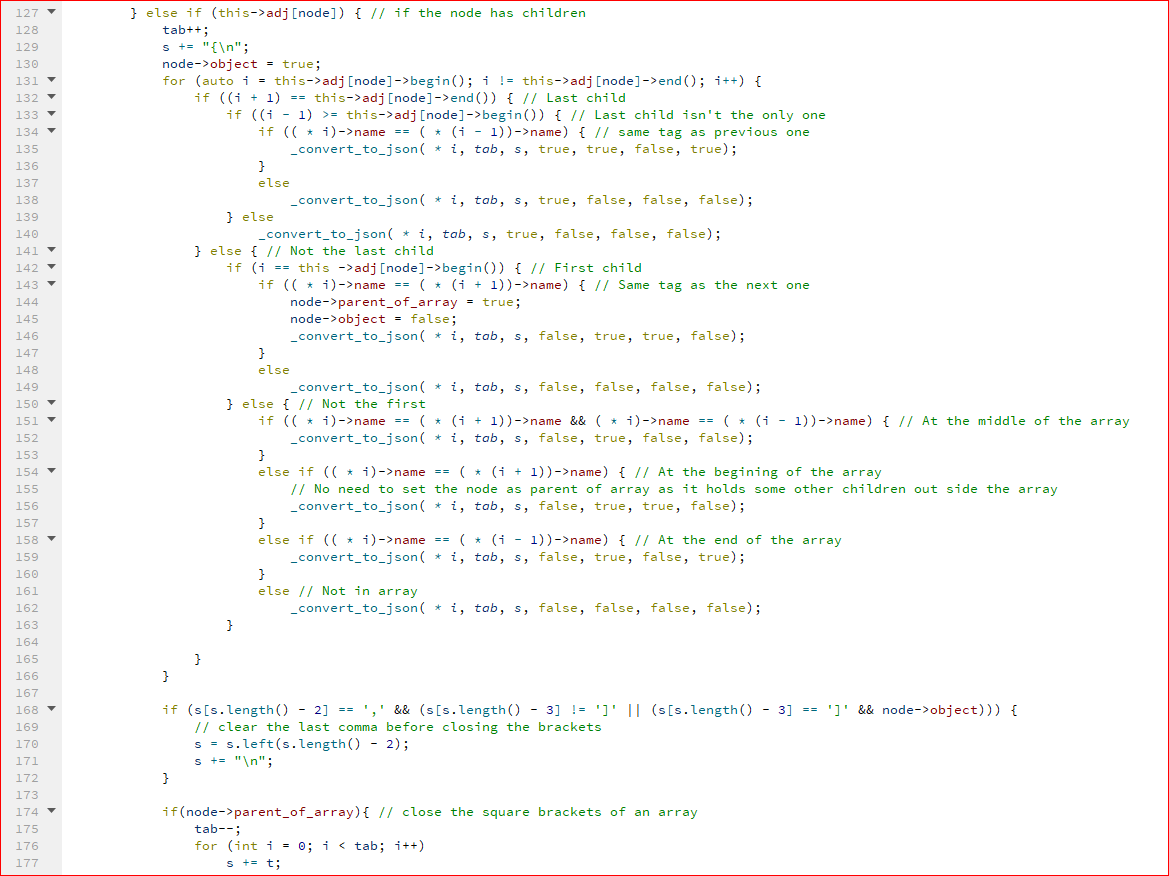
Th interesting part here we check if that node has an attributes we make a new node for each attribute and make it a child to that node, the xml tag doesn’t have a value ang child tags at the same time so if the node has a value and her attributes are converted to child nodes, we put the value in a new node it also becomes a child to that node, we can determine if the child node is an attribute by making its name property begins with @ then the attribute name, and if it is a value node by making its name property begins with # then the xml tag name.

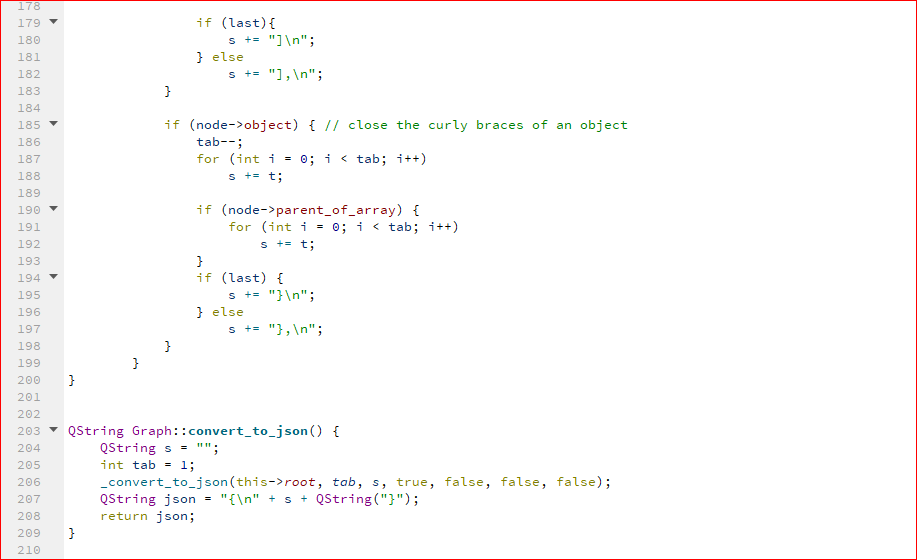
Then we check the closing tags to pop nodes from the stack.

build tree has a complexity of O(n) where n is the length of the xml string.

**3. Convert to JSON function Implementation:**

****

****

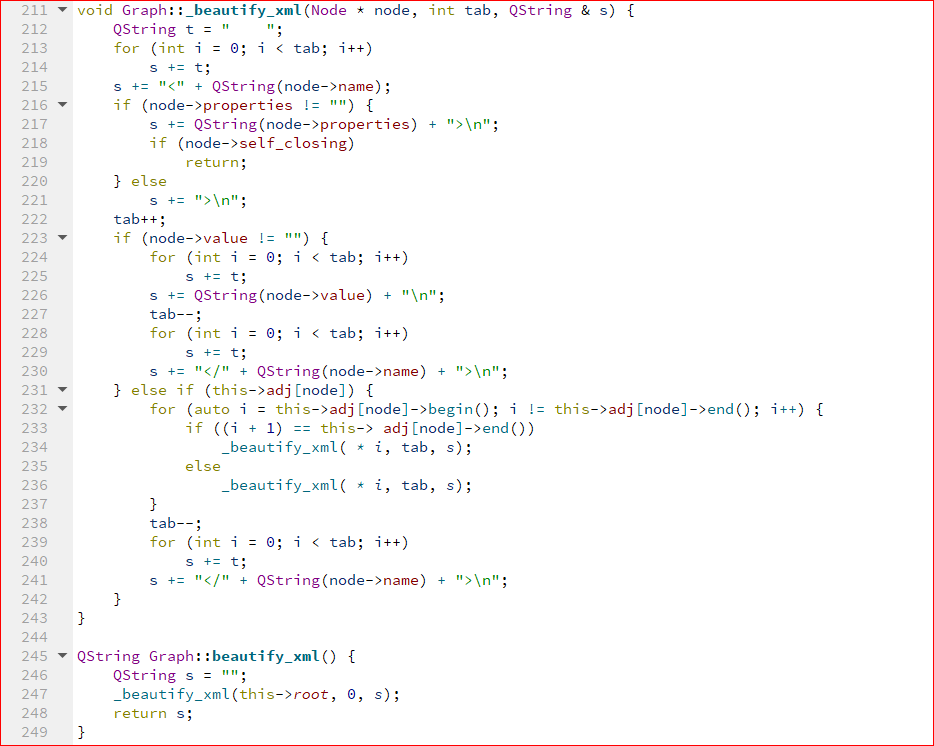
****

The convert\_to\_json function has no parameters to easily interface with GUI, it sets the parameters to the real function which converts the graph to json (\_convert\_to\_json), and after this we insert the json curly braces to the final string.

\_convert\_to\_json is a recursive function that has 7 parameters the tree root (tree is special kind of graphs), int tab to make sure indentations are correct at the first call it quals to 1 so we have one tab after the opening curly brace, the string which we write the json in it at the first call it’s an empty string, and Boolean variables to identify if that node the last node so we don’t need to insert a comma at the end of this node, and if the node is included into array and if true if this node is the first element in the array to open the square brackets and if the node is the element in the array to close the square brackets. At he beginning of the function we check if that node is the first element of an array so we need to open the brackets and check if it has an opening curly brace we remove it, then we put the indentations, after that we put the tag name as a key, and check if that key has value we put that value, and if has children we need to open an object curly brace and determine if that child is an array element or not by checking the tag name of the previous and the next children and if array we determine if that child is the first or the last or a middle element in the array and base on the state of the child recursively call the function on it, after we are done with the children nodes we need to close a square bracket or a curly brace based on if it was an array or not, also we delete the last comma before the closing bracket and if it was the last node in our graph we don’t append a comma after the closing tag.

\_convert\_to\_json has a complexity of O(N) where N is the number of nodes in our graph.

**4. Beautify xml function Implementation:**

****

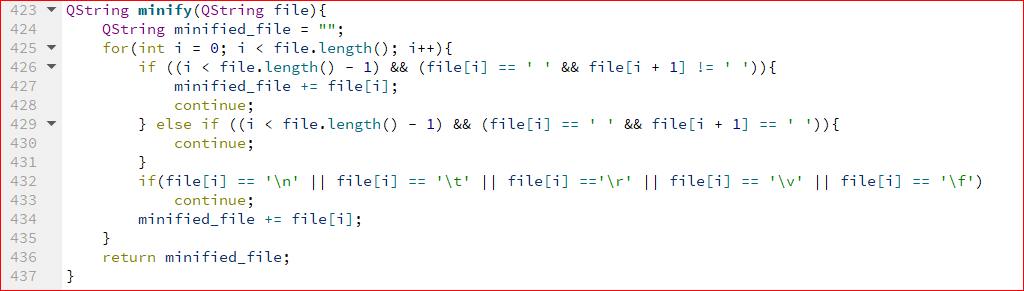
Like convert\_to\_json we have a beautify\_xml function that takes no parameters to easily interface with the GUI and it sets the parameters of the actual function where all the work done inside of it (\_beautify\_xml).

\_beautify\_xml is a recursive function that has 3 parameters the tree root (tree is special kind of graphs), int tab to make sure indentations are correct at the first call it quals to 0 so our xml begins at the beginning of the line, the string which we write the json in it at the first call it’s an empty string.

A the beginning we check if that node is not an attribute so we open a tag and put its name, the if the has an attributes children we put its attributes, then we check if it’s a self-closing tag we close the tag and the function ends here, and if not we check if the node has a value we put its value and close the tag, and if has a children not value we recursively call the function on its children which is an xml tag not an attributes to that node, when we done with its children we need to close the opened tag.

\_beautify\_xml has a complexity of O(N) where N is the number of nodes in our graph.

**5. Minify function implementation:**

****

The minify function takes the XML file as a string and outputs the file without any unnecessary spaces and without any new lines. This is done by iterating over the string and checking for unnecessary spaces and new lines, then adding the rest of the content to a new string and then return this string in O(N) time, where N is the size of the XML file.

**6. Compress and decompress functions implementation:**

****

Compression here is done by LZ77 compression algorithm[[1]](#endnote-1), the algorithms aims to replace certain byte streams by pointer to previous equivalent byte streams. Certain terms like “window” and “look-ahead buffer” are associated with the algorithm; please refer to the reference for more information.

The function “compress” takes in a file as a string, then, it minifies the file and converts it to a byte array. After that it iterates over the byte array and calls “\_largest\_match” function , which we will discuss in the next section, to return the Begin (B) pointer and the Length (L) pointer for each byte, it’s important to note the size of the windows cannot exceed 255 because otherwise there will be an overflow, since an byte can at most hold 256 values. For each byte we append the B and L pointers (if necessary) and the character at the look ahead buffer (if necessary) to a new byte array called “compressed\_byte\_array”. We then return the compressed byte array. This operation takes O(N + Nmin\*O(\_largest\_match)), where N is the size of the XML file and Nmin is the size of the minified file. But O(\_largest\_match) = O(W), where W is the size of window, this will be discussed in the next section. So the complexity is O(N + Nmin\*W).

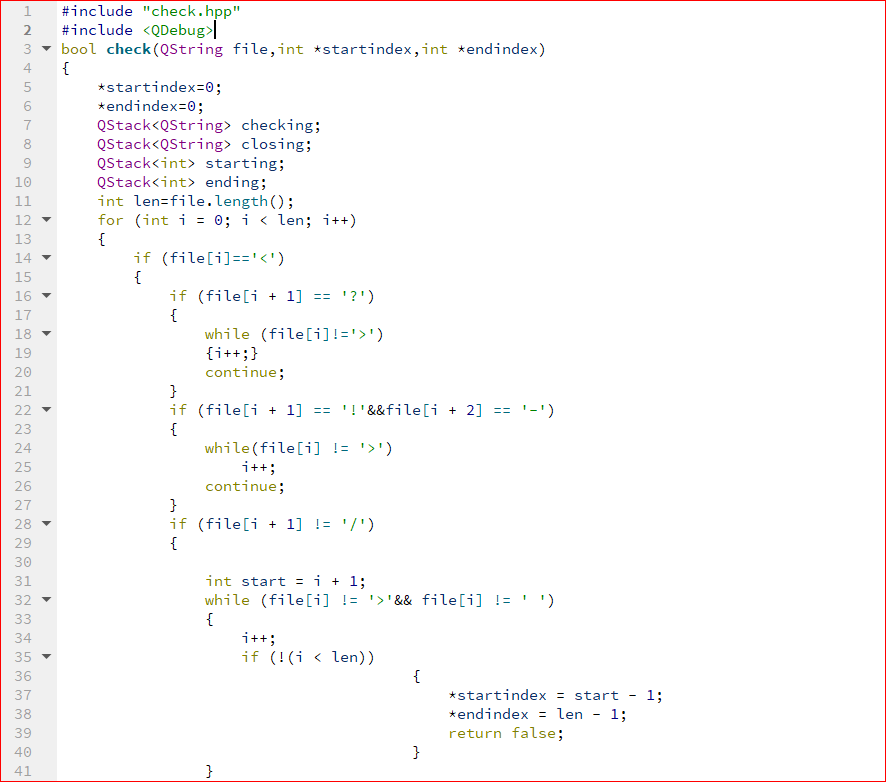
The “decompress” function builds the XML file as a string from the passed compressed byte array. It iterates over the byte array, reads the B pointer, if it’s zero then we just append the character to the output string, if not then we read the value of L pointer, and then we append the byte stream resulting from B and L to the output string. This operation takes O(N), where N is the size of decompressed information.

**7.Largest match function implementation:**

****

This is a helper function for “compress”. It takes and iterator that points at the start of the window and the one that points at the start of the look-ahead buffer. It then iterates over both the window and the look ahead buffer, if it finds a match then we write the B and L pointers in the variable “current\_match\_pointer”, if this current match pointer has a larger length than the largest match pointer, we replace “largest\_match\_pointer” with “current\_match\_pointer” and then we continue iterating. When we then stop when the whole window is finished and we return ”largest match pointer”. This function is of O(W), where W is the size of the window.

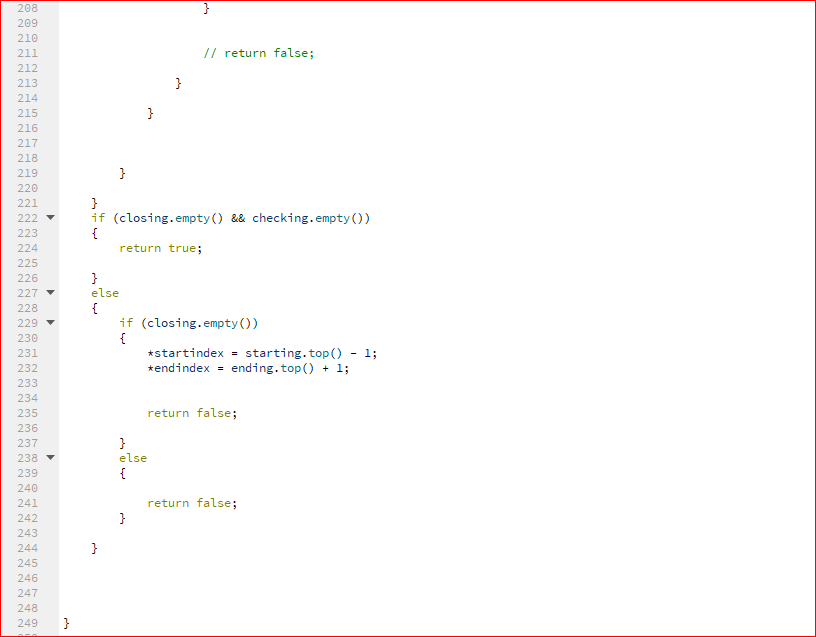
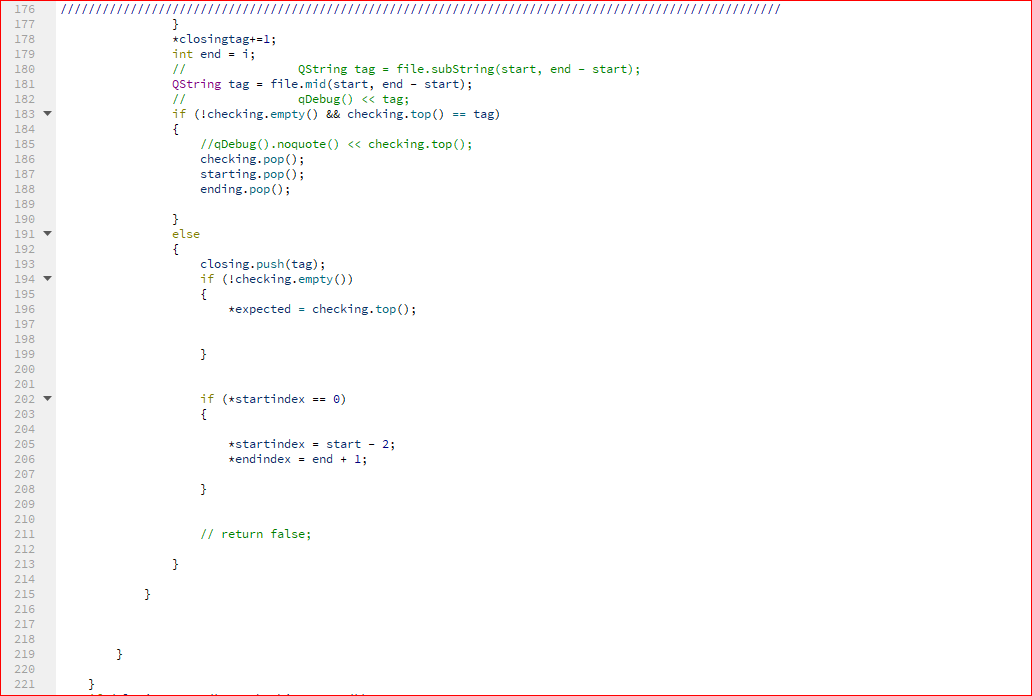
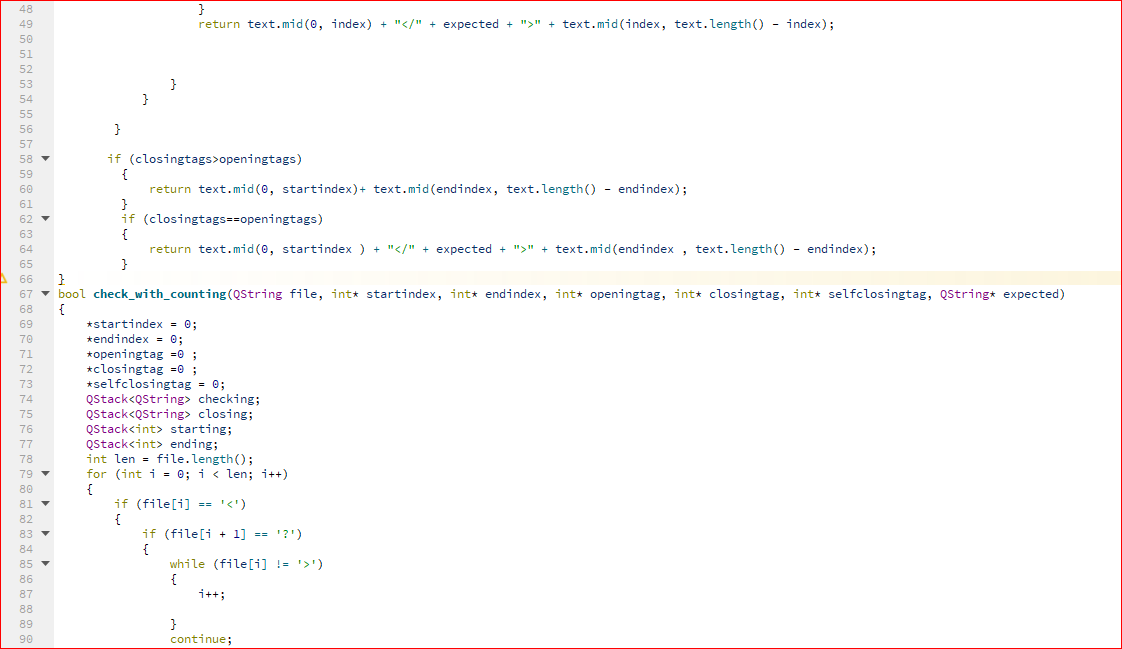
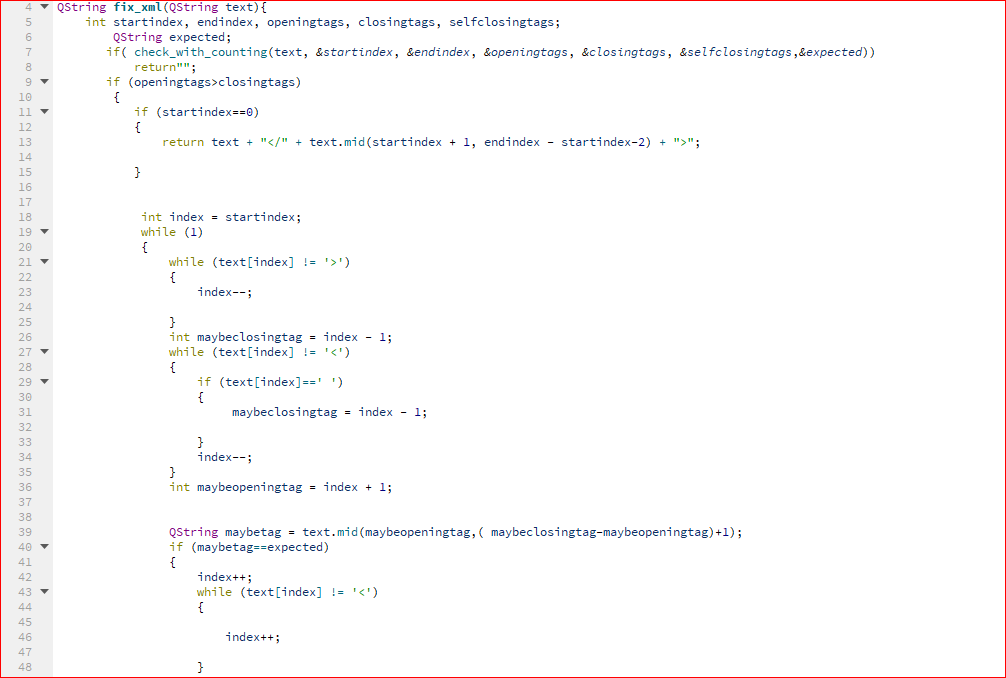
**8.Check errors Function:**



****

Checking errors depends mainly on the stack data structure.  
we search for the tags inside the string, if opening tag was found it is pushed into the stack, if it is a self-closing so we ignore it and it is not pushed into the stack, if it was a closing tag so we get the top of the stack if it was same as closing so we pop it from the stack, but if the closing was different from the opening tag so there is an error in the consistency of the xml and the closing tag is pushed in another stack called closing and at the end of the function we check on the two stacks if they are empty so the xml is consistent and it returns true.

**9.Fix errors Function:**

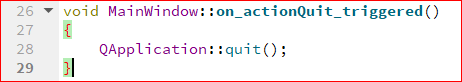


Fixing function mainly depends on another version of checking function which does the same algorithm as check consistency but returns some parameters which we will need in our fixing algorithm like number of closing and opening and self-closing tags also the place of the error in the string and also a string called expected which refers to the correction which may be used to correct the error .First fixing function call the check function to assure that there is an error in the xml, if no error is found so it returns an empty string, but if error was found so error should be 1 of three a missing closing tag and this will results than number of opening tags will be greater than number of closing tags, Missing opening tag and this will result that number of closing tag will be more than opening tags, Or miss matching tags and this will result that number of opening tags will be equal to closing ones .

First the case of miss matching we return the first part of the string untill the error is found then add the right closing tag from the string that is called expected then remove the wrong one and add the rest of the string, Second in the case of missing opening tag ,We will find that the error is in the closing tag of that missing opening tag so we had two options first one is to add the opening tag right before the closing tag but it will be meaningless or add it before the closing one but right after the first tag that is found before the closing one and this may also be wrong in case of nested tags so we went to the second option and it is removing the closing tag. Third a missing closing tag: First after we know the place of the error We go back in the string until we find the opening tag which we are searching for then we go forward until the first tag we find and we add the Closing tag .

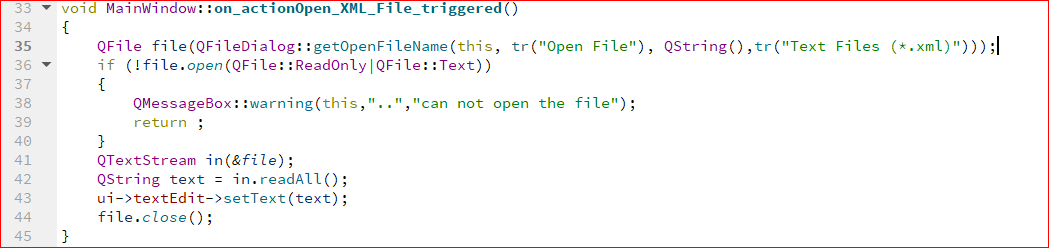
**A-Frontend functions:**

**1.Action Quit:**



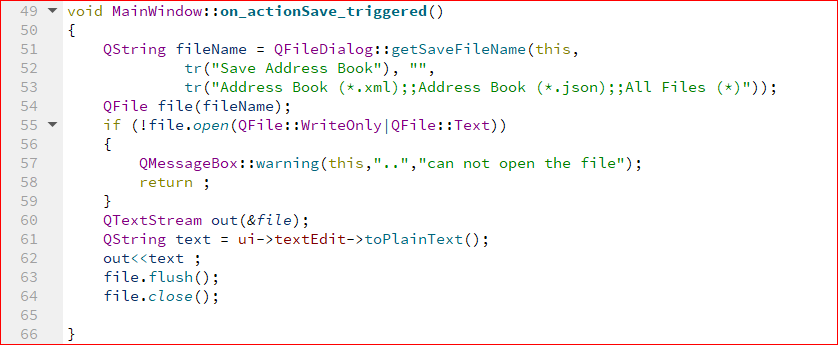
Here we call the built in function quit to close the program.

**2.Action Open:**



Here we open the file and make only xml file visible to the user to open then checking that the file is opened and if it’s not opened a warning massage is displayed to the user that the program cannot open the file. Then the text in the file is read using readAll() function so we can display the text in the xml file in the text area in line 43.

**3.Action Save:**



**4.Action Convert to json:**



First in this function we check if the xml file is consistent or not using check function (toPlainText() returns the text in the text area as a string) then build the graph with the text then convert this text to json then writing the output in the text area and if the xml file is not consistent a warning will be opened that the file is not consistent.

**5.Action Copy, Paste, Undo and Redo:**



in these actions we use the built in functions (copy, paste, undo, redo)

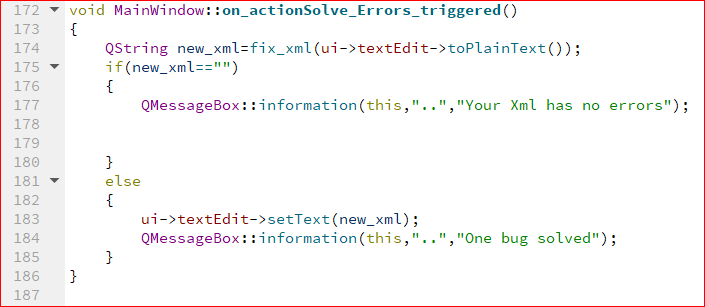
**Note:** copy function works on the selected text in the text area and undo or redo functions work if the user adds a new text.

**6.Action Check Consistency:**

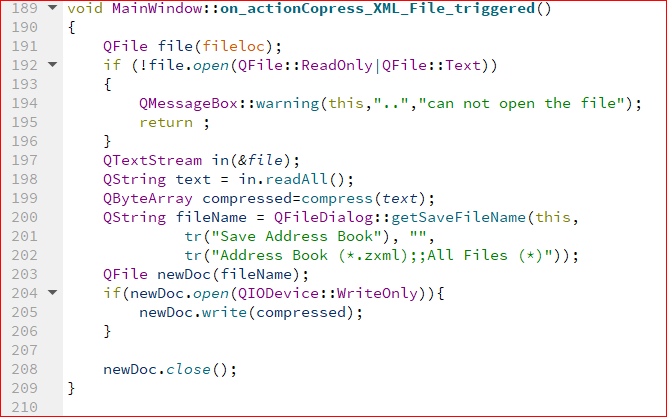


Here we check if the xml file has no errors so a massage is displayed that the file is consistent and if it’s not then we loop on the text till we find the error then coloring the text before the error with black, the error with red and the text after the error with black. (we set the color using the function (setTextColor) then append the wanted text to be colored to the text area)

**7.Action Solve Errors:**

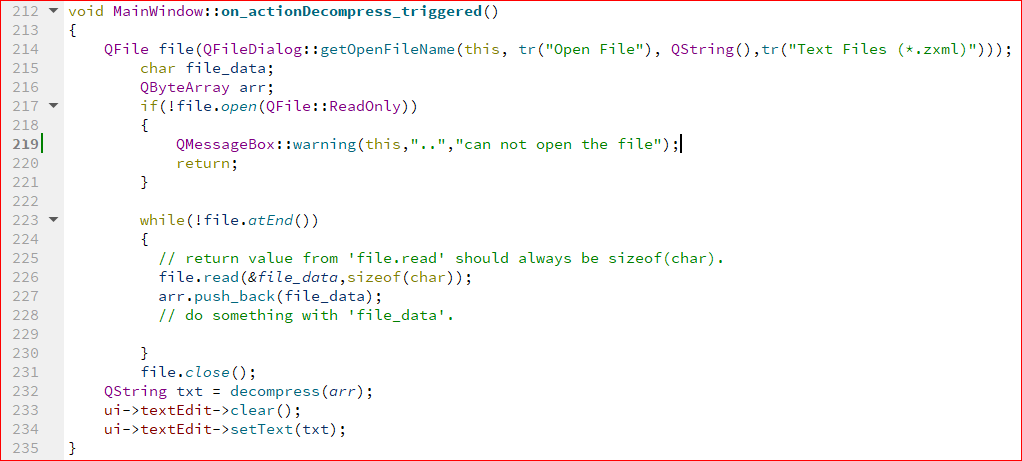


**8.Action Compress XML File:**



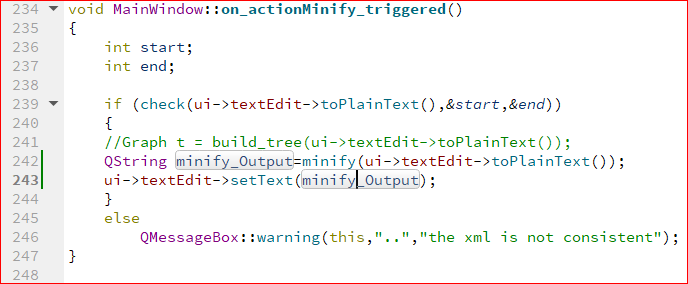
Here we take the text from the text area , passing it to the compress function and saving the compressed file as a **.zxml** so the user can find the compressed file easily.

**9.Action Decompress File to XML:**



In this function first the user can open a .zxml file to be decompressed and if the file is not opened a warning is displayed to the user that it’s not opened then parsing the data that is in the file into characters and pushing it into QByteArray then decompressing this array and displaying the text in the text area.

**10.Action Minify XML File:**

****

First in this function we check if the xml file is consistent or not using check function (toPlainText() returns the text in the text area as a string) then build the graph with the text then minify this text then writing the output in the text area and if the xml file is not consistent a warning will be opened that the file is not consistent.

**3.Complexity of operations:**

1. L7ZZ compression algorithm https://docs.microsoft.com/en-us/openspecs/windows\_protocols/ms-wusp/fb98aa28-5cd7-407f-8869-a6cef1ff1ccb [↑](#endnote-ref-1)