
**Faculty of Engineering, Ain Shams University**

**CSE331s - Data Structures and Algorithms**

# <<<<XML EDITOR PROJECT>>>>


| Name | ID |
|---|---|
| Saher Attia Farid | 1901612 |
| Hazem Hamdy Abd El-Sattar | 1900465 |
| Mohamed Emad Mostafa | 1900304 |
| Mohamed Ashraf Ibrahim | 1901607 |
| Ahmed Reda Kamal | 2001177 |

# Introduction:

This is XML editor that allows user to load, edit, save XML files and many other features with XML files. This XML editor can detect errors, show the number of errors, point to places of errors, and fix them to make the XML file consistent. Also, user can format XML file, minify it, convert it into JSON file, compress it to reduce its size ,and able to decompress it, user can generate graph visualization for social network represented by xml file, and get some important analysis from it.

## Features:

- **Browsing XML Files**
- **Checking the XML consistency**
- **Detecting & Correcting Error in consistency in XML File**
- **Format XML Files**
- **Minifying (Reducing file size by removing extra spaces).**
- **Compressing XML File**
- **Decompressing XML File**
- **Converting XML File to JSON File**
- **Graph representation**
- **Social Network Analysis**

# 1.XML Consistency:

- **Checking the XML consistency**

```
bool consistency_checker(string opent, string closedt, stack <string>& s);
bool check_consistency(vector <string> xml_vector);
```

checking consistency process is done using these two functions.

First function is a helper function that push or pop tags from the stack and return false if it faced error during this process.

Second function use the first function for each line in xml file saved in the form of **vector <string>** to check if the total file is consistent or not.

- **Detecting Errors in consistency in XML File**

```
struct err_data
{
    string err_type;
    int err_loc;
};
```

```
bool error_detector(string opent, string closedt, stack <string>& s, string& error_type);
vector <err_data> detect_error(vector <string> xml_vector);
```

Error detecting process is done using these two functions.

First function is a helper function that check if there is an error and if there is an error it detects the error type (missing open tag or missing closed tag or not matching tags) and put it in error_type variable.

The second function use the first one for each line to detect if there is error or not and get the error type if there is an error as it iterates on the xml line by line, then it stores the error type and location in a **vector<err_data>** then return it at the end of the file

Note: error type detection process is done by the following.

If the closed tag is not matching the top of the stack then there are two option that there is missing open tag or missing closed tag.

If the current closed tag exist in the stack but not the top then the type of error is missing closed tag = top of the stack

If the current closed tag doesn't exist in the stack, then the type of error is missing open tag = the current closed tag

- **Correcting Error in consistency in XML File**

```
vector <string> error_corrector(vector <string> xml_vector, vector <err_data> error_vector);
```

The correction process is done by this function that takes the vector <err_data> result from error detection process and the original vector containing xml data the return corrected xml vector by adding the missing open or closed tags to the original vector.

Note: error correction process may not provide a hundred  percent correct xml file as it only solve errors in consistency like missing open tag or missing closed tag or not matching tags. But it will provide you with hundred percent consistent xml.

| | **Time complexity** | **Space complexity** |
|---|---|---|
| **Check consistency** | **O(n)** | **O(n)** |
| **Detect errors** | **O(n)** | **O(n)** |
| **Correct errors** | **O(n)** | **O(1)** |

# 2.XML Formatting:

```
1   #include"Helpers.h"
2
3   void format(vector <string> xml_vector);
```

the formatting process is done by the above function.

First the xml file is read then stored in format of vector <string> that is used in all the previous functions. in this process all leading or ending spaces got deleted for each line in xml file then the line is stored as a string in the xml_vector.

For format function it takes that vector and print it in a new file “formatted_file”. The algorithm used is simple as whenever there is open tag it print it the increase the number of spaces and whenever there is closed tag it decrease the number of spaces first then print the closed tag

**Example for input and output files:**

```
<users>
  <user>
                      <id>1</id>
      <name>Ahmed Ali</name>
   <posts>
          <post>
    <body>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
            </body>
                                              <topics>
                <topic>
                    economy
     </topic>
                <topic>
                    finance
                                          </topic>
            </topics>
                       </post>
     <post>
            <body>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
   </body>
```

```
<users>
   <user>
       <id>1</id>
       <name>Ahmed Ali</name>
       <posts>
           <post>
               <body>
                   Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
               </body>
               <topics>
                   <topic>
                       economy
                   </topic>
                   <topic>
                       finance
                   </topic>
               </topics>
           </post>
           <post>
               <body>
                   Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
               </body>
```

| | Time complexity | Space complexity |
|---|---|---|
| **XML Format** | **O(n)** | **O(1)** |

## 3.Minifying:

this process is done by function **Minify** that exist in format file

the function takes xml vector representing xml and return a one string without any new lines or spaces

in creating xml vector process leading and ending spaces got deleted for each line and then stored.so he Minify function role is to provide one string from this vector

<users><user><id>1</id><name>Ahmed Ali</name><posts><post><body>saher dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>economy</topic><topic>finance</topic></topics></post><post><body>saher attia dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>economy</topic></topics></post></posts><followers><follower><id>2</id></follower><follower><id>3</id></follower></followers></user><user><id>2</id><name>Yasser Ahmed</name><posts><post><body>saher attia dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>economy</topic></topics></post></posts><followers><follower><id>1</id></follower></followers></user><user><id>3</id><name>Mohamed Sherif</name><posts><post><body>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.</body><topics><topic>sports</topic></topics></post></posts><followers><follower><id>1</id></follower></followers></user></users>

This is an example for the xml file after being minified all he file become one line.

| | Time complexity | Space complexity |
|---|---|---|
| **XML Format** | **O(n)** | **O(1)** |

## XmlTree construction:

xml tree is a tree of nodes. Each node has name, value and children as the following:

```
class Node {
public:
    std::string name;
    string value;
    std::vector<Node*> children;
};
```

Then the tree class that has a root node and some functions as build xml tree and print xml tree functions as following:

```
class Tree
{
private:
    Node* Root;

    void _print_xml(Node* node);

public:
    void bulid_xml_tree(vector <string> xml_vector);
    void print_xml();
    Node* getRoot();
};
```

Process of building xml tree is done using stack as build function take xml vector as an input an then start to build tree as following:

1. First Create an empty stack to store the elements and their relationships.
2. Iterate over the xml vector and extract individual tokens
3. For each token, do the following:
    - If the token is an opening tag, put it as child for the current node(at the top of the stack) and then push it onto the stack to make it the current node
    - If the token is an attribute, store it with the current element on the top of the stack.
    - If the token is closing tag, pop the top element from the stack.

4. Repeat steps 3 until all tokens have been processed.

Converting xml files into xml tree is very useful and make it easier to deal with xml data and so can use it in other operations like converting xml to json and build a graph representing the data to make it easier to make some analysis.

Before using xml tree in Json or the graph you should build it using build function by passing xml vector to it.

| | **Time complexity** | **Space complexity** |
|---|---|---|
| **XML Format** | **O(n)** | **O(n)** |

Note: The space complexity of this process is O(n), where n is the maximum depth of the XML tree.

Time complexity is O(n) where n is number of lines in xml file or number of tokens in xml file.

## 4. Convert to Json:

Conversion to Json process is done using tree. First you construct a xml tree for the desired xml file to be converted to Json as illustrated above, then traverse the tree recursively to print it into Json.

```
#include"tree.h"

void _print_json(Node* node, bool print);
string convert_to_json(Node* node);
```

the above two function are being used for conversion process

the first function is the recursive function that take root of the tree and print the result Json formatted

the second function call the first and return the result of it into one string containing the Json data to be used in the GUI

Notes:

in order to print the Json file formatted the first function keep track of open and closed brackets for each open bracket it prints it and then increase the number of printed spaces before each line and for closed bracket it reduces the number of spaces the it prints the bracket.

In order to convert xml to Json some tags that exist in xml file doesn't exist in Json like follower tag that is done when the parent tag is a list of similar tags like follower's tag so each follower tag is not printed but print only the id that represent that follower.

| | **Time complexity** | **Space complexity** |
|---|---|---|
| **XML Format** | **O(n)** | **O(1)** |

"n" represents the number of nodes in xml tree.

## 5.compression of Xml files:

Compressing xml is done by 2 steps. First Step: removing space and convert unique names in xml file to index to reduce number of repeated unique names by using function:

```
string removeSpacingAndUnique(string path);
```

This function takes approximately o(n) which return string without spacing and unique names.

Second step: Using Huffman Coding Huffman coding is done with the help of the following steps.

- Calculate the frequency of each character in the string.
- Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.
- Make each unique character as a leaf node.
- Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.
- Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above)
- Insert node z into the tree.
- Repeat steps 3 to 5 for all the characters.
- For each non-leaf node, assign 0 to the left edge and 1 to the right edge.

```
HuffmanNode* getNode(char ch, int freq, HuffmanNode* left, HuffmanNode* right);
void compress(priority_queue<HuffmanNode*, vector<HuffmanNode*>, comp>& pq);
HuffmanNode* buildHuffmanTree(unordered_map<char, int> freq);
void encode(HuffmanNode* root, string str, unordered_map<char, string>& huffmanCode);
string compressStringByHuffman(string text);
```

Then, take every 8char (zero’s & one’s) to one ascii char. & add Huffman table to file.

- Size of original file is 2,848 bytes.
- 

- Size after compression is 1,055 bytes.
- 


→ Time complexity of the Huffman algorithm is O(nlogn). Using a heap to store the weight of each tree. each iteration requires O(logn) time to determine the cheapest weight and insert the new weight. There are O(n) iteration, one for each item.

→ Space complexity is O(k) for tree and O(n) for text.

| | **Time complexity** | **Space complexity** |
|---|---|---|
| **removeSpaceAndUnique** | **O(n)** | **O(n)** |
| **CompressStringByHuffman** | **O(nlog(n))** | **O(n)** |

## 6. Decompressing XML File:

Decompressing xml is done by 3 steps.

First Step: extract Huffman table from text to build Huffman tree to use it in decoding.

Second Step: Convert String from compressed file to zero's and one's char then decompressing them by Huffman.

Third Step: store each line in xml file in vector of string then send it to format function.

```
struct HuffmanNode;
string convertStringToBinary(string compressedText);
void decode(string s, HuffmanNode* root, int& index,string & str);
vector<string> decompress(string compressedText);
```

Time complexity to build Huffman algorithm is O(klogk) which k is number of repeated chars in tree. Time complexity of decompress is O(n) which n is a number of characters in compressed file plus build Huffman tree.

Space complexity is O(n) which n is number of characters in original text.

## 7. Graph Representation:

The graph in this editor is represented by a vector of users each user has a list of followers so the graph is represented an adjacency list as all social networks are spares.

This is user class:

```
class User
{
public:

    string name;
    string id;
    vector<string> followers;
    unordered_map<string, vector<string>> posts;
};
```

Each user has name and id and map of posts and topics and list of followers.

The creation of vector of user process is done by the following function:

```
void create_useres_vector(Node* node);
```

This function takes a tree that represents Xml data and then traverse this tree extracting all desired information.

This function calls the following recursive function:

```
void buid_graph(Node* node, stack <Node*>& nstack, User* user, bool idFlag, stack <string>& pstack);
```

This function use two stacks to keep track of xml tree nodes and posts in the xml file.

During creating of vector of user process the function create a map of posts to support the functionality of **post search** as it make it easier to access the posts of a certain topic.

```
unordered_map<string, vector<string>> posts;
```

And also stores the number of users and their location in users' vector to support other functions like mutual followers and followers' suggestion.

| | **Time complexity** | **Space complexity** |
|---|---|---|
| **Build graph process** | **O(n)** | **O(n)** |

As "n " represents the number of nodes exist in the Xml tree for time complexity

The space complexity of this process is O(n), where n is the maximum depth of the XML tree. As we use stack to keep track of the current node in the tree.

## 8. social network analysis:

social network analysis is done through the following functions:

```
User most_influencer_user();

User most_active_user();

vector<User> mutual_followers(string id1, string id2);

vector<User> follow_suggestion(string id);

vector <string> post_search(string topic);
```

- The most influencer user is done by iterating over users vector checking the size of the followers list of the user the user that has the biggest size is the user that has the largest number of followers and so the most influence user.

- The most active user is done by checking the count of user existence in other users' followers list the user with biggest count is the most active user.

- The mutual followers is done by comparing the list of followers for both users and returning the mutual followers.

- Follow suggestion is done by get the followers of the user follower if and store it if the current follower doesn't exist between them then remove duplicates between those followers.

- The post search is done using the posts map created during building the graph. You enter the topic and it returns vector of posts related to this topic.

| | Time complexity | space complexity |
|---|---|---|
| **most influencer** | **O(n)** | **O(1)** |
| **most active** | **O(n*m)** | **O(n)** |
| **mutual followers** | **O(m^2)** | **O(n)** |
| **Follow suggestion** | **O(n*m)** | **O(n)** |
| **post search** | **O(1)** | **O(1)** |

Note: n represents the number of users and m represent number of followers of a user

---

GitHub Link:

https://github.com/HazemHamdyy/data-structure-project

Drive Link:

https://drive.google.com/drive/u/0/folders/17H2T-CPaFEAyjtL02wACYzjTPnNA8qh2