



Faculty of Engineering, Ain Shams University

CSE331s - Data Structures and Algorithms

XML Viewer Project

Name	ID
Ahmed Tarek Sayed Abbas	1809660
Abdelrahman Elsayed Mohamed	1803842
Ahmed Tyseer Farouk Mohamed	1801347
Abdelrahman Abdelaziz Ramadan	1809918
Elzoheery Ahmed Elzoheery Aly	1807813

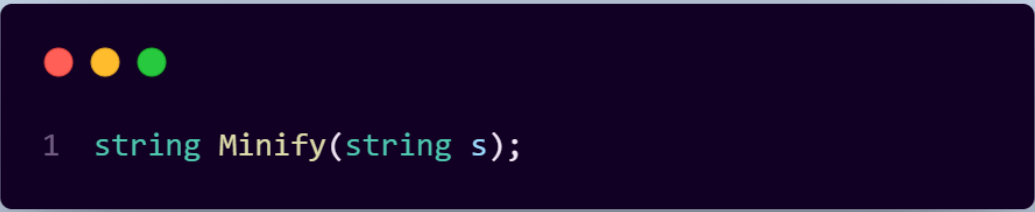
1. Contents

1. Contents.....	2
2. Minify.....	3
3. XML Consistency	5
4. XML Text Format.....	9
5. XML to JSON.....	11
6. Compression	14
7. Decompression	18

2. Minify

- **Background**

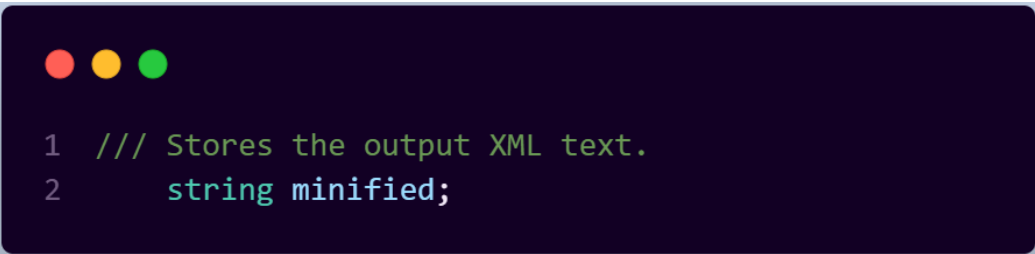
This function Deletes unnecessary spaces and newlines from an XML file. It accepts a string containing the XML text and returns a string containing the XML text without unnecessary spaces and newlines (i.e., the same XML as the input but written in a single line.)



```
1 string Minify(string s);
```

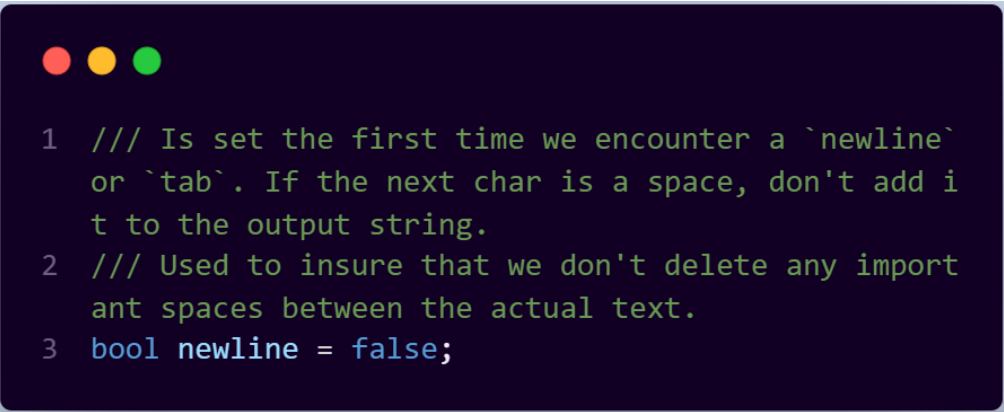
- **Implementation**

Defining a variable to store the output XML text.



```
1 /// Stores the output XML text.  
2     string minified;
```

Because we are dealing with words and sentences, we don't want to delete any space between words or sentences to keep everything as it was meant to be sent. That's why we need to keep track of where we are now. If we are inside a sentence, we will not delete any spaces, but we will delete newlines and tabs, and if we are outside a sentence (and outside tags, obviously,) we can safely remove any spaces (i.e., to strip the sentence). This is the job of the variable *newline*, to tell us if we are inside a sentence. A value of **true** means that we are outside sentences.



```
1 /// Is set the first time we encounter a `newline`  
   or `tab`. If the next char is a space, don't add i  
   t to the output string.  
2 /// Used to insure that we don't delete any import  
   ant spaces between the actual text.  
3 bool newline = false;
```

Now, we are going to loop over the XML text. All **newlines** and **tabs** will be deleted *regardless of where they are*. Only **spaces** will be deleted *if they are not inside a sentence*.

```
1  for(int i=0; i<s.length(); i++) {
2      /// Deleting spaces and tabs
3      if(s[i] == '\n' || s[i] == '\t') {
4          newline=true;
5          continue;
6      }
7
8      /// Deleting unnecessary spaces.
9      if(newline) {
10         if(s[i] == ' ') {continue;}
11         else {newline=false;}
12     }
13
14     /// If we encountered a char other than new
    lines and tabs, add it to the output text.
15     minified.append(1, s.at(i));
16 }
```

Finally, we conclude our work by returning the *minified* string.

```
1      /// return the minified XML.
2      return minified;
3  }
```

- **Complexity**

- Time Complexity $O(N)$, because we need to loop over the whole string.
- Space Complexity $O(N)$, because we need to store the string to minify it.

3. XML Consistency

1) XML Error Detection Function

```
void xmlErrordetection(LinkedList& l, queue<string>& in_tag, queue<int>& in_index, queue<int>& del_index )
```

- **Implementation:** this function detects some errors related to xml consistency such as any missed closed tag and any wrong named closed tag and fills the queues with result of detection.
- **Time and space complexity:**
 - Time: $O(N)$
 - Space: $O(N)$ (using stack)

2) XML Error Correction Function

```
void xmlErrorcorrection(LinkedList& l, queue<string>& in_tag, queue<int>& in_index, queue<int>& del_index)
```

- **Implementation:** this function corrects the errors in linked list based on queues filled by the error detection function.
- **Time and space complexity:**
 - Time: $O(N)$
 - Space: $O(1)$

3) XML Correct Function

```
bool xmlcorrect(LinkedList& l) {  
    queue<int> in_index;  
    queue<int> del_index;  
    queue<string> in_tag;  
  
    xmlErrordetection(l, in_tag, in_index, del_index);  
    return((in_index.empty()) && (in_tag.empty()) && (del_index.empty()));  
}
```

- **Implementation:** this function returns true if the XML doesn't include any of the mentioned errors.
- **Time and space complexity:**
 - Time: $O(N)$
 - Space: $O(N)$

4) XML Error Position

```
void xml_errorPositions(LinkedList& lP, queue<string>& in_tagP, queue<int>& in_indexP, queue<int>& del_indexP) {
    if (xmlcorrect(lP)) return;
    xmlErrordetection(lP, in_tagP, in_indexP, del_indexP);
}
```

- **Implementation:** this function is called with linked list and queue parameters and its functionality to filled them with the positions of deleted and inserted closed tags and the closed tags to be inserted.
- **Time and space complexity:**
 - Time: $O(N)$
 - Space: $O(N)$

5) XML Automatically Correct Function

```
void xml_automatically_correct(LinkedList& l) {
    queue<int> in_index;
    queue<int> del_index;
    queue<string> in_tag;

    xmlErrordetection(l, in_tag, in_index, del_index);
    xmlErrorcorrection(l, in_tag, in_index, del_index);
}
```

- **Implementation:** this function is called with linked list (which includes the parsed string) as parameter and returned it after correcting.

- **Time and space complexity:**

- Time: $O(N)$
- Space: $O(N)$

6) XML Conversion Functions

- **Background**

There are two conversion functions used in the checking of the xml one to convert the xml to linked list and the other to convert the linked list back to xml string after correction.

- **Parameters and Return**

- **xmlToLinked**

- The parameters: XML string and a pointer to a linked list
- The return type: void

```
void xmlToLinked(string xml, LinkedList* l);
```

- **linkedToXml**

- The parameters: the head of a linked list
- The return type: XML string

```
string linkedToXml(Node* head);
```

- **Implementation**

- **xmlToLinked**: convert the xml string to linked list have nodes contain open tag, closed tag, and text included inside two tags with the same order in the xml string.
- **linkedToXml**: append each node data in xml string and return it.

- **Time and Space Complexity**

- **xmlToLinked**: $O(N)$, N is the number of characters in the xml output string

- **linkedToXml:**
 - Time: $O(M)$, M is number of nodes in the linked list (tags and texts)
 - Space: $O(N)$, N is the number of characters in the xml output string
- **Data Structures Used**
 - **String**: which uses vector contains a **dynamic array** object.
 - **Linked list**
 - **Stack**: implemented with linked list (list) and has additional function
 - **Search(string closeTag)**: this function search in stack for the open tag that match the closed input tag. It has time complexity of $O(M)$, M is elements in the stack and space of $O(1)$.

4. XML Text Format

- **Background**

The formatting function make the right format for an xml file by keeping the right indentation for each tag level. It also depends on the GUI window to set the indentation and the new lines.

- **Parameters and Return**

- The function takes as input:
 - XML string
 - The width of the characters of the GUI window
- The function returns:
 - The formatted XML string




```
string format(string xml, unsigned int windowWidth);
```

- **Implementation details**

- First the function calls the minify function to remove any wrong formatting.
- Takes the XML version tag if any.
- Iterate over the minified xml and copy the data into the output string.
- Update the tag with the current one and update the flags according to the coming tag or paragraph



```
// Evaluate the flags
openTag = (input[i + 1] == '<' && input[i + 2] != '/');
paragraph = (input[i] == '>' && input[i + 1] != '<');
closeTag = (input[i + 1] == '<' && input[i + 2] == '/');
```



```
// Get the tag
if (input[i] == '<')
{
    tag.erase();
    for (int j = i + 1; input[j] != '>'; j++)
        tag.append(1, input[j]);
}
```

- It updates generally the level of indentation in case of open or close tags.
- For long paragraph it split it into lines match the width of the GUI window.

```
<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>
```

- If a tag has not any children and can fit the width of the GUI window, it will put this tag with its open, close and text in one line with the last indentation

```
<!--output-->
<author>Gambardella, Matthew</author>
```

- An image of output formatted XML on the console window

```
<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. 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>
    </posts>
  </user>
</users>
```

- Complexity

- Time Complexity $O(N)$, n is the number of characters in the XML input string.
- Space Complexity $O(N)$, n is the number of characters in the XML input string.

5. XML to JSON

Aim: converting XML format to JSON format

Input: XML format as a string

Output: JSON format as a string

- **Background**

JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays (or other serializable values). It is a common data format with a diverse range of functionality in data interchange including communication of web applications with servers. JSON is a language-independent data format.

- **Main Idea**

First, we loop on the XML string and convert it to a tree data structure. Each tag is represented as a node in this tree which has tag name, attributes, and content. Then we use recursion to print tags and their value as it is in JSON format.

Main steps in the XML to JSON Conversion:

- 1) Build an XML Tree from input string
- 2) Traverse the XML Tree and print it in JSON format.

- **Steps to Build the XML Tree**

- 1) We loop until we find an opening tag, attach it to the root node.
- 2) Search about nested tags if it has, we update current root to this node that we are having now.
- 3) We attach content to this node.
- 4) Repeat doing this until we reach the end of the string.

- **Implementation Details**

There are many auxiliary functions, but I will focus on the main ones:

1) Node Class

```
/*
 * class Node is basic element of a Tree
 * used to build a tree of XML Nodes
 */
class Node
{
public:
    // name of the tag
    string name = "";
    // its content
    string value = "";
    // its attributes
    string attributes = "";
    // helper for find and print methods
    bool found = false;
    // parent of that tag
    Node* parent = nullptr;
    // children of a tag
    vector<Node*> children;
    // no arg constructor
    Node( );
    // one arg constructor
    Node(string name);
    // destructor
    ~Node( );
};
```

2) Tree

The tree data structure (a data organization, management, and storage format that enables efficient access and modification) is a widely used abstract data type that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

```

#ifdef TREE_H
#define TREE_H

#include "Node.hpp"

/*
 * used to build a tree of XML nodes
 */
class Tree
{
private:
    // root of the XML tree
    Node* root;

    /*
     * private method that used by Print_json method
     * print Tree as a json format using cout object
     */
    /* @param r: pointer to the root element
     * @param flag: helper to know where to stop
     */
    void _print_json(Node* r, bool flag);

public:
    // no arg constructor
    Tree();

    /*
     * parse xml string and convert it to a
     * XML_Tree of nodes
     */
    /* @param xml_file: the xml string that will be parsed
     */
    void parse_and_build_xml_tree(const std::string& xml);

    /*
     * the public version of _Print_json
     * it prints Tree as a JSON format using cout object
     */
    void print_json();

    /*
     * public interface with other classes
     * it call method that converts XML to JSON
     */
    /* @param xml_file: the xml string that will be converted to json
     * @output json_file: a string that has the converted text as json format
     */
    string convert_to_json();

    // destructor for the tree which call
    // destructor for Node class
    ~Tree();
};

#endif // TREE_H

```

- **Complexity**

- 1) **parse_and_build_xml_tree method**

- Time Complexity $O(N)$, because we need to loop over the whole string.
- Space Complexity $O(M)$, where M is the number of tags as we need to store the string in Tree

- 2) **convert_to_json method**

- Time Complexity $O(M)$, where M is the number of tags (Nodes) because we recursively print each node's name and value
- Space Complexity $O(N)$, because we need to store the string in JSON format.

6. Compression

- **Background**

- **Compression Technique:** Huffman Encoding
- **Aim:** Compressing the data in the XML/JSON file, simply we reduce file size
- **Input:** input File that we want to compress it
- **Output:** Compressed File that its size is smaller than the original file
- **Main Idea:** As we know that every character has 8 bit or 1 byte size, The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters in the file. In a way, this makes:
 - **The most frequent character gets the smallest code**
 - **The least frequent character gets the largest code**

so, if we replace every input character code by its new code, we can reduce the total size of file.

Note: the codes assigned to input characters are Prefix Codes, means the codes are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. so, no errors happen during decoding.

- **Main steps in Huffman Compression**

- 1) Build a Huffman Tree from input characters.
 - 2) Traverse the Huffman Tree and assign codes to characters.
 - 3) Encode every character in input file using new codes to compressed file
- **Steps to build Huffman Tree**
 - 1) Get frequency of occurrences of every input character
 - 2) Create a leaf node for each unique character and build a min heap (used as priority queue) of all leaf nodes, The value of frequency field is used to compare two nodes in min heap
 - 3) Extract two nodes with the minimum frequency from the min heap

- 4) Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
 - 5) Repeat steps 3 and 4 until the heap contains only one node. The remaining node is the root node, and the tree is complete.
- **Steps to assign codes to characters from Huffman Tree**
 - 1) Traverse the Huffman tree starting from the root.
 - 2) Maintain an auxiliary string.
 - 3) While moving to the left child, string = string + 0
 - 4) While moving to the right child, string = string + 1
 - 5) When encounter with leaf node (input character node), set the code filed with the string
 - 6) We will do that for every leaf node

- **Steps to encode every character to compressed file**

Compressed file will consist of 2 parts (header, actual data)

Header will contain:

- i. Number of unique input character
- ii. Every input character + its code

Actual data: compressed data (replacement of every character in file to its code)

Note: we add Header to compressed file that will be used to rebuild Huffman tree so we can do Decompression

- **Implementation details and Complexity**

```
//user interface for all steps for Compressing operation
void huffman_compress::compress_file()
{
    this->build_priority_queue();
    this->build_huffman_tree();
    this->calculate_huffman_codes();
    this->compression_saving_to_compressed_file();
}
```

1) build_priority_queue()

*/*gets frequency of each character in file and set the frequency of character node to it then add the node of character that exists in the file to priority queue if any character of 128 character in the Ascii Table does not exists in the file then its character nodes will not add to priority queue*/*

Time Complexity: $O(N)$, as n is the number of characters in input file, as I visit every character in the file till end of the input file, we do not care about $O(\log(N))$ of inserting nodes in priority queue as it is less than $O(N)$

Space Complexity: $O(N)$, as N is number of nodes that added to priority queue

2) build_huffman_tree()

*/*build the Huffman tree, making characters nodes as leaves*

- the most frequency character will be leaf node near to root (smallest depth) (small Huffman code)
- the least frequency character will be leaf node far from root (largest depth) (big Huffman code)*/

Time Complexity: $O(N \log(N))$, as iteration or repetition (as described above in section of (Steps to build Huffman Tree)) we be dependent on the size of priority queue $O(N)$ and time for extract minimum nodes from priority queue(min heap) will be $O(\log(N))$

Space Complexity: $O(N)$, as we copy the main priority queue in temporary priority queue to execute steps described above in section of (Steps to build Huffman Tree)

3) calculate_huffman_codes()

/ this the function only calls the auxiliary function called traverse () in this form*

*traverse (root, ""). So, I will focus on it */*

- void traverse(node_ptr, string)

/ We traverse the Huffman tree to get Huffman code for a character save the Huffman code in code data member of node of every character as described above in section of (Steps to assign codes to characters from Huffman Tree)*/*

Time Complexity: $O(N)$, as we visit every node in tree until we reach a leaf node and set its code to it.

Space Complexity: $O(1)$, as no extra space needed.

4) `compression_saving_to_compressed_file()`

*/*creating the Compressed File that will consist of 2 main parts first we create header to use it in decompression then encode actual data*/*

Time Complexity: $O(N^2)$, as we visit every character in input file until end of file to encode every character and save it in compressed file and while visiting, we used auxiliary function called **binary_to_decimal()** which has time complexity of $O(N)$ so we need $O(N^2)$ code to it.

Space Complexity: $O(N)$, as we copy the main priority queue in temporary priority queue, and we used string variable that its size depend on the frequency and number of characters in the input file.

7. Decompression

- **Background**

- **Decompression Technique:** Huffman Decoding
- **Aim:** Decompressing the data in .huf file, simply we restore the original data
- **Input:** Compressed file which generated from compression operation
- **Output:** Original file (its size is bigger than the compressed file)
- **Main Idea:** To decompress the compressed data, we need to rebuild the Huffman tree. So, we will need to know every character and its Huffman code. In compression operation I store this required data in header section of compressed file so I will read it first to rebuild Huffman tree We go through the binary encoded data. To find character corresponding to current bits, we use following simple steps:
 - 1) Start from root and do following until a leaf is found.
 - 2) If current bit is 0, we move to left node of the tree.
 - 3) If the bit is 1, we move to right node of the tree.
 - 4) If during traversal, we encounter a leaf node, we print character of that leaf node and then again continue the iteration of the encoded data starting from step 1.

No errors happen during the decompression operation as the codes are prefix codes.

- **Implementation details and Complexity**



```
//user interface for all steps for decomprssing operation
void huffman_decompress::decompress_file()
{
    this->rebuild_huffman_tree();
    this->decompression_saving_to_decompressed_file();
}
```

- 1) **rebuild_huffman_tree()**

/*reading the compressed file and read the header part to access character + its code pass character and its code to auxiliary function called **build_tree_paths()** we do that specific number of times equal the number of unique characters in the original file (simply the size of priority queue) with these we can rebuild the Huffman tree*/

- **build_tree_paths(string& huffman_code, char character)**

*/*As traversing we loop on every character in the Huffman code:*

- if it is '0', go to left child node.
- If there is not left child node, we create it.
- If it is '1', go to right child node.
- If there is not right child node, we create it

** We continue looping until reaching the end of string, the latest appended node is the character node (leaf node)* /*

Time Complexity: $O(N)$, as it depends on the length of code string.

Space Complexity: $O(K \times M)$, as K is the number of nodes created to build character node path, M is the size of Node.

So, the complexity of the rebuild_huffman_tree() function:

Time Complexity: $O(N^2)$, as it depends on the number of unique characters in the original file(first byte in the compressed file) $O(N)$ and after getting character and its code we use build_tree_paths() function $O(N)$.

Space Complexity: $O(K \times M)$, as K is the total number of nodes created to build Huffman tree, M is the size of Node.

2) decompression_saving_to_decompressed_file()

*/*reading the compressed file from the part of actual data (after header) decoding this encoded data to original data and save it in the decoded file as described above in section of (Main Idea)* /*

Time Complexity: $O(N)$, as N is number of characters to decompress.

Space Complexity: $O(N)$, as N is number of characters to decompress.