BBM 203 Assignment 4

REPORT

Name: Anıl

Surname: AKKAYA

Number: 21945781

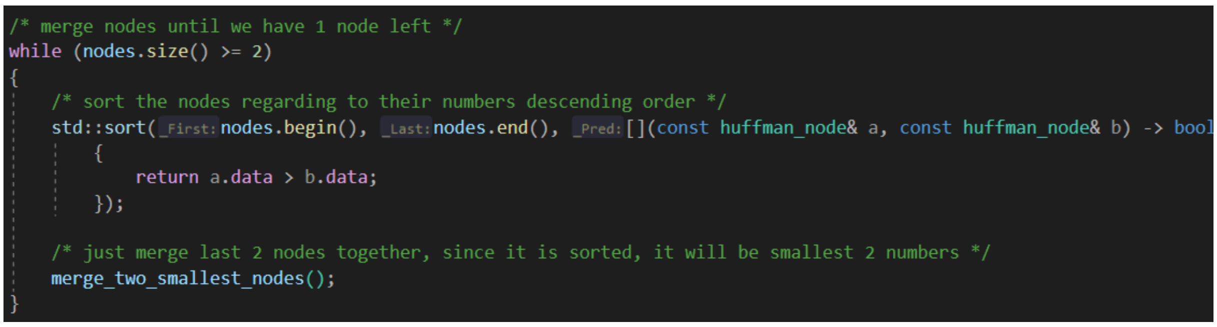
Date: 30 December 2020

Program Hierarchy

- The class ‘huffman_tree’ is responsible for all the encoding and decoding operations.
- Inside this class, ‘huffman_node’ structure is defined whose task is to represent a single node and its relationship between its left and right child. For the sake of proper memory management, copy and swap idiom was used, so there is no unnecessary copy operations and no memory leaks.
- The ‘input_helper’ namespace contains a few helper functions for string operations.
- My code is highly commented, so reading source is easy and understandable. I have tried to write code as small as possible.

Encoding Algorithm

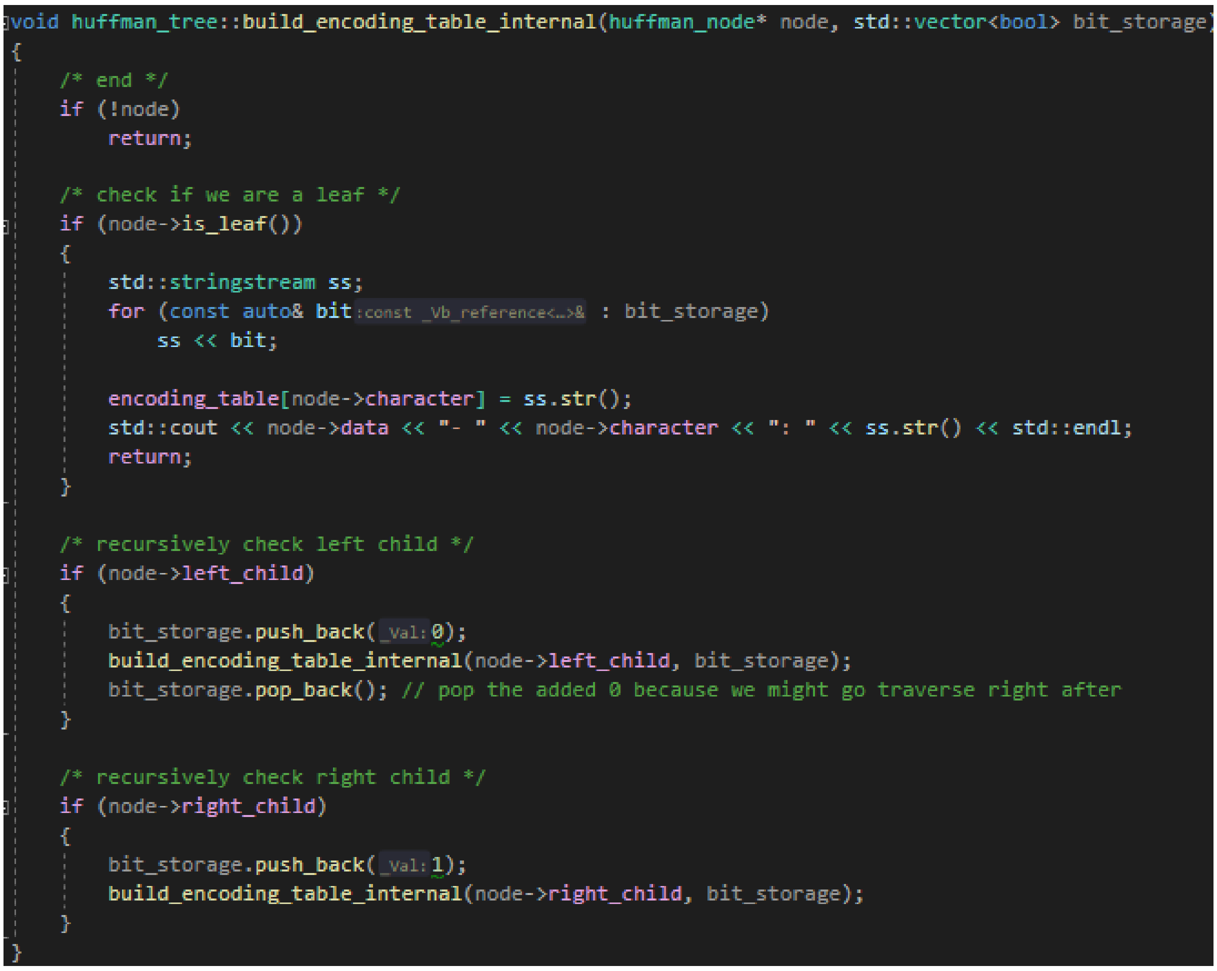
- ‘huffman_tree’ class constructor requires the occurrence data of the letters inside the input string and creates the encoding table to be used in encoding.
- To do it, function first stores every leaf node inside a vector, and until there is only one node left, it merges the two smallest nodes. Main algorithm:



```
/* merge nodes until we have 1 node left */
while (nodes.size() >= 2)
{
    /* sort the nodes regarding to their numbers descending order */
    std::sort(_First: nodes.begin(), _Last: nodes.end(), _Pred: [](const huffman_node& a, const huffman_node& b) -> bool
        {
            return a.data > b.data;
        });

    /* just merge last 2 nodes together, since it is sorted, it will be smallest 2 numbers */
    merge_two_smallest_nodes();
}
```

- After the tree is created, a function named ‘build_encoding_table’ is called, which calls ‘build_encoding_table_internal’. That function uses <u>recursion</u> to traverse the binary tree.
- Most important part here is that it fills a **vector<bool>** as it traverses, that represents the road to the leaf nodes, which will ultimately be the encoding bit codes for the reached character at the leaf.
- After the encoding table for each character is filled, it is trivial to encode the message, ‘encode_message’ function is responsible for it.

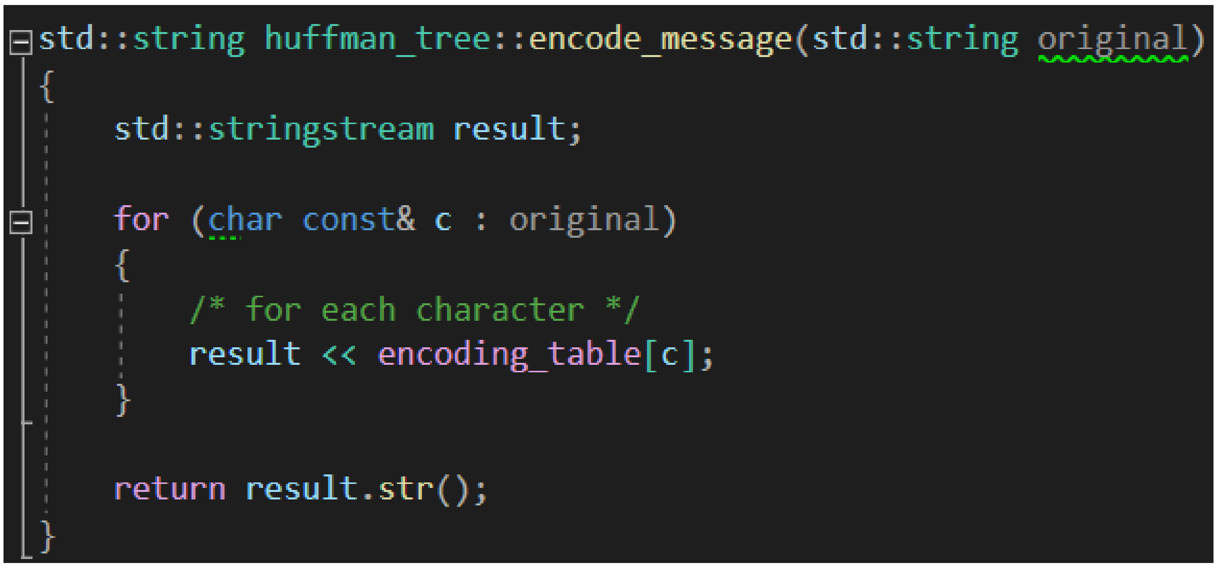
build_encoding_table_internal:



```cpp
void huffman_tree::build_encoding_table_internal(huffman_node* node, std::vector<bool> bit_storage)
{
    /* end */
    if (!node)
        return;

    /* check if we are a leaf */
    if (node->is_leaf())
    {
        std::stringstream ss;
        for (const auto& bit :const _Vb_reference<…>& : bit_storage)
            ss << bit;

        encoding_table[node->character] = ss.str();
        std::cout << node->data << "- " << node->character << ": " << ss.str() << std::endl;
        return;
    }

    /* recursively check left child */
    if (node->left_child)
    {
        bit_storage.push_back(_Val: 0);
        build_encoding_table_internal(node->left_child, bit_storage);
        bit_storage.pop_back(); // pop the added 0 because we might go traverse right after
    }

    /* recursively check right child */
    if (node->right_child)
    {
        bit_storage.push_back(_Val: 1);
        build_encoding_table_internal(node->right_child, bit_storage);
    }
}
```

(std::cout command at this function was later removed, because it was there for debugging purpose)

encode_message:

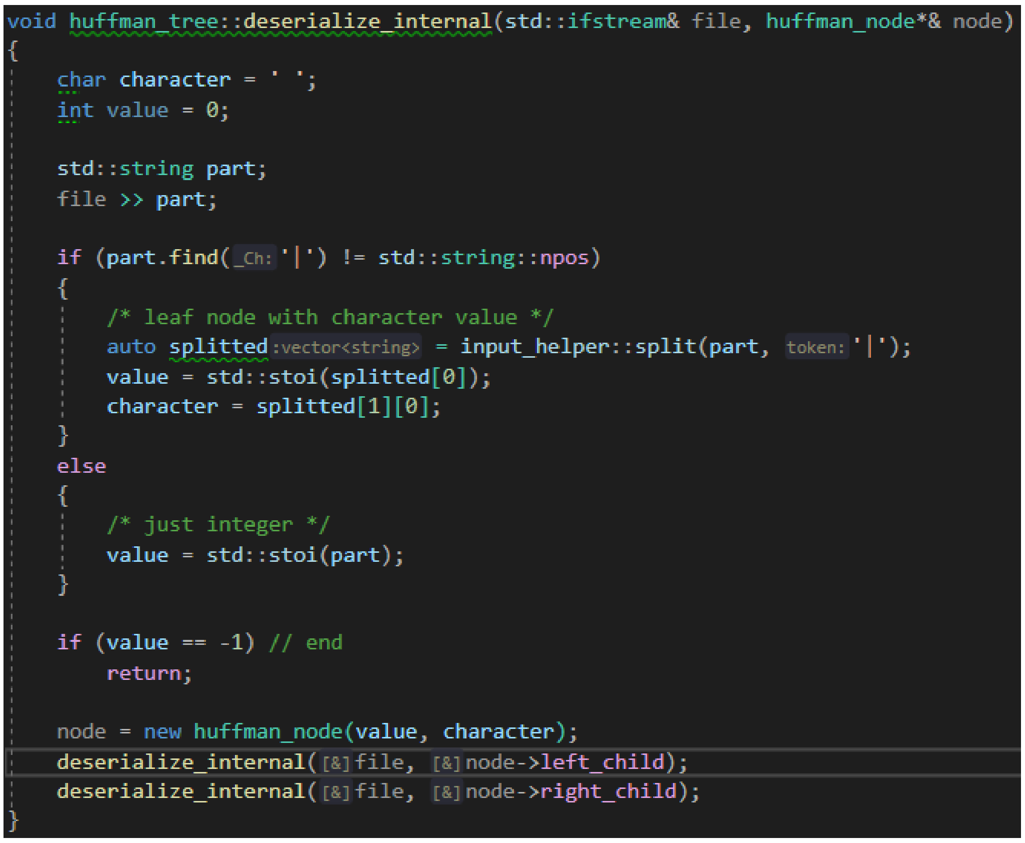


```cpp
std::string huffman_tree::encode_message(std::string original)
{
    std::stringstream result;

    for (char const& c : original)
    {
        /* for each character */
        result << encoding_table[c];
    }

    return result.str();
}
```

Decoding Algorithm

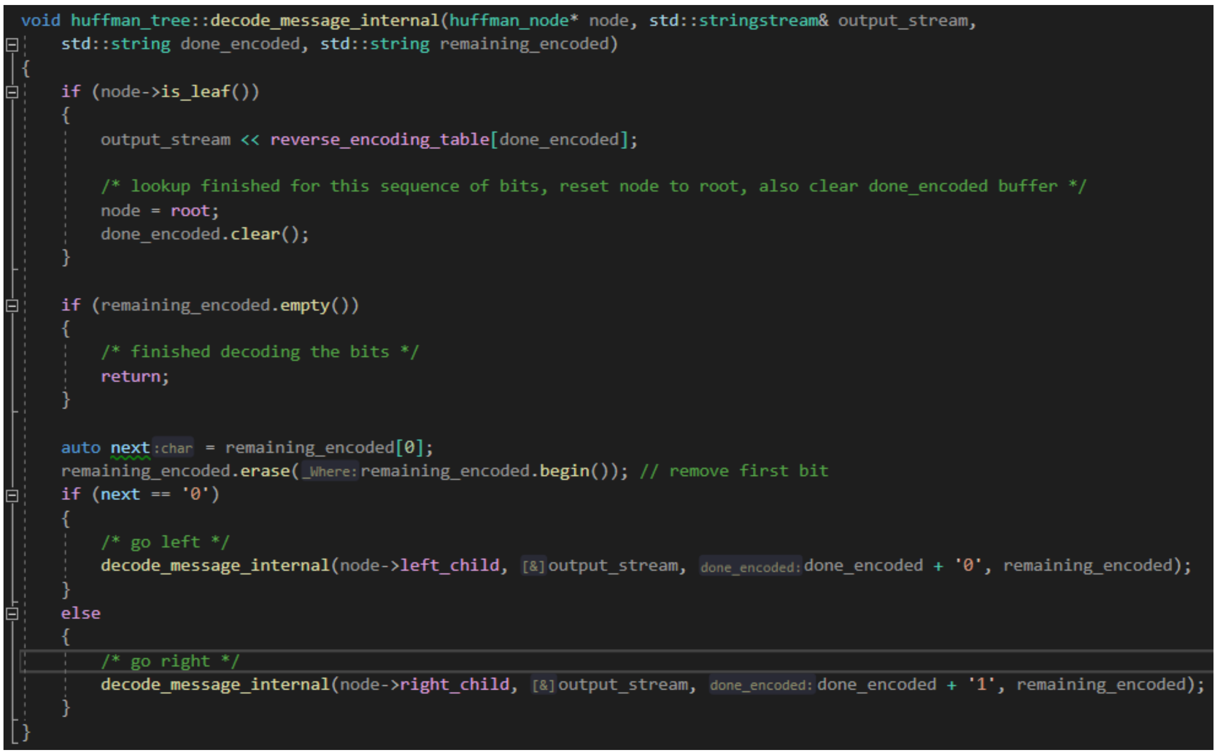
- For decoding to work, we need both the encoded data and the tree that was used to encode that message.
- First, we need to read the tree file that was created when the encoding was made. There is a special constructor inside ‘huffman_tree’ class that takes **std::ifstream&** as input. This function is responsible for deserializing the tree, with the help of ‘deserialize_internal’ function. Deserializing is pretty simple, we just create the nodes in the same order we serialized them.
- After deserialization, a reverse encoding table is created, which maps the bits of encoded letters to their ASCII values, Example: ‘001’ -> ‘A’, ‘11’ -> ‘C’…
- Decoding algorithm is mainly done by the function ‘decode_message_internal’, which again, uses recursion to decode the encoded input.
- Design of that function is simple and effective, it uses 2 strings as their state buffers, and pops and pushes the bits from one to another, until there are no data to decode, algorithm traverses the list by using encoded bits(0=go left, 1=go right) until a leaf is reached, the decoded character is appended into output stream.

deserialize_internal:



```
void huffman_tree::deserialize_internal(std::ifstream& file, huffman_node*& node)
{
    char character = ' ';
    int value = 0;

    std::string part;
    file >> part;

    if (part.find(_Ch: '|') != std::string::npos)
    {
        /* leaf node with character value */
        auto splitted :vector<string> = input_helper::split(part, token: '|');
        value = std::stoi(splitted[0]);
        character = splitted[1][0];
    }
    else
    {
        /* just integer */
        value = std::stoi(part);
    }

    if (value == -1) // end
        return;

    node = new huffman_node(value, character);
    deserialize_internal([&] file, [&] node->left_child);
    deserialize_internal([&] file, [&] node->right_child);
}
```

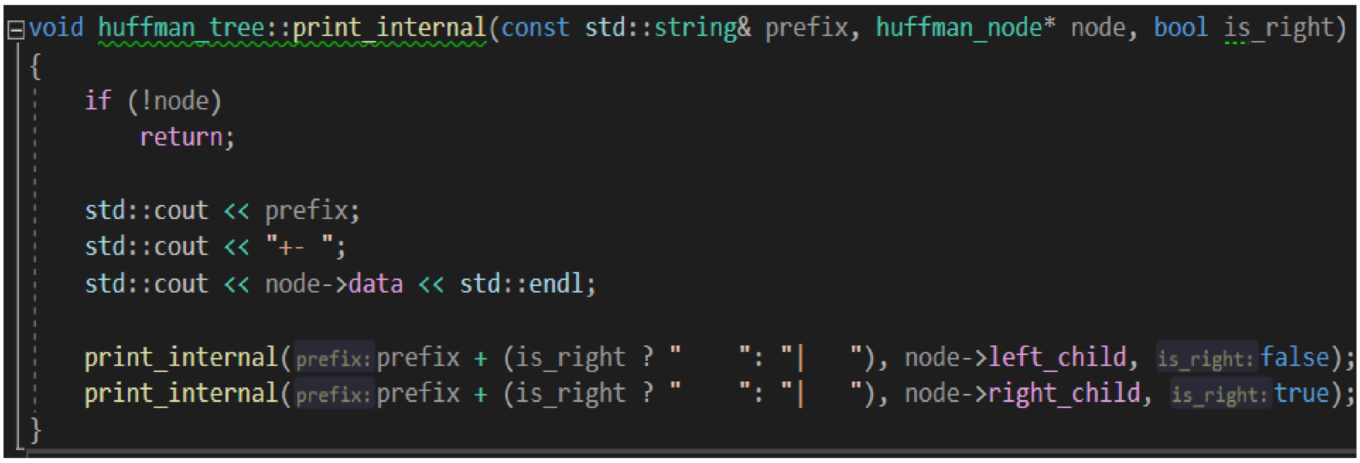
decode_message_internal:



```
void huffman_tree::decode_message_internal(huffman_node* node, std::stringstream& output_stream,
    std::string done_encoded, std::string remaining_encoded)
{
    if (node->is_leaf())
    {
        output_stream << reverse_encoding_table[done_encoded];

        /* lookup finished for this sequence of bits, reset node to root, also clear done_encoded buffer */
        node = root;
        done_encoded.clear();
    }

    if (remaining_encoded.empty())
    {
        /* finished decoding the bits */
        return;
    }

    auto next :char = remaining_encoded[0];
    remaining_encoded.erase(_Where: remaining_encoded.begin()); // remove first bit
    if (next == '0')
    {
        /* go left */
        decode_message_internal(node->left_child, [&] output_stream, done_encoded: done_encoded + '0', remaining_encoded);
    }
    else
    {
        /* go right */
        decode_message_internal(node->right_child, [&] output_stream, done_encoded: done_encoded + '1', remaining_encoded);
    }
}
```

List Tree Command

- This command uses recursion to print the binary tree in a visual way, so we can understand the data structue better.
- It has 2 important arguments, string prefix and boolean is_right, by preserving prefix along recursive calls we set the horizontal spacing between line start and the node itself. By using is_right, we can deduce if the node we are printing is the left or the right node, and adjust the output accordingly.
- Here is the code snippet of the function 'print_internal', which is responsible for printing the tree:



```cpp
void huffman_tree::print_internal(const std::string& prefix, huffman_node* node, bool is_right)
{
    if (!node)
        return;

    std::cout << prefix;
    std::cout << "+- ";
    std::cout << node->data << std::endl;

    print_internal(prefix: prefix + (is_right ? "    ": "|   "), node->left_child, is_right: false);
    print_internal(prefix: prefix + (is_right ? "    ": "|   "), node->right_child, is_right: true);
}
```

How to Run

- After compiling the program with “make” command, the final executable name is main, here is the documentation of supported commands:

```
./main -i input_file.txt -encode
```

➔ This command will print the encoded text on console output.

```
./main -i input_file.txt -encode -s character
```

➔ This command will print the encoded text on console output also print the huffman code used used for encoding the character given after -s.

```
./main -i input_file.txt -decode
```

➔ This command will print the decoded text on console output. Make sure to run this command after encoding, so generated tree file can be used.

```
./main -l
```

➔ This command will print the binary tree on console output. Make sure to run this command after encoding, so generated tree file can be used.