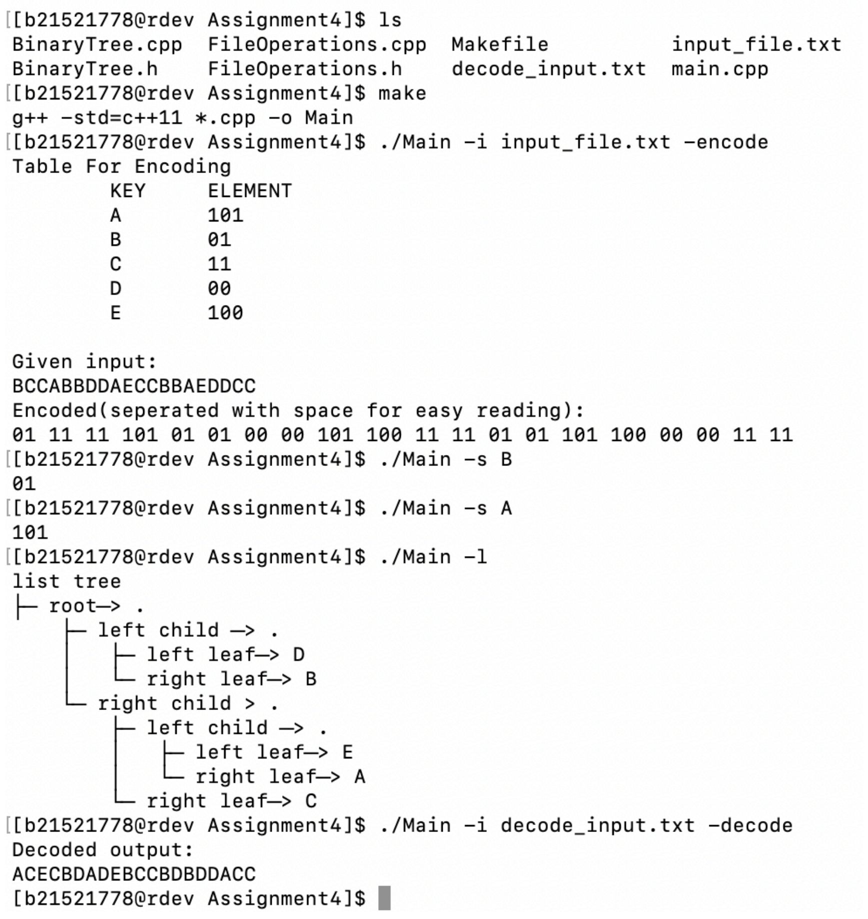
Hacettepe University
Computer Engineering Department

Name and Surname	: Berke Keser
Identity Number	: Y21521778
Course	: BBM203 Software Laboratory I
Experiment	: Assignment 4
Subject	: Huffman Algorithm / Binary Tree
Due Date	: 31.01.2020
Advisors	: R.A. Yunus Can Bilge
e-mail	: b21521778@cs.hacettepe.edu.tr
Main Program	: Main.cpp

1 – Program Compilation and Run Details

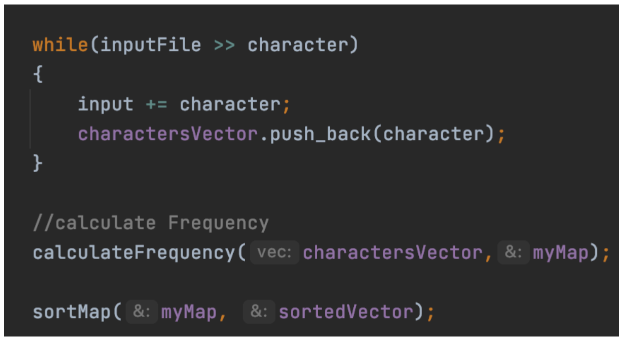
I compiled my program using make file on our department server. After the compilation, program waits for commands line arguments for running. The program takes command-line arguments as mentioned in the assignment explanation pdf. In the given example below, you could see the running demonstration.



```
[[b21521778@rdev Assignment4]$ ls
 BinaryTree.cpp  FileOperations.cpp  Makefile          input_file.txt
 BinaryTree.h    FileOperations.h    decode_input.txt  main.cpp
[[b21521778@rdev Assignment4]$ make
 g++ -std=c++11 *.cpp -o Main
[[b21521778@rdev Assignment4]$ ./Main -i input_file.txt -encode
 Table For Encoding
         KEY     ELEMENT
         A       101
         B       01
         C       11
         D       00
         E       100

 Given input:
 BCCABBDDAECCBBAEDDCC
 Encoded(seperated with space for easy reading):
 01 11 11 101 01 01 00 00 101 100 11 11 01 01 101 100 00 00 11 11
[[b21521778@rdev Assignment4]$ ./Main -s B
 01
[[b21521778@rdev Assignment4]$ ./Main -s A
 101
[[b21521778@rdev Assignment4]$ ./Main -l
 list tree
 ├─ root—> .
     ├─ left child —> .
     │   ├─ left leaf—> D
     │   └─ right leaf—> B
     └─ right child > .
         ├─ left child —> .
         │   ├─ left leaf—> E
         │   └─ right leaf—> A
         └─ right leaf—> C
[[b21521778@rdev Assignment4]$ ./Main -i decode_input.txt -decode
 Decoded output:
 ACECBDADEBCCBDBDDACC
 [b21521778@rdev Assignment4]$ █
```

2 – Encoding Algorithm

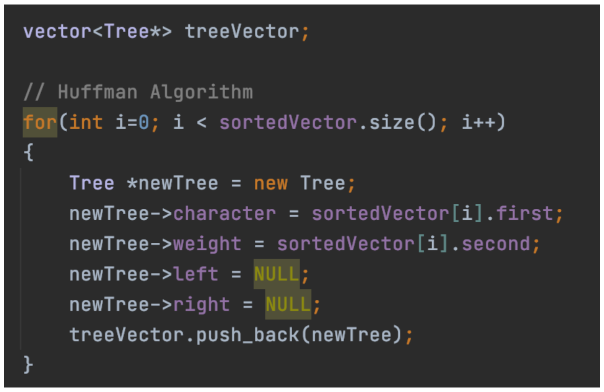
The program takes the second command-line argument as a file name and while reading that file its stores characters inside the charactersVector which is defined in FileOperation class. After that, I calculate frequencies inside calculateFrequency function and store character and their occurrences inside the pair-based vector which is called sortedVector. I sort this vectors with sortMap function so first element of vector has minimum occurrences. (Figure 1)



```
while(inputFile >> character)
{
    input += character;
    charactersVector.push_back(character);
}

//calculate Frequency
calculateFrequency( vec: charactersVector, &: myMap);

sortMap( &: myMap,  &: sortedVector);
```

Figure 1

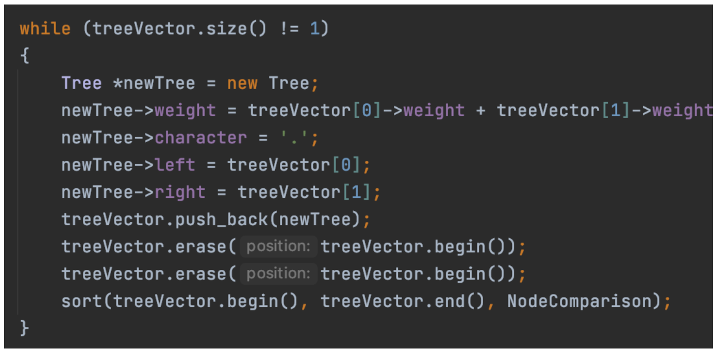
At this point, we have sufficient information for beginning the use Huffman algorithm. For this purpose, I create a node that has weight, character, left node and right node. While I am iterating my sorted vector, I create a new node for each element and fill that node with the corresponding character and weight(frequencies). Each node also points NULL for left and right. I push backed every node inside a treeVector which is hold tree node pointers. At the end of the iteration, I have separated tree nodes inside the tree vector. (Figure 2)



```
vector<Tree*> treeVector;

// Huffman Algorithm
for(int i=0; i < sortedVector.size(); i++)
{
    Tree *newTree = new Tree;
    newTree->character = sortedVector[i].first;
    newTree->weight = sortedVector[i].second;
    newTree->left = NULL;
    newTree->right = NULL;
    treeVector.push_back(newTree);
}
```

Figure 2

At this time, none of the trees(nodes) has any linked between each other. The vector of trees has a number of elements that equals a number of different characters of input. For instance, if the input is "ABCDEABC" there will be 5 nodes. Thus, there are 5 node addresses inside pointers of the vector tree.

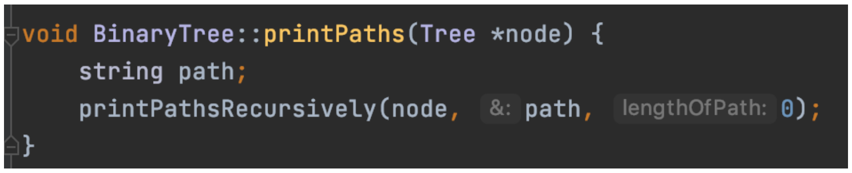


```
while (treeVector.size() != 1)
{
    Tree *newTree = new Tree;
    newTree->weight = treeVector[0]->weight + treeVector[1]->weight
    newTree->character = '.';
    newTree->left = treeVector[0];
    newTree->right = treeVector[1];
    treeVector.push_back(newTree);
    treeVector.erase( position: treeVector.begin());
    treeVector.erase( position: treeVector.begin());
    sort(treeVector.begin(), treeVector.end(), NodeComparison);
}
```

Figure 3

We want to combine these trees according to Huffman encoding algorithms. Every tree connects another tree and at the end of the iteration and we have only a root node. For this operation in each iteration, I create a new Tree and assign the corresponding weight. This weight has been taken from the first and second elements of the tree vector. After that, I added these two weights I assign into a new tree. The new tree has a left tree that has been taken from the first tree from the vector of a tree and the right tree has been taken from the second element of the vector of a tree. New tree pushed back into tree vector. After new tree insertion, I delete the first and second tree from the tree vector and sort the tree vector according to the weight of the trees. In this implementation, we add one new tree and delete two trees from the vector of trees. In every iteration, we have one less tree inside the vector of trees. When we have only one element inside the tree vector, the loop will terminate. Finally, we construct a binary tree according to the given input. If the node has linked another node their characters are represented with "." otherwise, it has character letters which have taken from the given input. (Figure 3)

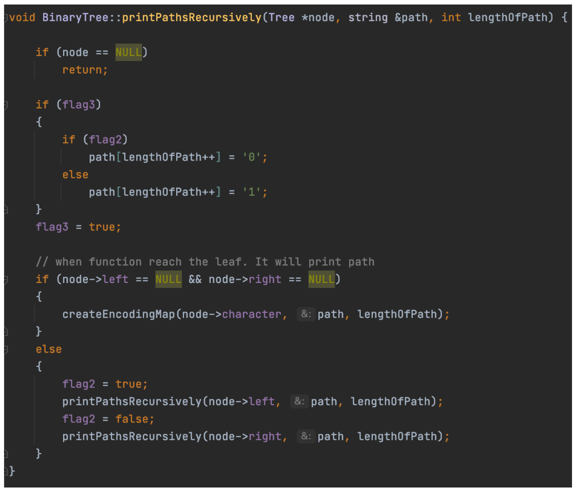
For creating an encoding table, we need to print paths inside that binary tree. For this operation, I use a recursive method. Before the calling recursive function, I use the helper function that holds the path as a string and the length of the path. (Figure 4)



```
void BinaryTree::printPaths(Tree *node) {
    string path;
    printPathsRecursively(node,  &: path,  lengthOfPath: 0);
}
```

Figure 4

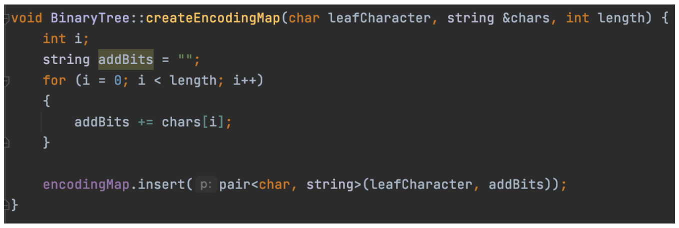
In the recursive method, I use two flags for tracking binary tree route. Flag3 is false when this function has been called, and this will help to understand code is doing the operation in the root node. After the first call, the function calls itself with the left node. If flag2 is true which is the case now, it inserts '0' into the path string. After the left part complete, flag2 returned the false and function call itself with the right node. At this point, the function inserts, '1' into the path string, due to flag2 is false. If the function reaches the leaf node which means left and right nodes point to null, it will call the encoding map function. (Figure 5)



```
void BinaryTree::printPathsRecursively(Tree *node, string &path, int lengthOfPath) {

    if (node == NULL)
        return;

    if (flag3)
    {
        if (flag2)
            path[lengthOfPath++] = '0';
        else
            path[lengthOfPath++] = '1';
    }
    flag3 = true;

    // when function reach the leaf. It will print path
    if (node->left == NULL && node->right == NULL)
    {
        createEncodingMap(node->character,  &: path, lengthOfPath);
    }
    else
    {
        flag2 = true;
        printPathsRecursively(node->left,  &: path, lengthOfPath);
        flag2 = false;
        printPathsRecursively(node->right,  &: path, lengthOfPath);
    }
}
```

Figure 5

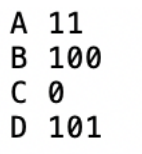
Inside the createEncodingMap function, I insert the character and corresponding bit sequence into the encoding map. After this operation has done encoding is complete and the encoding table is holding inside the encoding map. (Figure 6)



```
void BinaryTree::createEncodingMap(char leafCharacter, string &chars, int length) {
    int i;
    string addBits = "";
    for (i = 0; i < length; i++)
    {
        addBits += chars[i];
    }

    encodingMap.insert( p: pair<char, string>(leafCharacter, addBits));
}
```

Figure 6

Encoding map is written into decodeTable.txt file after all recursion operation is complete. You could see the example of decodeTable.txt in the below. (Figure 7)

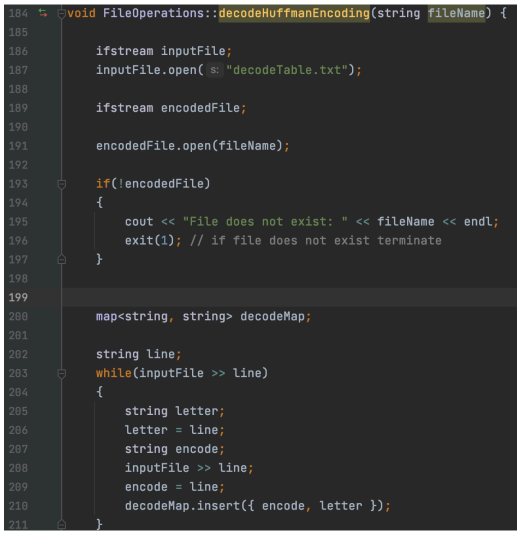


```
A 11
B 100
C 0
D 101
```

Figure 7

3 – Decoding Algorithm

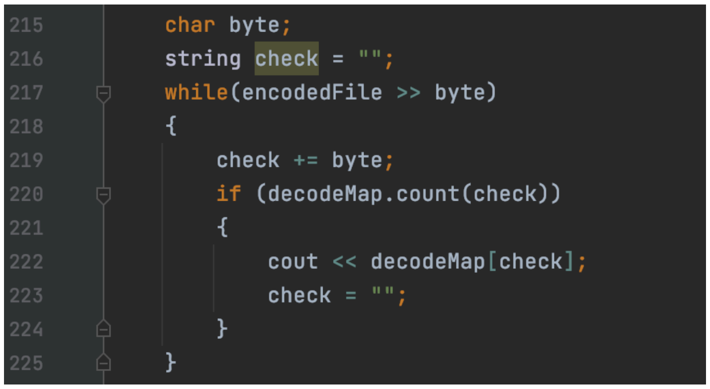
Inside the decodeHuffmanEncoding function, I read decodeTable.txt which is demonstrated in Figure 7. While reading that file, I create a new decodeMap which has keys as bits, and values as characters. Basically, I just reversed encodingMap and store it inside the decodeMap. (Figure 8)



```
void FileOperations::decodeHuffmanEncoding(string fileName) {

    ifstream inputFile;
    inputFile.open( s: "decodeTable.txt");

    ifstream encodedFile;

    encodedFile.open(fileName);

    if(!encodedFile)
    {
        cout << "File does not exist: " << fileName << endl;
        exit(1); // if file does not exist terminate
    }


    map<string, string> decodeMap;

    string line;
    while(inputFile >> line)
    {
        string letter;
        letter = line;
        string encode;
        inputFile >> line;
        encode = line;
        decodeMap.insert({ encode, letter });
    }
```

Figure 8

Inside the same function, I read given encodedFile by char. I insert char into the check string. After that, I am checking that check string inside decodeMap or not. If it is, I print out the map value which is corresponding to the given key and clear out the check string for the next iteration. If it is not, I continue to insert the coming char into the check string. (Figure 9)

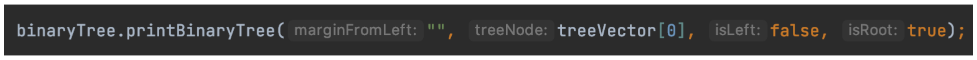


```
char byte;
string check = "";
while(encodedFile >> byte)
{
    check += byte;
    if (decodeMap.count(check))
    {
        cout << decodeMap[check];
        check = "";
    }
}
```

Figure 9

4 – List Tree Command

In the encoding algorithm explanation, I gave details about my approach for gathering nodes and construct a binary tree from scratch. I store a binary tree inside a treeVector which is held tree node pointers. I send the root of my binary tree to print the binary tree function. Additionally, I send isLeft as a false, and isRoot as a true. In this way, the function knows it starts from the root. The margin is for aesthetic demonstration. (Figure 10)



```
binaryTree.printBinaryTree( marginFromLeft: "", treeNode: treeVector[0], isLeft: false, isRoot: true);
```

Figure 10

printBinaryTree function is a recursive function. It starts from the root and iterate all nodes inside the binary tree.

Algorithm for Printing Binary Tree

1. Base case: check tree node is NULL. While it‘s not NULL call itself recursively
2. If is root true then write out the root character into “binaryTreeEncoding.txt”
3. If the is root false then the check is left is true. If it’s you’re inside the left node. Check that node has any child. If that node has any child write out the left child into “binaryTreeEncoding.txt”. If that node hasn’t got any child then it’s a leaf. Write out left leaf into “binaryTreeEncoding.txt”
4. If is left is false. You’re inside the right node. Check that node has any child. If that node has any child write out the right child into “binaryTreeEncoding.txt”. If that node hasn’t got any child then it’s a leaf. Write out right leaf into “binaryTreeEncoding.txt”
5. Print out characters for the corresponding node into “binaryTreeEncoding.txt”. If a node’s character is equal to 32 that means it is equal to ‘.’ in the ASCII table so print out “.” instead of characters.
6. Call itself with the left tree of a node. It means iterate all left nodes inside the binary tree.
7. Call itself with the right tree of a node. It means iterate all right nodes inside the binary tree.
8. Continue to call itself until satisfying the base condition.

5 – Important Note About Inputs

If the given input has \n newline character at the end of the file, my program does not print out correct results.