ASSIGNMENT4 REPORT

BBM203
SOFTWARE LABORATORY 1

Name and Surname : Meltem Kaya
Student ID : 21827555
Due Date : 01/01/2021

HACETTEPE
University

Department of Computer Engineering

—

Teaching Assistant
Yunus Can Bilge

Table of Contents

SOFTWARE DESIGN NOTES

THE MAIN PROBLEM

Textual data is stored using ASCII encoding with 8 bits per character. 8 bits fixed encoding is not an efficient way to store large data. For this reason, there would be an efficient way to store large data. Decoding, encoding and serialise for transfer phases must be implemented with an algorithm chosen to be efficiency.

SOLUTION APPROACH

One of ways storing large data is called as "Huffman Encoding Algorithm". Huffman encoding uses binary encoding with variable sizes. The advantage of this idea is that characters with high frequency is given short encoding while ones with low frequency is given high encoding. Using Huffman Encoding Algorithm, decoding, encoding and serialise for transfer phases can be implemented.
Firstly, I encoded given input data by using Huffman Encoding Algorithm, then I stored each character and its encoded data in dictionary.txt file. Lastly, I print diagram of tree out to tree.txt file.
If given command is "-s", I searched expected decoded value of given character in dictionary.txt and print decoded value out to console.
If given command is "-l", I print contents of tree.txt out to console line by line.
If given command is "-decode", I decoded given input file according to values included by dictionary.txt file.
Note: The program constructs the binary tree exactly once, not over and over.

Data Structures Used

Binary Tree
Binary tree is used for encoding a given string with space efficiency by using Huffman Encoding Algorithm.

Map
Map is used to store characters in a particular string with their frequency.
  Key: character
  Value: frequency
Map is also used to store characters in a particular string with their encoded data.
  Key: character
  Value: Encoded Data (binary code)

List
After reading input string, list is used for storing nodes which are includes each character in input string. Each node will be used for constructing binary tree later.

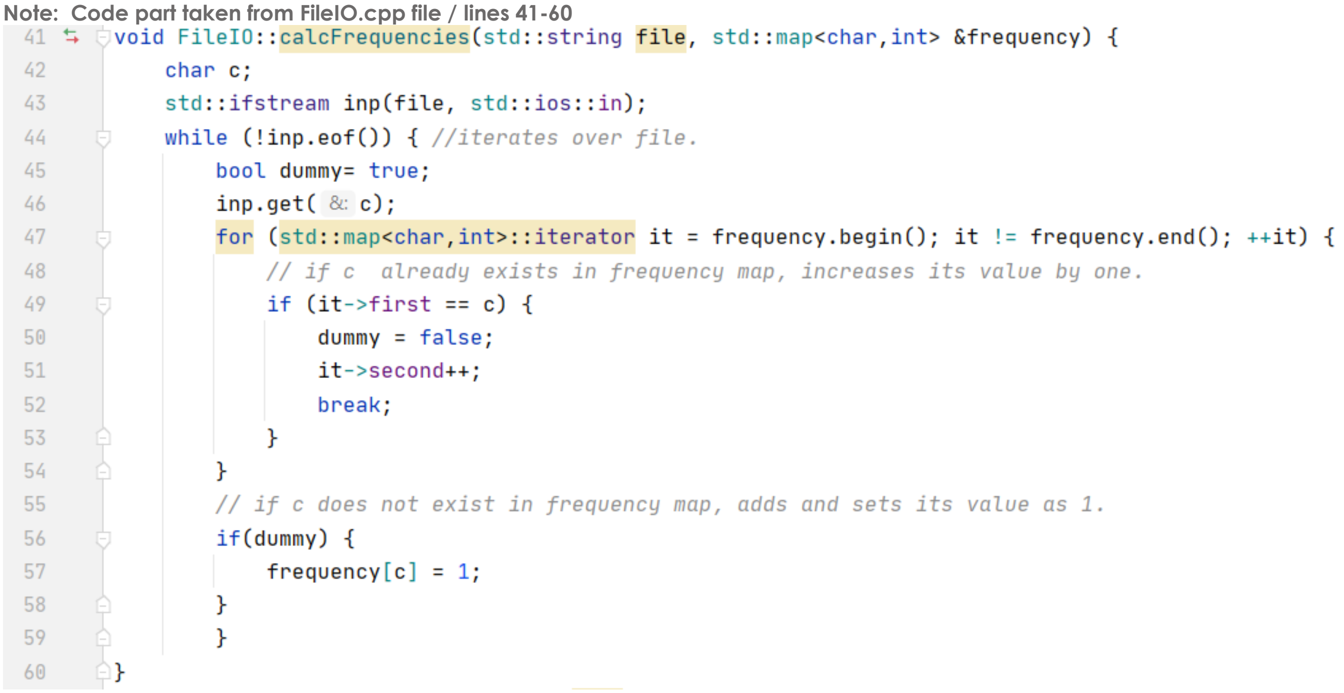
Encoding Algorithm

Encoding Algorithm is based on six steps.
1. Read input file and calculate frequencies of each character given in the input file.
2. Create Nodes to store each character with their frequencies and store nodes in a list in ascending order by their frequencies.
3. Construct a binary tree by using Huffman Encoding Algorithm.
4. Encode input file.
5. Print encoded data out to dictionary.txt file.
6. Print Diagram of Tree out to tree.txt file.

1. Read input file and calculate frequencies of each character given in the input file

1. While there is no error occurred, read input file character by character (lines 44-60)
- **1.1.** get character c from input file (line 46)
- **1.2.** iterate over frequency map in for loop (lines 47-54)
  - **1.2.1.** if character exists in frequency map, increase its value by one. (lines 49-53)
  - **1.2.2.** else add character c into frequency map and set its value as 1. (lines 56-58)

Note: Code part taken from FileIO.cpp file / lines 41-60



```
41  void FileIO::calcFrequencies(std::string file, std::map<char,int> &frequency) {
42      char c;
43      std::ifstream inp(file, std::ios::in);
44      while (!inp.eof()) { //iterates over file.
45          bool dummy= true;
46          inp.get( &: c);
47          for (std::map<char,int>::iterator it = frequency.begin(); it != frequency.end(); ++it) {
48              // if c  already exists in frequency map, increases its value by one.
49              if (it->first == c) {
50                  dummy = false;
51                  it->second++;
52                  break;
53              }
54          }
55          // if c does not exist in frequency map, adds and sets its value as 1.
56          if(dummy) {
57              frequency[c] = 1;
58          }
59          }
60  }
```

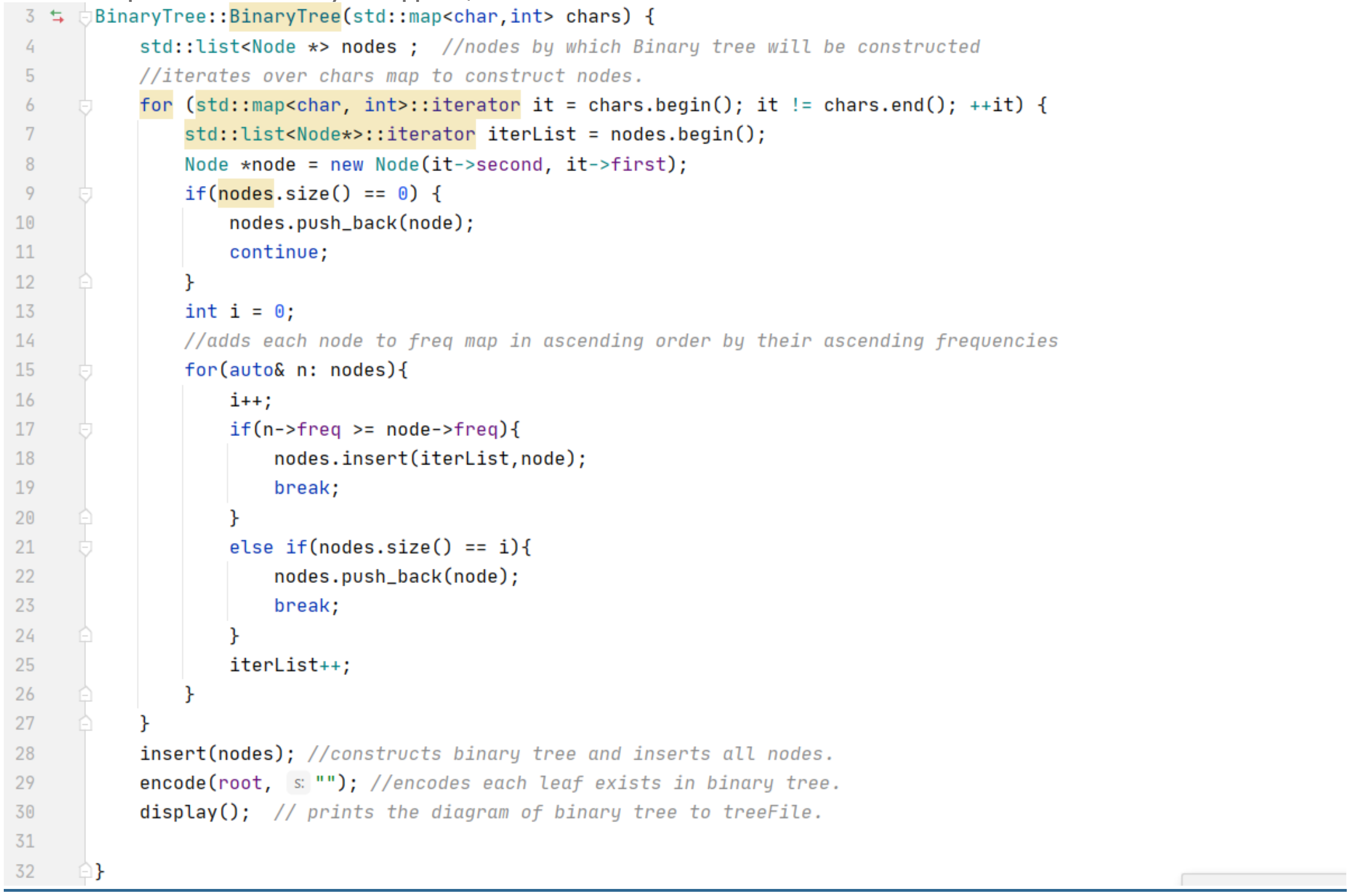
2. Create Nodes to store each character with their frequencies and store nodes in a list in ascending order by their frequencies

1. Create nodes list which contains Node pointers. (line 4)
2. Iterate over each member of chars map in for loop. (lines 6-27)
- **2.1.** create a node with member key and value. (line 8)
- **2.2.** If nodes size equals to zero, push back node to nodes list and continue. (lines 9-12)
- **2.3.** Else, iterate over each “n“ node in nodes list in for loop. (lines 13-26)

2.3.1 If "n " node frequency is greater than or equal to node frequency, insert node to index of "n" and break. (lines 17-20)

2.3.2 else if, node frequency is greater than all of the "n" nodes exist in nodes list, push back node to nodes list and break. (lines 21-24)

Note: Code part taken from BinaryTree.cpp file / lines 3-32

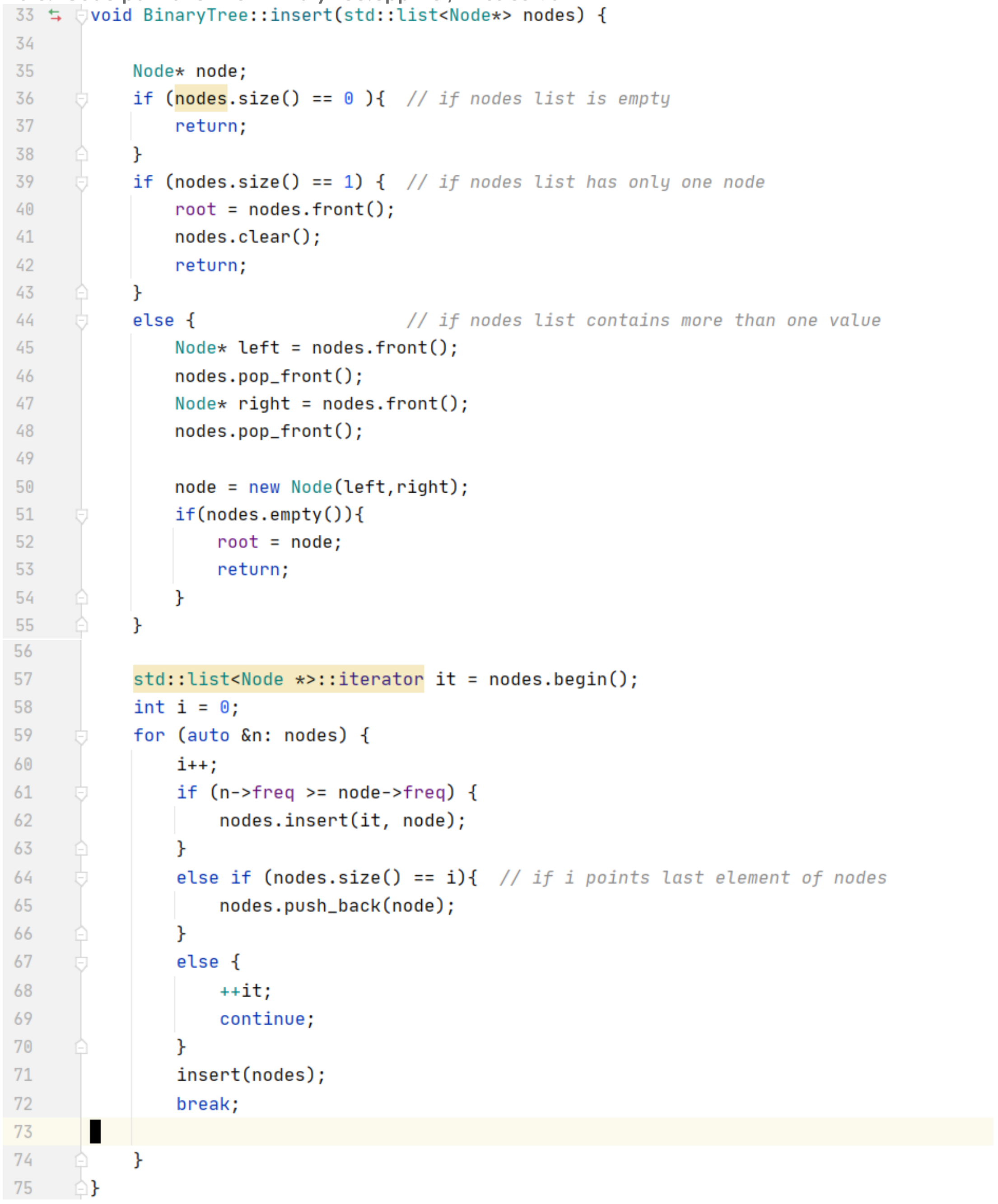


```
 3  BinaryTree::BinaryTree(std::map<char,int> chars) {
 4      std::list<Node *> nodes ;  //nodes by which Binary tree will be constructed
 5      //iterates over chars map to construct nodes.
 6      for (std::map<char, int>::iterator it = chars.begin(); it != chars.end(); ++it) {
 7          std::list<Node*>::iterator iterList = nodes.begin();
 8          Node *node = new Node(it->second, it->first);
 9          if(nodes.size() == 0) {
10              nodes.push_back(node);
11              continue;
12          }
13          int i = 0;
14          //adds each node to freq map in ascending order by their ascending frequencies
15          for(auto& n: nodes){
16              i++;
17              if(n->freq >= node->freq){
18                  nodes.insert(iterList,node);
19                  break;
20              }
21              else if(nodes.size() == i){
22                  nodes.push_back(node);
23                  break;
24              }
25              iterList++;
26          }
27      }
28      insert(nodes); //constructs binary tree and inserts all nodes.
29      encode(root,  s: ""); //encodes each leaf exists in binary tree.
30      display();  // prints the diagram of binary tree to treeFile.
31
32  }
```

3. Construct a binary tree by using Huffman Encoding Algorithm

1. If nodes list is empty, return. (lines 36-38)
2. If nodes list has only one node, set root to node which exists node list. (lines 39-43)
3. If nodes list has more than one node create **a new node.** (lines 45-50)
   3.1. Set **new node**'s left node to first element of nodes list and delete first element of nodes list.
   3.2. Set **new node**'s right node to first element of nodes list and delete first element of nodes list.
4. If nodes list is empty, set root to **new node**. (lines 51-54)
5. Iterate over each element *(declared as "**n**")* of nodes list in for loop. (lines 59-74)
   5.1. If frequency of **n** is greater than or equal to new node's frequency, insert **new node** to index of **n** in nodes list. (lines 61-63)
   5.2. Else if frequency of **new node** is greater than all elements, push back new node to nodes list. (lines 64-66)
   5.3. Recursive call of insert function with updated nodes list. (line 71)

Note: Code part taken from BinaryTree.cpp file / lines 33-75



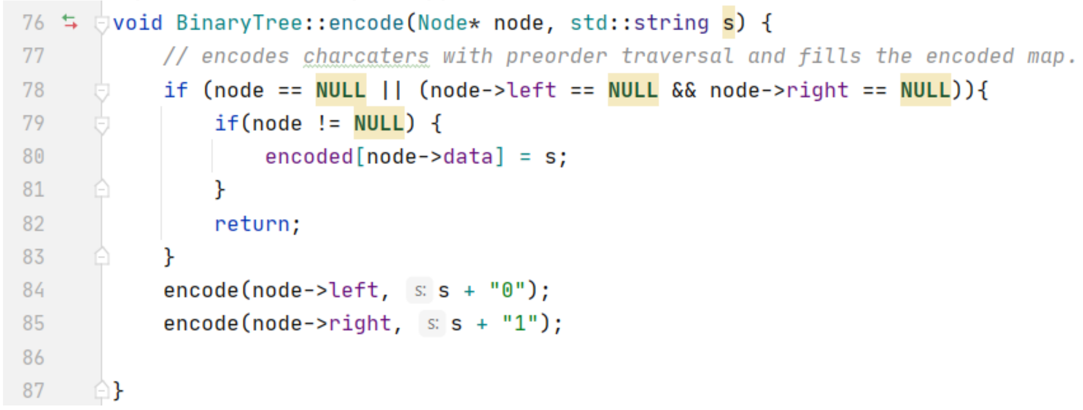
```
33  void BinaryTree::insert(std::list<Node*> nodes) {
34
35      Node* node;
36      if (nodes.size() == 0 ){  // if nodes list is empty
37          return;
38      }
39      if (nodes.size() == 1) {  // if nodes list has only one node
40          root = nodes.front();
41          nodes.clear();
42          return;
43      }
44      else {                    // if nodes list contains more than one value
45          Node* left = nodes.front();
46          nodes.pop_front();
47          Node* right = nodes.front();
48          nodes.pop_front();
49
50          node = new Node(left,right);
51          if(nodes.empty()){
52              root = node;
53              return;
54          }
55      }
56
57      std::list<Node *>::iterator it = nodes.begin();
58      int i = 0;
59      for (auto &n: nodes) {
60          i++;
61          if (n->freq >= node->freq) {
62              nodes.insert(it, node);
63          }
64          else if (nodes.size() == i){  // if i points last element of nodes
65              nodes.push_back(node);
66          }
67          else {
68              ++it;
69              continue;
70          }
71          insert(nodes);
72          break;
73
74      }
75  }
```

4. Encode input file

1. Traverse in binary tree by using preorder traversal with recursive call.
1.1. if node is not a leaf,
  1.1.1. recursive call for left branch of node and **s** is concatenated with "0". (line 84)
  1.1.2. recursive call for right branch of node and s is concatenated with "1". (line 85)
2. if node is a leaf and leaf is not empty , add data of node with value **s** to ***encoded map****. (lines 78-83)

** **encoded map** is a variable of Binary Tree class which stores characters and their encoded data*

Note: Code part taken from BinaryTree.cpp file / lines 76-87

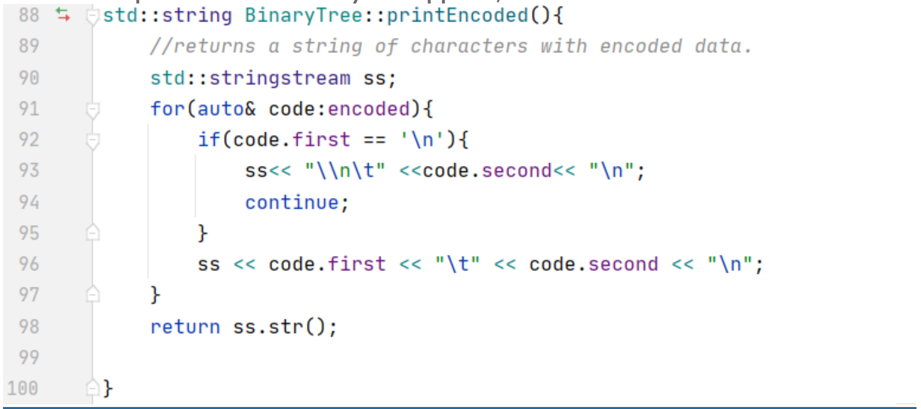


```
76  void BinaryTree::encode(Node* node, std::string s) {
77      // encodes charcaters with preorder traversal and fills the encoded map.
78      if (node == NULL || (node->left == NULL && node->right == NULL)){
79          if(node != NULL) {
80              encoded[node->data] = s;
81          }
82          return;
83      }
84      encode(node->left,  s: s + "0");
85      encode(node->right,  s: s + "1");
86
87  }
```

5. Print encoded data out to dictionary.txt file

1. Create a stringstream object declared as "ss". (line 90)
2. Iterate over each element declared as ***code*** of encoded map. (lines 91-100)
   2.1. if key equals to "\n" , concatenate ss with "\\***n******tvalue******n***". (lines 92-95)
   2.2. Else, Concatenate ss with "***key******tvalue******n***". (line 96)
3. Convert stringstream object to string object and return its value. (line 98)

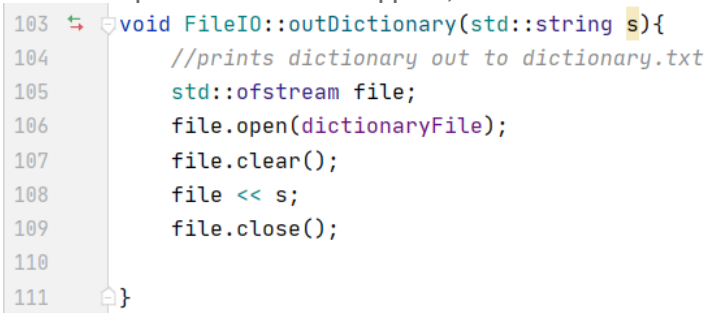
Note: Code part taken from BinaryTree.cpp file / lines 88-100



```
88  std::string BinaryTree::printEncoded(){
89      //returns a string of characters with encoded data.
90      std::stringstream ss;
91      for(auto& code:encoded){
92          if(code.first == '\n'){
93              ss<< "\\n\t" <<code.second<< "\n";
94              continue;
95          }
96          ss << code.first << "\t" << code.second << "\n";
97      }
98      return ss.str();
99
100 }
```

4. Open dictionary file(dictionary.txt) (line 106)
4.1. print string which returned by printEncoded() method out. (line 108)
5. Close file. (line 109)

Note: Code part taken from FileIO.cpp file / lines 103-111

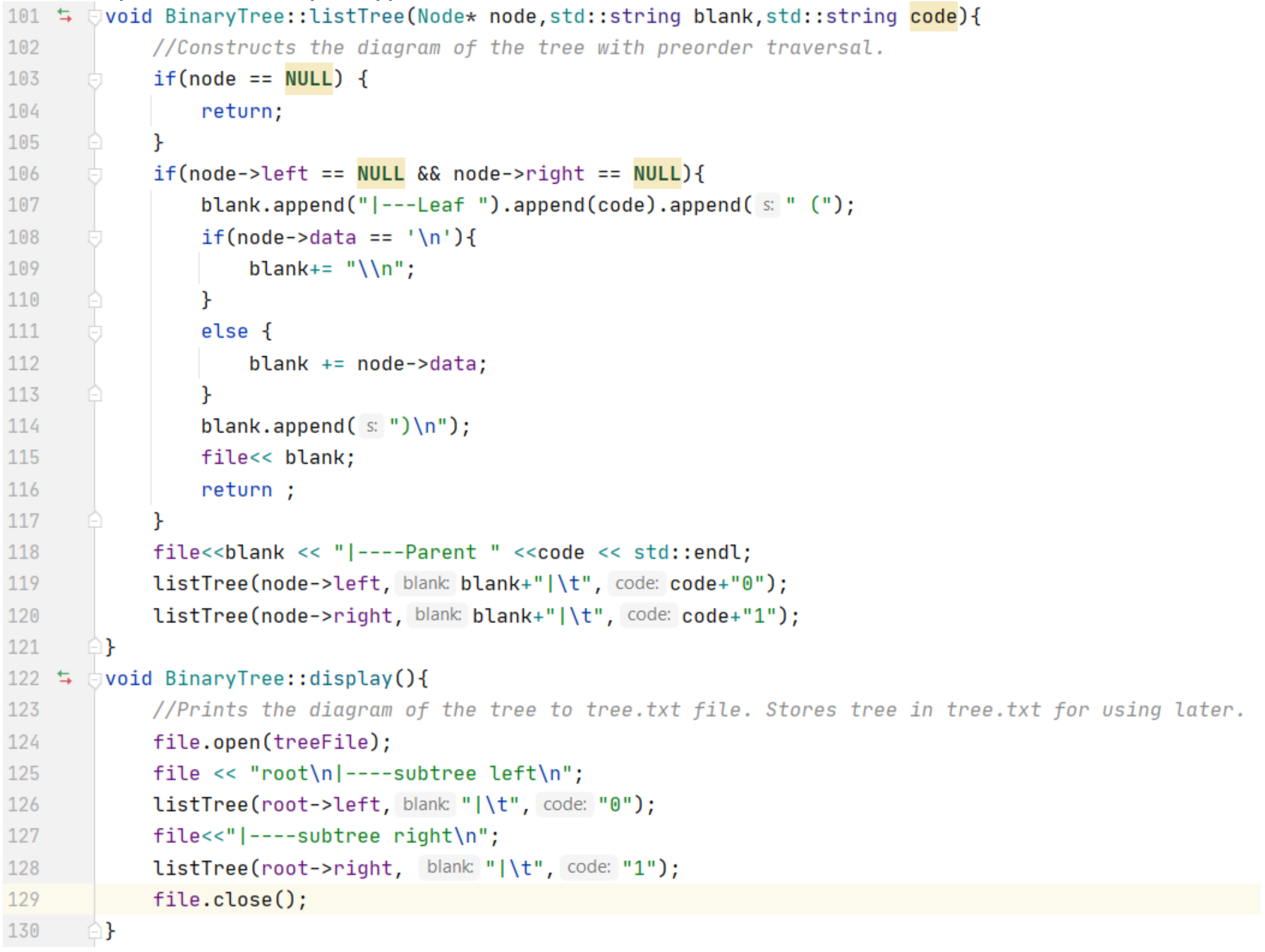


```
103 void FileIO::outDictionary(std::string s){
104     //prints dictionary out to dictionary.txt
105     std::ofstream file;
106     file.open(dictionaryFile);
107     file.clear();
108     file << s;
109     file.close();
110
111 }
```

6. Print Diagram of Tree out to tree.txt file

1. Traverse on binary tree by using preorder traversal. (lines 101-121)
  1.1. If node is null, return. (line 103)
  1.2. If node is a leaf update blank parameter and return. (lines 106-117)
2. Recursive call for left branch of tree. (line 119)
3. Recursive call for right branch of tree. (line 120)
4. open tree.txt. (lines 124-128)
  4.1. Print blank out to tree.txt file.
5. Close file. (line 129)

Note: Code part taken from BinaryTree.cpp file / lines 101-130

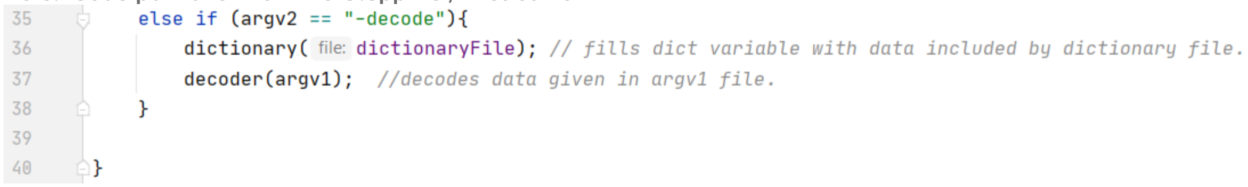


```
101  void BinaryTree::listTree(Node* node,std::string blank,std::string code){
102      //Constructs the diagram of the tree with preorder traversal.
103      if(node == NULL) {
104          return;
105      }
106      if(node->left == NULL && node->right == NULL){
107          blank.append("|---Leaf ").append(code).append( s: " (");
108          if(node->data == '\n'){
109              blank+= "\\n";
110          }
111          else {
112              blank += node->data;
113          }
114          blank.append( s: ")\n");
115          file<< blank;
116          return ;
117      }
118      file<<blank << "|----Parent " <<code << std::endl;
119      listTree(node->left, blank: blank+"|\t", code: code+"0");
120      listTree(node->right, blank: blank+"|\t", code: code+"1");
121  }
122  void BinaryTree::display(){
123      //Prints the diagram of the tree to tree.txt file. Stores tree in tree.txt for using later.
124      file.open(treeFile);
125      file << "root\n|----subtree left\n";
126      listTree(root->left, blank: "|\t", code: "0");
127      file<<"|----subtree right\n";
128      listTree(root->right,  blank: "|\t", code: "1");
129      file.close();
130  }
```

Decoding Algorithm

Decoding Algorithm is based on two steps.
1. Read dictionary.txt to store all characters with their encoded data in a ***map***.
2. To decode given data, iterate over ***map*** and print decoded data out to console.

Note: Code part taken from FileIO.cpp file / lines 35-40



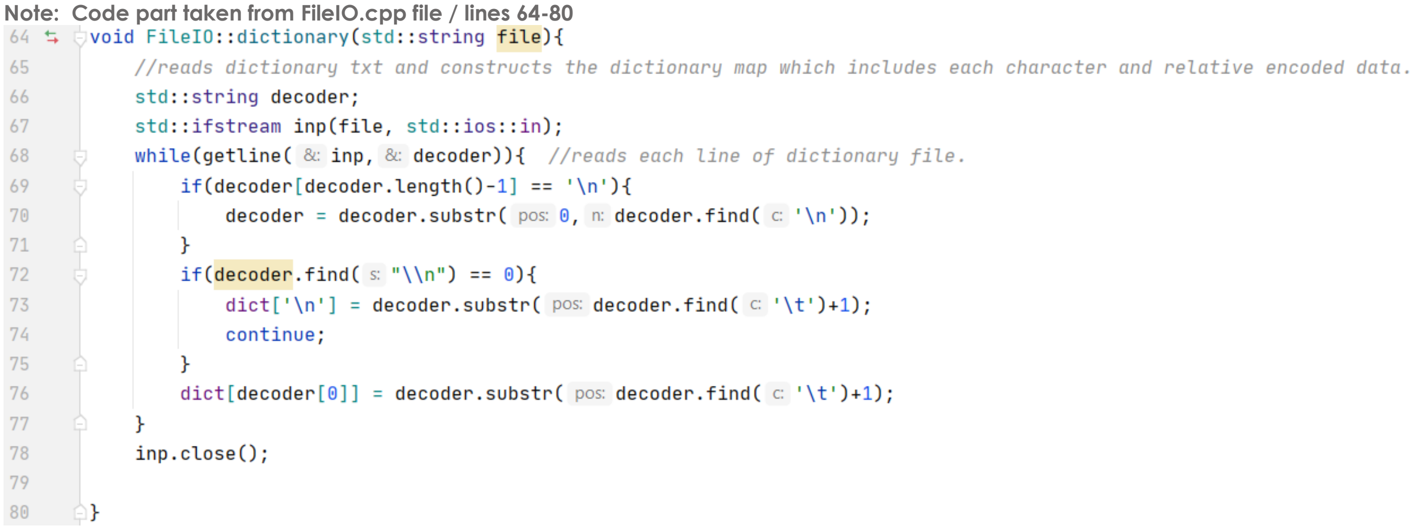
```
35      else if (argv2 == "-decode"){
36          dictionary( file: dictionaryFile); // fills dict variable with data included by dictionary file.
37          decoder(argv1);  //decodes data given in argv1 file.
38      }
39
40  }
```

1. Read dictionary.txt to get each character and encoded data

1. Iterate over dictionary.txt file line by line until reach last line. (lines 67-77)
1.1. if line ends with "\n", throw the last character off the line. (lines 69-71)
1.2. if line starts with "\\n", add ***dict**** to "\n" and set its value given in line. (lines 72-75)
1.3. else add ***dict**** key given in line and set its value. (line 76)
2. Close file. (line 78)

dict* is a variable of FileIO class which all characters and encoded data are stored.

Note: Code part taken from FileIO.cpp file / lines 64-80



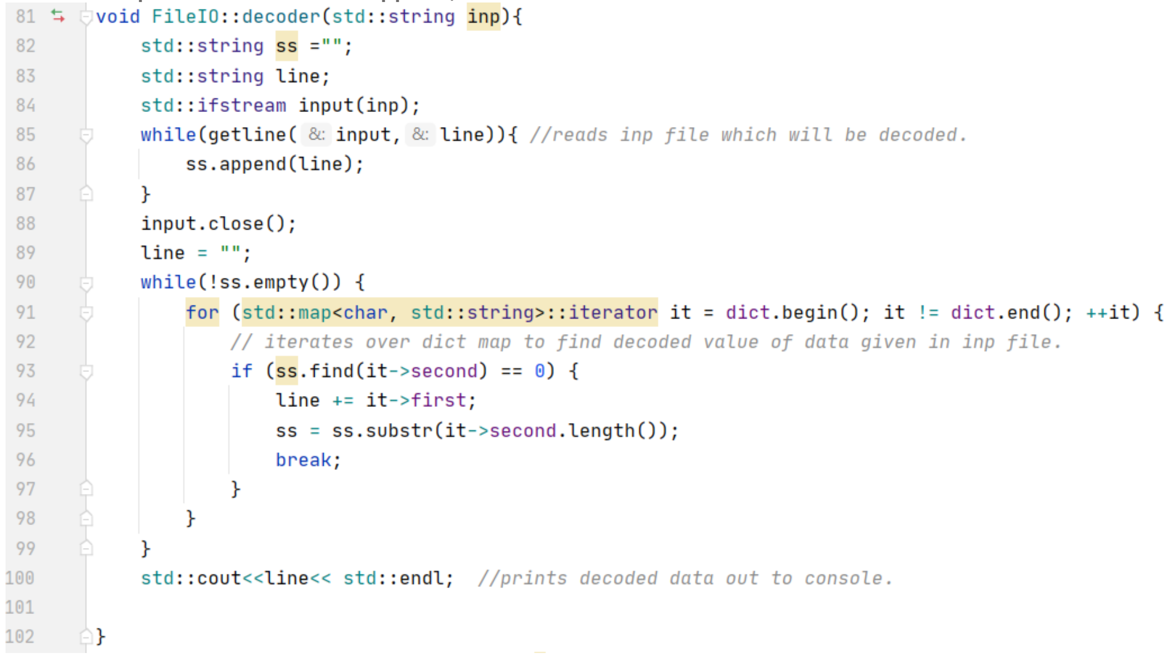
```
64  void FileIO::dictionary(std::string file){
65      //reads dictionary txt and constructs the dictionary map which includes each character and relative encoded data.
66      std::string decoder;
67      std::ifstream inp(file, std::ios::in);
68      while(getline( &: inp, &: decoder)){  //reads each line of dictionary file.
69          if(decoder[decoder.length()-1] == '\n'){
70              decoder = decoder.substr( pos: 0, n: decoder.find( c: '\n'));
71          }
72          if(decoder.find( s: "\\n") == 0){
73              dict['\n'] = decoder.substr( pos: decoder.find( c: '\t')+1);
74              continue;
75          }
76          dict[decoder[0]] = decoder.substr( pos: decoder.find( c: '\t')+1);
77      }
78      inp.close();
79
80  }
```

2. To decode given data, iterate over map and print decoded data out to console

1. Create a stringstream object called as **ss** and string object called as **line**. (lines 82-83)
2. Read input file which would be decoded line by line until end of file. (lines 85-87)
2.1 Concatenate **ss** with each line read in file. (line 86)
3. Iterate over ***dict**** map until ss is decoded and concatenate **line** with stored characters.
4. Print **line** out to console. (line 100)

dict* is a variable of FileIO class which all characters and encoded data are stored.

Note: Code part taken from FileIO.cpp file / lines 81-102

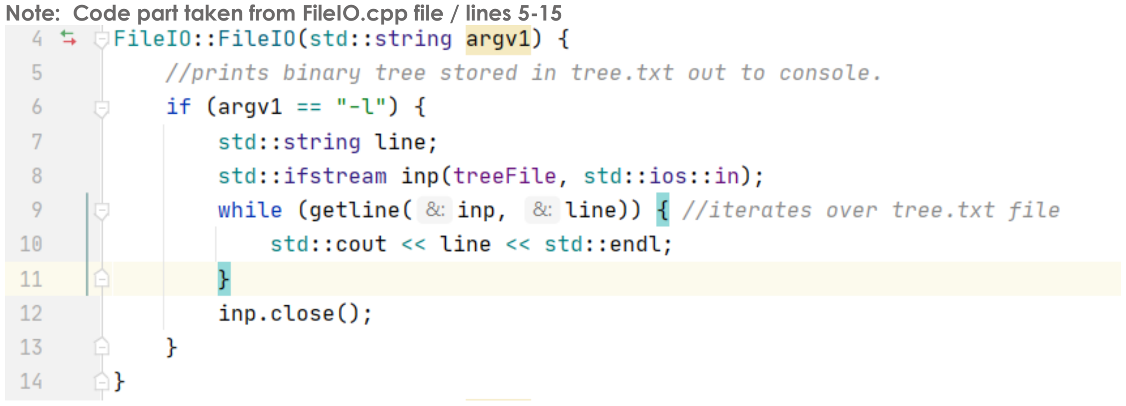


```
81  void FileIO::decoder(std::string inp){
82      std::string ss ="";
83      std::string line;
84      std::ifstream input(inp);
85      while(getline( &: input, &: line)){ //reads inp file which will be decoded.
86          ss.append(line);
87      }
88      input.close();
89      line = "";
90      while(!ss.empty()) {
91          for (std::map<char, std::string>::iterator it = dict.begin(); it != dict.end(); ++it) {
92              // iterates over dict map to find decoded value of data given in inp file.
93              if (ss.find(it->second) == 0) {
94                  line += it->first;
95                  ss = ss.substr(it->second.length());
96                  break;
97              }
98          }
99      }
100     std::cout<<line<< std::endl;  //prints decoded data out to console.
101
102 }
```

List Tree Operation

1. Read tree.txt line by line (lines 9-12)
  1.1 Print each line out to console. (line 11)
2. Close file. (line 13)

Note: Code part taken from FileIO.cpp file / lines 5-15



```
4  FileIO::FileIO(std::string argv1) {
5      //prints binary tree stored in tree.txt out to console.
6      if (argv1 == "-l") {
7          std::string line;
8          std::ifstream inp(treeFile, std::ios::in);
9          while (getline( &: inp, &: line)) { //iterates over tree.txt file
10             std::cout << line << std::endl;
11         }
12         inp.close();
13     }
14 }
```