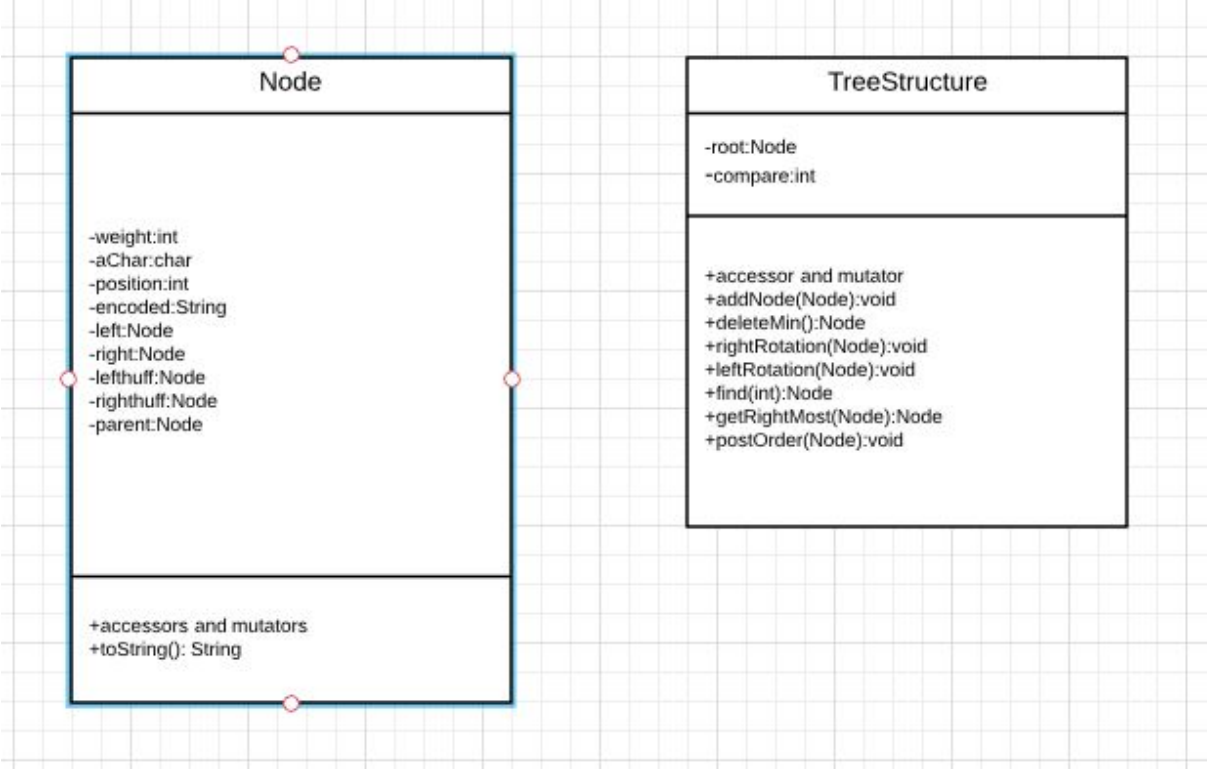
Pierre Tran, 40007601
Assignment 3

1 c)

UML diagram for Node and TreeStructure:



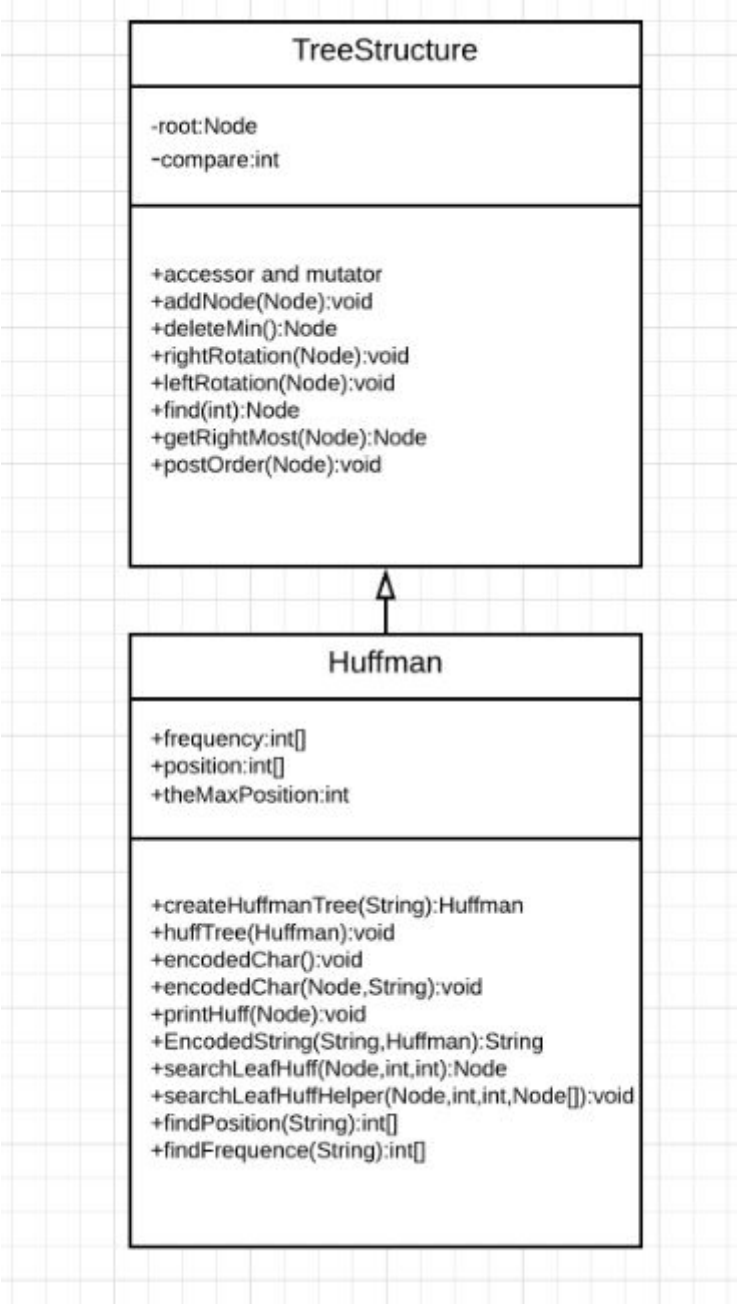
For the tree structure, I implemented a regular binary search tree in “TreeStructure” class. It uses objects from a class called “Node” as its nodes.

I choose to implement the binary search tree for its flexibility, and because it has an average of O(log n) for insertion,deletion and find; when performing one of these operations, half of the current binary tree is “discarded”, if the tree is well balanced. While using a heap would be me efficient for the Huffman coding problem, since it takes O(1) to return the minimum value, solving the Splay tree problem with heap would be more challenging mathematically and less efficient, especially when searching for any node that is not a maximum or a minimum. The binary search tree efficient for both problems.

The class “Node” is implemented with class “TreeStructure” because an object was needed to contain the references and connections that constitutes at the very least a binary search tree. Each node has at the very least a reference to a left and right node and has a weight.

However, when implementing more specific trees such as the Huffman tree and the Splay tree, objects from the class “Node” have more specialized members (therefore, they might not always be used). For the Huffman tree implementation, the object Node has in addition a right/left Huffman node, holds a character and its position and holds an encoded string of bits. For the Splay tree implementation, the object Node has in addition a reference to its parent.

1 e)

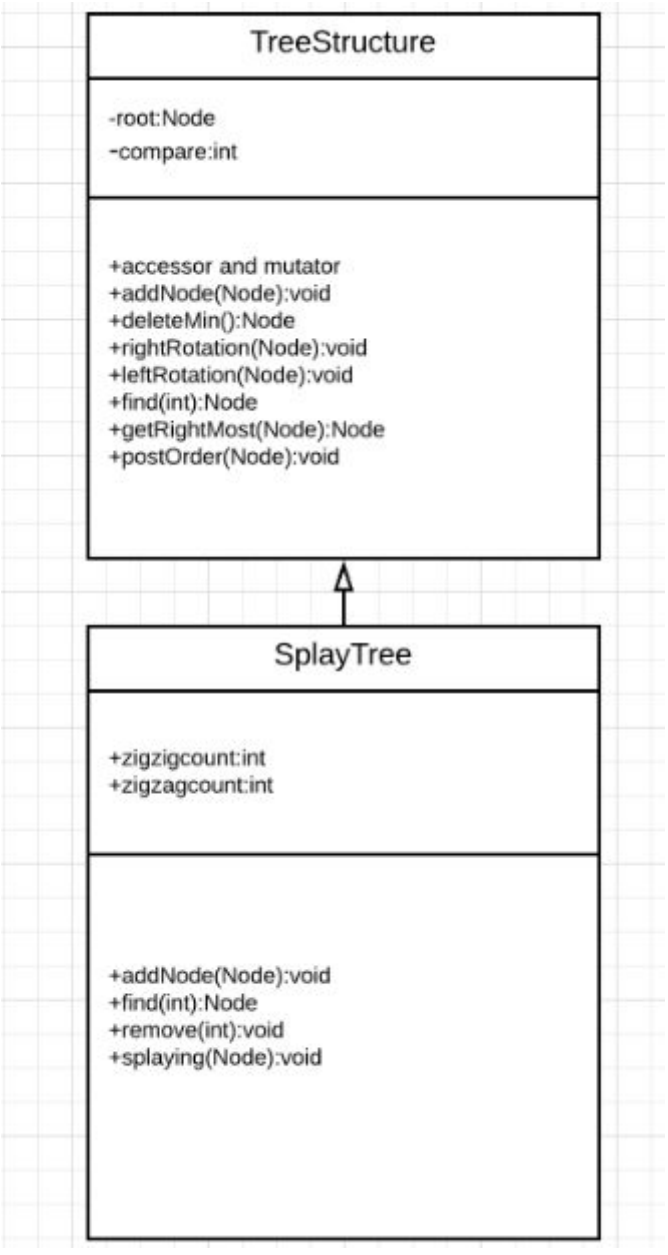


The class “Huffman” extends the class “TreeStructure” because much of the methods needed to create a Huffman tree are already used in the implementation of regular binary search tree. Methods from “TreeStructure” can be used to create a Huffman tree, but methods from “Huffman” are too specialized to create an ordinary binary tree. The “Huffman” class uses getRoot(), setRoot(), addNode() and deleteMin() from the “TreeStructure” class.

addNode() is used in the createHuffmanTree() method. createHuffmanTree() creates a node for every character in its argument and adds the node in a binary search tree. Every node represents a character, has a weight (the frequency of the character in the passed argument) and has a position (the first absolute occurrence of the character in the passed argument). Thanks to the addNode() method, every node has a reference to a left/right/parent node in the tree (although the reference of the parent is not used in the implementation of the Huffman tree). This method initially creates a binary search tree, then creates a Huffman tree from it using the method huffTree(), and finally encodes every leaf node using method encodedChar().

The deleteMin() method is used in the huffTree() method. huffTree() creates little by little a Huffman tree off the original binary tree; it uses deleteMin() to find two nodes with the least weight and merges them together. The merged node’s weight is the sum of the weights of the nodes that have been merged, its position is incremented starting from the highest position (the position of the very last character in the argument passed to createHuffmanTree()) and its character is initialized to the null character ‘\0’ (every parent node holds a null character). The merged node is then added to the binary tree. Every merged node that is sent back to the binary tree has references pointing to a left/right “Huffman” node, this is to prevent interferences when using deleteMin() method; had “Huffman” left/right nodes not been implemented, deleteMin() would always return the same node.
This process is repeated until the size of the binary tree reaches one. The huffTree() method imitates priority queuing.

1 g)



The class “SplayTree” extends class “TreeStructure” because much of the methods found in a regular binary search tree can also be found in a Splay tree. Methods used in the class “TreeStructure” can be used to build other kinds of trees including Splay trees, but methods used in class “SplayTree” are too specialized to be used elsewhere. The ”SplayTree” class uses getRoot(),setRoot(), addNode(), rightRotation(), leftRotation(), find(), getRightMost() and postOrder() from the class “TreeStructure”.

addNode() is used in the overridden addNode() method from the “SplayTree” class. After adding a node, the node gets splayed and becomes the new root node of the tree object.

rightRotation() and leftRotation() are used in splaying() method, which in return is used in other methods in “SplayTree” class. rightRotation() and leftRotation() are used when a zig, a zig-zig or a zig-zag situation occurs.

splaying() method is used in addNode(), remove() and find() methods. Essentially, when a node has been recently accessed, it is splayed until it becomes the new root of the tree. When removing a node, the node that is to be removed is splayed, the rightmost node of the left subtree of the root is splayed, the root gets deleted and the references between the left and right subtree are connected.

postOrder() method is used to display the content of the tree in post order.