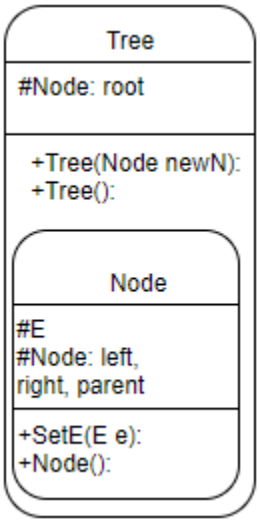
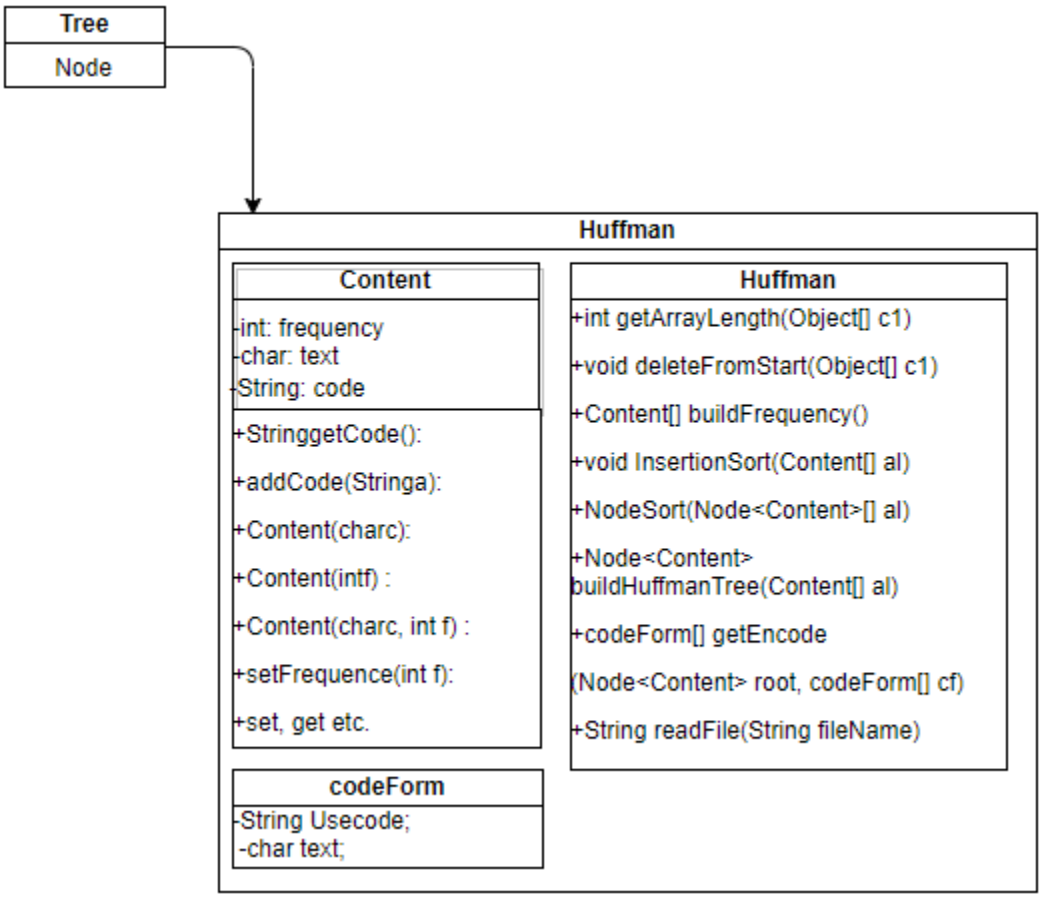
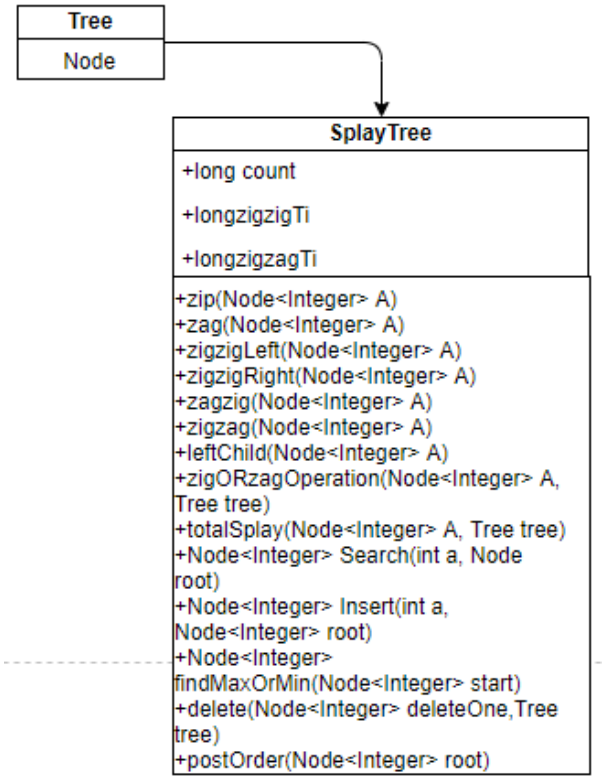
Question 1 (b)



Question 1 (d)



Question 1 (f)


Tree
Node
SplayTree
+long count
+longzigzigTi
+longzigzagTi
+zip(Node<Integer> A)
+zag(Node<Integer> A)
+zigzigLeft(Node<Integer> A)
+zigzigRight(Node<Integer> A)
+zagzig(Node<Integer> A)
+zigzag(Node<Integer> A)
+leftChild(Node<Integer> A)
+zigORzagOperation(Node<Integer> A, Tree tree)
+totalSplay(Node<Integer> A, Tree tree)
+Node<Integer> Search(int a, Node root)
+Node<Integer> Insert(int a, Node<Integer> root)
+Node<Integer> findMaxOrMin(Node<Integer> start)
+delete(Node<Integer> deleteOne,Tree tree)
+postOrder(Node<Integer> root)

1(c) Using LinkedList as the tree structure to design a tree. Using linkedlist because it's easier to track using Node left, right and parent. Also, using generic type as the object of the tree, for further implement of the tree. For Huffman coding, use a Content class as generic type E for storing both frequency and characters. For splay tree, use Integer as generic type E for add/delete and further splay.

About the tree, for Huffman code we use Node left and right in order to track and compare them and then put them in certain place of the tree. For splay tree, we use left, right and parent because when doing zigzag/zigzig it's inevitable to track to parent and change their position.

1(e)First, extends Tree class as Huffman structure. For the generic type in Tree E, use Content, which store int frequency and char text. Then use BuildFrequency() method to build an array of Content, which store character and its frequency. BuildFrequency() method call readFile() method to get the String which the frequency array based on. Once get the frequency array, call InsertionSort() method using insertion sort to sort the array base on frequency in descending order. Then we put the sorted array into buildHuffmanTree() mehod, which will generally build a Huffman Coding tree based on frequency array. For buildTree part, it first convert all Content object into Node, then it takes the [0] [1] of Node array combine into a new node, [0] is the left child as it weighted less or appears later, [1] is the right child it weighted more or appears first (This also apply to new combined Node as it will be put on the start of the array then using in-place sorting, so if same weight, it weighted less therefore appears on the left). For the new combined node, we put it to the start of the array and call NodeSort() method, which is an in-place insertion sort, it will make sure if new combined node and the node that already in array are the same weight, new combined node appears later, so weighted less, so is the left child of the tree. We keep doing this combine-put into array-sorting process until all elements in array are sorted. Then the buildHuffmanTree() method return a Node which is the root of the tree. After built the tree, we call getEncode method, which will trace the entire tree, for each character turn left store 0 and turn right store 1 until reach the leaf, then store the encoding into each codeForm object. The we read the required String from user input, for each character in String print out its encode.

1(g)

For SplayTree class, it use left, right, parent to track within the tree. First we use Scanner to read command from each line of the file, split it to add/delete/search operations, and then get the number.

For add operation, first check if the tree exists, if not, create the root. If yes, first call Insertion() method, it will take an int that need to be added and its tree, using BinarySearch to find the right place to insert it(the proper leaf to insert it) and then create a Node and linked it into the tree and return the Node. After added the number, we need splay it into root or root's child. We call totalSplay() method, first it track back to the leaf's parent to see if its left/right zigzag or zigzag operation, then it call method accordingly for example zigzigLeft() to finish the splay process. For one zigzag it generally change the position of the added Node, parent, grandparent and change their children accordingly. If grandparent is not root, we have to link new Node to its new parent. After finish this zigzag process, totalSplay() method check if the new Node now is the root or root's child, if not, keep doing the splay process until it is. After splay finished, if the new Node is now root, change tree's root to it. If not, do nothing.

For search operation, first we check if the tree exists, if not, do nothing. If yes, we use Search method, which will take an int to be searched and the tree. Trace all the way to leaf, if find the int, return the Node, if not, return null. After search it, we still use totalSplay() method to push the search element to root or root's child.

For delete operation, first we check if the tree exists, if not, do nothing. If yes, we first use Search method and return the Node. If node is null, do nothing. If we got a returned Node, delete the Node first. For delete method, it first call findMaxorMin() method to find the max element of left subtree or min of right subtree, replace current Node with the found Node. For found node, delete its original place also. Keep doing this until replace Node is the

leaf and has nothing to replace to. After delete, call totalSplay method push the delete Node's parent to root or root's child.

For a certain stage of the add/delete/ search process (which is in a loop, we keep track the steps). When it's the required print step, call postOrder() method, which will take the root of a tree and keep track it from left to right to parent, and print the track process in postOrder.

For compare/zigzig/zigzag times, we create static long for each of them. At each zigzig/zigzag method, we count++. For compare, each time a method Search, Insertion is executed, count++. After recording these data, print it.

1(h)

Generally AVL tree guarantee an average good lookup $O(\log(n))$. Even though splay tree's worse case is $O(n\log(n))$, but if we look for most recent visit element or near the root element, it will be much faster. In question 3, most find/remove method is to find recent visited element, so here we use splay tree rather than AVL tree to guarantee less comparison. Even for bigger tree for example after operations 2000+, we still just need no more than 15 comparison like listed below
8 2 7 12 4 13 10 17 6 17 5 8 5 8 16 8 13 11 14 16 10 8 10 16 14 10
So the advantages are 1) faster access time on average case 2)memory-efficient 3)less reshape because strictly balance is not required

4(a)

Encoding:

10111001111100111000110010000101000100010100010100010101101001100110001110011111110000010111001111110001100111111100100111111010011011110110101001111111001011

The ASCII is 7 bits for each character, there are 33 char. So Percent: $\frac{154}{231} = 66.67\%$

I think that the encoded string matched the source text frequencies. Take the encode characters as table, we can see that character appears more often has shorter encode.

Text	Frequency	Encode	Text	Frequency	Encode
Empty String	193	00	e	80	1111
t	66	1011	h	61	1001
s	38	11100	o	53	0110
u	21	01000	p	3	01010001
r	33	10101	c	16	101001
h	61	1001	.	5	11001011

4(b)
1) Output:
Traversal at 6: 280,
28438 compares
4637 Zig-Zigs
4408 Zig-Zags
2)
In this question, splay tree is the correct choice. 1) Splay tree takes less operation time than AVL tree, because in Q3 most data that need to be remove/find are at the top of the tree and nearby root and are most frequent used, it's therefore much easier and efficient to use splay tree rather than AVL tree because it always keep the most frequent Node at top. 2) Space efficiency. Splay tree don't have to store the balance condition and two tree's depth, and it don't have to rotate the whole tree every time it's unbalanced 3) Splay tree don't have strict requirement for balance. Also splay tree support splay/merge more efficient.
For further design, if the data we will implement is random, and it requires guaranteed $O(\log(n))$ time complexity, we should use AVL tree. If the data is organized or it's a huge data but we only need to access the most frequent partial data, also we expect an average $O(\log(n))$ or better time complexity, we should use splay tree.

4(a)

Final traversal result for SplayTree:

Traversal at 2677:
2,1,4,3,5,0,7,6,9,12,11,10,14,17,16,18,15,20,23,24,21,19,25,13,28,27,30,33,32,34,31,35,29,36,38,41,41,39,37,43,4
7,46,45,44,42,51,53,54,51,56,60,60,64,60,59,57,55,50,49,67,68,71,72,70,69,75,74,76,73,77,65,81,80,84,82,86,87,8
9,91,90,88,85,79,93,96,95,97,94,100,101,102,99,98,103,92,78,48,26,8,107,108,106,105,112,111,113,116,118,117,
115,114,122,121,123,120,124,119,110,126,128,127,129,131,133,135,132,130,125,109,104,139,138,137,141,142,1
43,140,146,145,147,148,149,151,152,153,157,156,158,155,160,159,154,165,163,161,150,144,168,170,169,170,17
5,174,173,171,177,178,180,184,185,183,188,190,189,192,191,187,196,195,194,200,201,199,198,202,204,205,206,
203,196,193,182,181,179,208,207,176,212,211,210,215,214,217,218,219,216,213,222,225,224,226,221,220,228,2
30,231,229,232,227,209,235,238,237,236,241,242,239,246,245,248,249,251,250,247,244,233,167,253,254,259,25
8,257,261,265,264,263,266,262,268,267,260,270,273,274,275,276,277,271,268,255,280,283,282,284,286,285,281,
287,279,252,136,291,290,291,289,298,299,297,302,301,303,300,305,307,306,304,295,294,292,311,309,313,314,3
17,316,319,320,317,322,323,321,325,326,324,315,312,288,330,330,330,329,332,331,336,335,334,341,342,340,34
5,344,348,350,352,351,347,354,356,360,358,357,361,367,366,368,365,369,364,363,362,355,353,346,372,370,343,
339,337,327,375,374,378,377,376,381,383,382,380,379,378,388,387,386,390,392,391,389,396,396,395,396,398,4
01,401,400,399,392,385,404,406,408,407,405,410,409,412,413,411,418,420,419,422,420,417,422,425,426,428,42
7,429,424,430,416,415,432,431,436,439,441,440,438,444,443,445,442,447,446,437,448,435,449,453,451,455,459,
457,460,462,463,461,456,464,454,450,433,466,467,465,414,402,469,470,474,475,476,472,468,481,481,480,489,4
83,482,492,491,478,495,493,477,498,497,500,501,499,502,504,505,506,503,508,509,510,513,516,518,519,517,51
2,511,507,524,523,524,522,526,529,531,530,527,520,533,538,540,539,537,543,542,543,545,540,536,534,549,548,
550,547,553,555,554,557,559,560,558,562,563,561,556,565,564,551,567,566,546,532,569,573,572,570,576,579,5
78,577,582,588,587,589,586,585,584,591,590,583,593,596,595,594,596,599,592,580,601,603,602,604,606,608,60
7,611,610,605,615,614,617,616,619,619,618,623,623,626,625,624,627,626,622,612,600,631,633,635,632,639,638,
637,636,630,641,642,640,629,643,574,568,646,646,645,653,652,651,654,650,648,644,657,660,661,659,663,664,6
62,658,656,667,666,655,496,670,669,672,674,677,676,677,681,682,677,675,673,671,686,685,687,684,688,692,69
4,693,695,690,697,700,699,698,704,703,702,706,708,707,705,709,712,711,710,714,713,715,701,696,716,689,719,
718,721,722,720,717,724,724,726,727,723,731,731,730,736,737,735,734,738,732,729,728,740,744,746,745,749,7
51,752,750,755,754,753,758,757,759,756,762,763,761,764,760,748,766,767,768,770,772,771,770,769,765,775,77
7,780,780,780,782,781,778,776,784,773,787,786,785,743,789,793,792,794,795,796,799,797,791,790,801,800,803,
808,808,807,805,802,788,813,812,810,815,814,815,809,818,817,741,823,822,821,825,827,826,824,829,830,831,8
32,834,833,837,836,837,835,839,830,841,828,844,847,846,844,852,851,852,850,849,856,858,861,863,864,862,86
5,860,859,856,855,867,871,870,869,868,871,866,852,848,843,872,878,879,877,881,880,875,874,882,883,873,886,
888,890,891,892,889,887,885,895,894,897,896,899,903,903,905,909,907,906,902,901,911,915,917,916,914,913,9
20,921,922,919,918,910,898,893,884,925,924,927,929,928,931,932,930,926,923,934,935,941,940,939,938,944,94
3,947,946,949,948,945,950,952,951,942,937,955,958,959,961,960,956,963,962,964,969,968,966,970,973,972,977,
980,979,978,976,985,984,983,986,982,981,989,988,991,990,987,994,997,999,998,996,995,993,992,975,974,971,9
65,954,953,933,842,819,739,683,668,373,