# Homework Turnin

| | |
|---|---|
| **Name:** | Xuqing Wu |
| **Account:** | xw88 (xw88@uw.edu) |
| **Student ID:** | 1933202 |
| **Section:** | AD |
| **Course:** | CSE 143 20wi |
| **Assignment:** | a8 |
| **Receipt ID:** | dfdbdc5b208bd03b4f28933b9af28688 |

**Warning:** Your turnin is 2 days late. Assignment a8 was due Friday, March 13, 2020, 11:00 PM.

Turnin script completed with output:

Note: support/BitInputStream.java uses or overrides a deprecated API. Note: Recompile with -Xlint:deprecation for details.

## Turnin Successful!

The following file(s) were received:

### HuffmanNode.java (1236 bytes, sha256: 276c5e7314c01a2cb1a7303ab2d813a2)

```
1.  // Xuqing Wu
2.  // 3/15/2020
3.  // CSE143
4.  // TA: Eric Fan
5.  // Assignment #8 part 1
6.  //
7.  // Class HuffmanNode for storing a single node of a binary
8.  // tree of integers and can be compared to each other
9.
10. public class HuffmanNode implements Comparable<HuffmanNode> {
11.    public int data;
12.    // frequency
13.    public HuffmanNode zero;
14.    // left node
15.    public HuffmanNode one;
16.    // right node
17.    public int letter;
18.    // integer value of letter
19.
20.    // construct a node with given data
21.    public HuffmanNode(int data) {
22.       this(data, null, null, -1);
23.    }
24.
25.    // construct a node with given data and given letter
26.    public HuffmanNode(int data, int letter) {
27.       this(data, null, null, letter);
28.    }
29.
30.    // construct a node with given data, left subtree,
```

```
31.     // right subtree and given letter
32.     public HuffmanNode(int data, HuffmanNode zero,
33.        HuffmanNode one, int letter) {
34.        this.data = data;
35.        this.zero = zero;
36.        this.one = one;
37.        this.letter = letter;
38.     }
39.
40.     // allows to comapre two HuffmanNodes
41.     // lower frequencies are less than higher frequencies
42.     // if two frequencies are equal, they are considered equal
43.     public int compareTo(HuffmanNode other) {
44.        return this.data - other.data;
45.     }
46. }
```

## HuffmanTree.java (4413 bytes, sha256: bd7d8ad3cbea418dc45349d4775ad4e5)

```
1.  // Xuqing Wu
2.  // 3/15/2020
3.  // CSE143
4.  // TA: Eric Fan
5.  // Assignment #8 part 2
6.  //
7.  // Class HuffmanTree allows client to compress text files by using a coding
8.  // scheme called Huffman coding based on the frequency of characters.
9.  // Instead of using the usual seven or eight bits per character, Huffman's
10. // method uses only a few bits for characters that are used often, more bits
11. // for those that are rarely used
12.
13. import java.util.*;
14. import java.io.*;
15.
16. public class HuffmanTree {
17.    private HuffmanNode tree;  // overall root
18.
19.    // post: construct Huffman tree using the given array of frequencies where
20.    // count[i] is the number of occurrences of the character with integer
21.    // value i. The use of priority queue helps figure out sequence to add
22.    // nodes. Add a eof character which has value one higher than the value
23.    // of the highest character in the array passed at the end of queue
24.    // for convenience
25.    public HuffmanTree(int[] count) {
26.       Queue<HuffmanNode> sequence = new PriorityQueue<>();
27.       for(int i = 0; i < count.length; i++) {
28.          if(count[i] > 0) {
29.             HuffmanNode cur = new HuffmanNode(count[i], i);
30.             sequence.add(cur);
31.          }
32.       }
33.       int max = count.length;
34.       sequence.add(new HuffmanNode(1, max));
35.       while(sequence.size() > 1) {
36.          HuffmanNode first = sequence.remove();
37.          HuffmanNode second = sequence.remove();
38.          HuffmanNode total = new HuffmanNode(first.data + second.data,
39.             first, second, 0);
40.          sequence.add(total);
41.       }
42.       tree = sequence.remove();
43.    }
44.
45.    // post: write tree to the given output stream in standard format
46.    // standard format is a series of pairs of lines where the first line has
47.    // an integer representing the characters integer value and the second
48.    // line has the code to use for that character.
49.    public void write(PrintStream output) {
50.       write(output, tree, "");
51.    }
52.
53.    // post: private method of write to write tree to the given output
```

```
54.     // stream in standard format with given tree and frequency
55.     private void write(PrintStream output, HuffmanNode tree, String
56.        frequency) {
57.        if(tree.zero == null && tree.one == null) {
58.           output.println(tree.letter);
59.           output.println(frequency);
60.        }
61.        else {
62.           write(output, tree.zero, frequency + "0");
63.           write(output, tree.one, frequency + "1");
64.        }
65.     }
66.
67.     // post: reconstruct the tree from the given input file. The Scanner
68.     // contains a tree stored in standard format. Frequencies are irrelevant
69.     // so all of the frequencies are set to -1.
70.     public HuffmanTree(Scanner input) {
71.        tree = null;
72.        while(input.hasNextLine()) {
73.           int data = Integer.parseInt(input.nextLine());
74.           String code = input.nextLine();
75.           tree = constructorHelper(data, code, tree);
76.        }
77.     }
78.
79.     // post: private method of constructor to reconstruct the tree with given
80.     // tree, frequency and tree and return it.
81.     private HuffmanNode constructorHelper(int data, String code,
82.        HuffmanNode current) {
83.        if(code.length() == 0) {
84.           current = new HuffmanNode(0, data);
85.        }
86.        else {
87.           if(current == null) {
88.              current = new HuffmanNode(0);
89.           }
90.           if(code.charAt(0) == '0') {
91.              current.zero = constructorHelper(data, code.substring(1),
92.                 current.zero);
93.           }
94.           else {
95.              current.one = constructorHelper(data, code.substring(1),
96.                 current.one);
97.           }
98.        }
99.        return current;
100.    }
101.
102.    // post: read individual bits from the input stream and write
103.    // corresponding characters to the output. Stop reading when encounter
104.    // a character with value equal to the eof parameter. Assume that the
105.    // input stream contains a legal encoding of characters.
106.    public void decode(BitInputStream input, PrintStream output, int eof) {
107.       HuffmanNode current = tree;
108.       while(current.letter < eof) {
109.          if(current.zero == null && current.one == null) {
110.             output.write(current.letter);
111.             current = tree;
112.          }
113.          else {
114.             if(input.readBit() == 0) {
115.                current = current.zero;
116.             }
117.             else {
118.                current = current.one;
119.             }
120.          }
121.       }
122.    }
123. }
```