**[Assignment 5 - Adaptive Huffman](https://courseweb.pitt.edu/webapps/blackboard/execute/uploadAssignment?content_id=_16756809_1&course_id=_171948_1&assign_group_id=&mode=view)**

|  |  |
| --- | --- |
| Attached Files: | * [[File](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11002384_2/xid-11002384_2) class2.bin](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11002384_2/xid-11002384_2) (9 B) * [[File](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11006871_2/xid-11006871_2) merchant.bin](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11006871_2/xid-11006871_2) (261 B) * [[File](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11006872_2/xid-11006872_2) abs.bin](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11006872_2/xid-11006872_2) (15 B) * [[File](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11011044_2/xid-11011044_2) alpha.bin](https://courseweb.pitt.edu/bbcswebdav/pid-16756809-dt-content-rid-11011044_2/xid-11011044_2) (6 B) |

 In class, we studied the way the Adaptive Huffman Tree grows and changes shape after each byte (ASCII character) is read from the input file (this procedure is called update). Implement the following class  
  
public class AdaptiveHuffmanTree{  
     Node[] rlo;          //list of nodes in the tree in reverse-level order rlo[255], rlo[254], ...  
     Node NYT;            //special node representing unseen characters  
     Node[] characters;   //pointers to the nodes in the tree holding each character (if null then the character has not been seen)  
     int size;            //number of Nodes in the tree  
  
     public AdaptiveHuffmanTree(){  
          characters = new Node[256];  
          rlo = new Node[256];  
  
          for(int k=0; k<256; k++){  
              characters[k] = null;  
          }  
  
          NYT = new Node(0);  
          NYT.place = 255;               //NYT is going to be placed at the root of the tree  
          NYT.weight = 0;                //NYT has weight 0  
          NYT.label = '0';               //NYT is always a leftchild  
          NYT.leaf = true;               //NYT is always a leaf  
          rlo[NYT.place] = NYT;          //place NYT at the root of the tree  
          NYT.parent = null;             //the root of the tree has no parent  
          size = 1;  
     }  
  
   /\*  
    \* Update the adaptive Huffman tree after inserting character c. This is the UPDATE procedure discussed in class.  
    \*/  
   public void update(char c){  
   }  
  
  
   /\*  
    \* Swap Nodes u and v in the reverse-level ordered array. Note: Node u is at index u.place!  
    \*/  
   private void swap(Node u, Node v){  
   }  
  
   /\*  
    \* Return true if character c has been seen, otherwise, return false.  
    \*/  
   public boolean characterInTree(int  c){  
   }  
  
  
   /\*  
    \* Return true if the reverse-level order traversal is a monotonically decreasing sequence, otherwise, return false.  
    \*/  
   private boolean satisfiesSiblingProperty(){  
   }  
  
  
   /\*  
    \* Return the sequence of labels (characters) from the root to the NYT node.  
    \*/  
   public StringBuffer getCodeWordForNYT(){  
       return null;   //change to the correct return value.  
   }  
  
  
   /\*  
    \* Return the sequence of labels (characters from the root to the Node for character c.  
    \*/  
   public StringBuffer getCodeWordFor(char c){  
       return null;    //change to the correct return value.  
   }  
  
  
   /\*  
    \* return the reference to the root node.  
    \*/  
   public Node root(){  
      return null;   //change to the correct return value.  
   }  
  
  
   public int size(){  
      return size;  
   }  
  
  
   /\*  
    \* I've provided this to help debug your tree.  
    \*/  
   public String toString(){  
       String result = "[";  
  
       for(int k=255; k>=0 ; k--){  
           if(rlo[k] != null)  
                result += "(" + rlo[k].weight + "," + rlo[k].place + "," + rlo[k].character + ") ";  
       }  
  
       return result + "]\nsize = " + size;  
   }  
}  
  
  
   public class Node{  
      char character;   //character stored in this node  
      int weight;       //frequency of this character  
      int place;        //node number in reverse-level order  
      Node parent;      //reference to parent of this node  
      char label;       //edge-label associated with this character (leftchild = 0, rightchild = 1)  
      boolean leaf;     //Node is a leaf  
  
      public Node(int w, char c){  
         weight = w;  
         character = c;  
         leaf = true;  
      }  
  
  
      public Node(int w){  
         weight = w;  
      }  
  
  
      public String toString(){  
         return "" + character + " " + weight + " " + place + "\n" + parent;  
      }  
   }  
  
Feel free to add any methods in the above classes that are needed. After implementing this class, construct a main() to start with an empty AdaptiveHuffmanTree and read a byte from the file class2.bin and call update() on that byte until all bytes have been read. Now  display the tree using the toString() method I've provided. For example:  
  
bash-3.2$ java AdaptiveHuffmanTree < class2.bin  
[(8,255, ) (5,254, ) (3,253,a) (3,252, ) (2,251,d) (2,250,r) (1,249, ) (1,248,v) (0,247,) ]  
size = 9  
  
The output above corresponds to the tree (note: only interior nodes are numbered in the diagram)

               8(255)   
             /   \  
           **a** 3    5 (254)  
                 / \  
               **d** 2  3 (252)  
                   / \  
            (249) 1   **r** 2   
                 / \  
               **NYT**  **v** 1  
                0  
    
  
The next step is to implement a class called Compress.   
  
public class Compress{  
   public static void main(String[] args) throws IOException{       
         //your program goes here  
   }  
}  
  
  
Add code to the main() so that the command  
  
     $ java Compress < file | java HexDump  
  
   produces the compressed sequence of bytes to the command-window and sends the characters read and the compression     
   stats to a file calle***d statistics.txt***.  
     
   For example, compressing class2.bin (a file with 8 bytes = 64 bits)  
     
     $ java Compress < class2.bin | java HexDump     
     61 9c 86 40 ec fc  
     48 bits  
     
   where class2.bin contains  
     
     $ cat class2.bin  
     aardvadr  
       
     Note: the hex values and their bit representations of the transmitted bits:   
     61 = 0110 0001  
     9c = 1001 1100  
     86 = 1000 0110  
     40 = 0100 0000  
     ec = 1110 1100  
     fc = 1111 11..  
       
   Note: the same bits shown above and the characters they represent in the compressed file:  
     0110 0001  1   0  0111 0010  00 0110 0100 000 0111 0110 0 111 111  
    |----a----|-a-|NYT|----r----|NYT|----d----|NYT|----v----|a|-d-|-r-|  
      
   where ***statistics.txt*** contains  
     
     $ cat statistics.txt     
     aardvadr       
     bits read = 64  
     bits transmitted = 46  
     compression ratio = 28.1  
  
    For example, compressing abs.bin (a file with 14 bytes = 112 bits)  
  
    $ java Compress < abs.bin | java HexDump  
   61 ff cc 25  
   32 bits      
  
where ***statistics.txt*** contains  
  
    $ cat statistics.txt   
    aaaaaaaaaaabbb  
    bits read = 112  
    bits transmitted = 31  
    compression ratio = 72.3  
  
     
 For example, compressing alpha.bin (a file with 6 bytes = 48 bits)  
  
 $ java Compress < alpha.bin | java HexDump  
 61 31 0c 71 90 32 e3 30  
 64 bits  
  
where ***statistics.txt*** contains  
  
$ cat statistics.txt   
abcdef  
bits read = 48  
bits transmitted = 61  
compression ratio = -27.1  
  
Part 2 (extra credit and optional)  
Write the Decompress program to undo what the Compress program does. (20 points additional)