

Nov 17, 20 16:22

**HEncode.java**

Page 1/9

```

1  /*Alan Stoloff
2   * Data Structures
3   * Dr. Benjamin
4   *
5   *    class HEncode - A program to encode a file using Huffman Code Algorithm.
6   */
7
8  import java.io.*;
9  import java.lang.reflect.Array;
10 public class HEncode {
11
12     private Node root = null;                // Root of the Huffman Code Tree.
13
14     private Node[] leafPtr = new Node[256]; // Array of pointers to leaf nodes.
15                                           // used to traverse up the code
16                                           // tree during the encoding process.
17
18     private int[] freq = new int[256]; // Frequency of the bytes being encoded.
19                                           // used to build the initial trees
20                                           // when building the code tree.
21
22     private String inputFilename; // The name of the file to encode.
23
24                                           // pq is a priority queue of root nodes to trees. It
25                                           // is used during the building of the code tree to
26                                           // select the roots with minimum frequency count.
27
28     private PriorityQueue<Node> pq = new PriorityQueue<Node>();
29
30                                           // stk is a stack used to store the 0's and 1's
31                                           // of the code for a byte being encoded. As a byte
32                                           // is being encoded, we travel from a leaf node, up
33                                           // the parent pointers to the root, pushing 0's and 1's
34                                           // as appropriate for the encoding. When the root
35                                           // is reached, we pop the stack of "bits" into the
36                                           // output file.
37
38     private Stack<Integer> stk = new Stack<Integer>();
39
40     private BitWriter bitw; // Writes bits to the outputfile.
41
42     public final boolean DEBUG = true; // When true debugging info
43                                           // is displayed.
44
45
46

```

Nov 17, 20 16:22

**HEncode.java**

Page 2/9

```
46
47
48     public static void main(String[] args)
49         throws FileNotFoundException, IOException
50     {
51         if (args.length != 1) {
52             System.out.println("Incorrect program argument");
53             System.exit(0);
54         }
55
56         HEncode coder = new HEncode(args[0]); // Construct a Huffman Encoder
57
58         coder.getFrequencies(); // Get the frequencies of bytes in inputfile.
59         if (coder.DEBUG)
60             coder.showFreq(); // For debug - Let's see if we got the fregs.
61         coder.getLeafPtrs(); // Get initial trees used to build code tree.
62         if (coder.DEBUG) // For debug - print priority queue of roots.
63             coder.showPQ();
64         coder.buildTree(); // Build the code tree.
65         if (coder.DEBUG) // For debug - print the code tree.
66             coder.printTree();
67         coder.encodeFile(); // Read the inputfile a second time, encoding
68                             // the inputfile.
69     }
70
71     /*
72     *     Constructor - The argument is the name of the file to encode.
73     */
74
75     public HEncode(String inputFilename)
76     {
77         this.inputFilename = inputFilename;
78     }
79
80
```

Nov 17, 20 16:22

**HEncode.java**

Page 3/9

```

80
81  /*
82  *   getFrequencies() - Open the given file and determine the frequency
83  *                       with which each byte (character) occurs.
84  *
85  *   The frequencies are store in the array freq at the index
86  *   location corresponding to the byte value 0 to 255.
87  */
88
89  public void getFrequencies()
90  {
91      FileInputStream inF;    // File object to read from.
92      int nextByte;          // Next byte from the file.
93
94      // Initialize the frequencies
95
96      for (int i = 0; i < 256; i++)
97          freq[i] = 0;
98
99      try {
100         inF = new FileInputStream(inputFilename); // Open the input file.
101
102         do {
103             nextByte = inF.read(); // Read the next byte (-1 on EOF)
104             if (nextByte != -1) //
105                 freq[nextByte]++; // Increment frequency counter
106             } while (nextByte != -1); // for the byte.
107
108             inF.close(); // Close the file.
109         }
110         catch (FileNotFoundException e) {
111             System.out.printf("Error opening file %s\n", inputFilename);
112             System.exit(0);
113         }
114         catch (IOException e) {
115             System.out.printf("IOException reading from: %s\n", inputFilename);
116             System.exit(0);
117         }
118     }
119
120
121
122  /*
123  *   showFreq() - display the byte frequency array.
124  *
125  *   For debugging purposes we want to show the frequency with
126  *   which each byte (or character) occurs.
127  */
128
129  public void showFreq()
130  {
131      for (int i = 0; i < 256; i++) { // Only show the bytes
132          if (freq[i] != 0) // having non-zero frequency.
133              System.out.printf("byte: %3d char: %c freq: %d\n",
134                                i, (char) i, freq[i]);
135      }
136  }
137

```

Nov 17, 20 16:22

**HEncode.java**

Page 4/9

```

137
138  /*
139  *      getLeafPtrs() - Create root nodes containing the bytes.  These nodes
140  *                      are the roots of the initial trees used to build the
141  *                      code tree.  They will become the leaf nodes of the
142  *                      Huffman Code tree.
143  *
144  *                      Enter the nodes in the priority queue of roots.
145  *                      as each node is created.
146  *                      Nodes with smaller frequency have higher priority.
147  */
148
149  public void getLeafPtrs()
150  {
151      //need a pointer to priority Queue
152      for(int i=0;i<256;i++){
153          if(freq[i]!=0){
154              Node temp=new Node();
155              leafPtr[i]=temp;
156              temp.data=(byte)i;
157              temp.frequency=freq[i];
158              pq.enqueue(temp);
159          }
160      }
161  }
162
163  public void showPQ()
164  {
165      System.out.println(pq.toString());
166  }
167
168  /*
169  *      buildTree() - A function to build the Huffman Code Tree.
170  *                  We start with a priority queue of the leaf nodes.
171  */
172
173  public void buildTree()
174  {
175      Boolean keepGoing=true;
176      Node temp=pq.dequeue();
177      while(keepGoing){
178          if(pq.isEmpty()){
179              pq.enqueue(temp);
180              this.root=temp;
181              keepGoing=false;
182              break;
183          }
184          Node rootTemp=new Node();
185          Node rtemp=pq.dequeue();
186          rootTemp.rchild=rtemp;
187          rootTemp.lchild=temp;
188          temp.parent=rootTemp;
189          rtemp.parent=rootTemp;
190          rootTemp.frequency=rootTemp.rchild.frequency+rootTemp.lchild.frequency;
191          temp=rootTemp;
192      }
193  }
194
195  /*
196  *      encodeFile() - a function to encode a file - that is to
197  *                    create a compressed file using the Huffman
198  *                    algorithm. The function:
199  *      1. Creates a BitWriter to write the compressed file.
200  *      2. Uses the BitWriter to write the size of the original file.
201  *      3. Calls writeTree() to write the code tree.
202  *      4. Opens the original file.
203  *      5. Repeatedly reads a byte from the file and
204  *         and calls writeCode to write the coded bits
205  *         for the byte to the compressed file.

```

Nov 17, 20 16:22

**HEncode.java**

Page 5/9

```
206      * 6. Closes the BitWriter
207      */
208
209      public void encodeFile()
210      {
211          bitw=new BitWriter("book.txt.huf");
212          bitw.writeInt(root.frequency);
213          writeTree(root);
214          FileInputStream inF;
215          int nextByte;
216          try {
217              inF = new FileInputStream(inputFilename); // Open the input file.
218
219              do {
220                  nextByte = inF.read(); // Read the next byte (-1 on EOF)
221                  if (nextByte != -1) //
222                      writeCode((byte)nextByte); // Increment frequency counter
223              } while (nextByte != -1); // for the byte.
224
225              inF.close();
226              bitw.close(); // Close the file.
227          }
228          catch (FileNotFoundException e) {
229              System.out.printf("Error opening file %s\n", inputFilename);
230              System.exit(0);
231          }
232          catch (IOException e) {
233              System.out.printf("IOException reading from: %s\n", inputFilename);
234              System.exit(0);
235          }
236      }
237
```

Nov 17, 20 16:22

**HEncode.java**

Page 6/9

```

237
238
239  /*
240  *    writeCode() - A function to encode byte b.  The function
241  *                  uses b as an index into the array of pointers
242  *                  to leaf nodes of the Huffman Code Tree.  Once at the leaf,
243  *                  parent pointers are used to climb to the root, pushing
244  *                  0's and 1's on a stack according to the encoding.  Once
245  *                  at the root, the 0's and 1's on the stack are popped off
246  *                  and written to the outputfile using the bitWriter.
247  */
248
249 public void writeCode(byte b)
250 {
251     Node ptr=leafPtr[b];
252     while(ptr!=root){
253         Node ptrParent=ptr.parent;
254         if(ptrParent.rchild==ptr){
255             stk.push(1);
256         }
257         if(ptrParent.lchild==ptr){
258             stk.push(0);
259         }
260         ptr=ptr.parent;
261     }
262     while(!stk.isEmpty()){
263         bitw.writeBit(stk.pop());
264     }
265 }
266 /*
267 *    writeTree() - A recursive function to write the Huffman
268 *                  Code Tree to the output file.  The tree
269 *                  must be stored with the encoded file so that it can
270 *                  be used to decode the file.
271 */
272
273
274 public void writeTree(Node root)
275 {
276     if(root==null){
277         return;
278     }
279     else if(root.lchild==null && root.rchild==null){
280         bitw.writeBit(0);
281         bitw.writeByte(root.data);
282     }
283     else{
284         bitw.writeBit(1);
285         writeTree(root.lchild);
286         writeTree(root.rchild);
287     }
288 }
289
290

```

Nov 17, 20 16:22

**HEncode.java**

Page 7/9

```
290
291     /*
292     *   printTree() - Print the Huffman Code Tree to
293     *                   standard output.
294     */
295
296
297     public void printTree()
298     {
299         rPrintTree(root,0);
300     }
301
302     /*
303     *   rPrintTree() - the usual quick recursive method to print a tree.
304     */
305
306     public void rPrintTree(Node r, int level)
307     {
308
309         if (r == null)           // Empty tree.
310             return;
311
312         rPrintTree(r.rchild, level + 1);    // Print the right subtree.
313
314         for (int i = 0; i < level; i++)
315             System.out.print("      ");
316
317         if (r.data > (byte) 31)
318             System.out.printf("%c-%d\n", (char) r.data, r.frequency);
319         else
320             System.out.printf("%c-%d\n", '*', r.frequency);
321
322         rPrintTree(r.lchild, level + 1);
323     }
324
325
```

Nov 17, 20 16:22

**HEncode.java**

Page 8/9

```

325
326  /*
327  *    Node - an inner class to represent a node of
328  *          a Huffman Code Tree.
329  */
330
331  private class Node implements Comparable<Node>
332  {
333      byte data;           // A byte of data - stored in an Integer.
334      Node lchild;        // Left child pointer.
335      Node rchild;        // Right child pointer.
336      Node parent;        // Pointer to parent node.
337      Integer frequency;   // Frequency the data within
338                          // a file being encoded.
339
340      /*
341      *    Basic node constructor.
342      */
343
344      public Node()
345      {
346          data = 0;        // Each Huffman Code Tree node
347          lchild = null;   // contains data, pointers to
348          rchild = null;   // children and parent nodes
349          parent = null;   // plus a frequency count
350          frequency = 0;   // associated with the data.
351      }
352
353      /*
354      *    Constructor specifying all values
355      *    of the node instance variables.
356      */
357
358      public Node(byte data, Node lchild, Node rchild,
359                  Node parent, int frequency)
360      {
361          this.data = data;
362          this.lchild = lchild;
363          this.rchild = rchild;
364          this.parent = parent;
365          this.frequency = frequency;
366      }
367
368      /*
369      *    compareTo() - Compare two frequency values. We want Nodes
370      *                  with lower frequencies to have higher priority
371      *                  in the priority queue.
372      */
373
374      public int compareTo(Node other)
375      {
376          if(this.frequency>other.frequency){
377              return -1;
378          }
379          else if(this.frequency<other.frequency){
380              return 1;
381          }
382          else{
383              return 0;
384          }
385      }
386
387      public String toString()
388      {
389          char ch = (char) this.data;
390
391          String str = "byte: " + data + "   char: ";
392
393          if (data > (byte) 31)

```



Nov 17, 20 16:22

**HEncode.java**

Page 9/9

```
394         str = str + (char) data + " freq: " + frequency;
395     else
396         str = str + " " + " freq: " + frequency;
397
398     return str;
399 }
400 }
401 }
```