HW1 Guide

Contents

1 High-level factoring

1.1 `huff`

Compression proceeds in several high-level phases:

1. Count the number of occurrences of each byte in the input file, as well as the total length in bytes.
2. Make a forest of Huffman tree leaves, one for each byte value, and then construct a Huffman tree (see §3.1).
3. Walk the Huffman tree to build a table mapping bytes to their codewords (§2.2).

4. Write metadata to the output file: first the total length of the file, and then a serialization of the Huffman tree (§3.2).

5. Encode the input file to the output file using the codeword table (§3.3).

I suggest writing a function or method to encapsulate each phase.

The phases communicate with other phases via (potentially) well-defined data structures:

- Phase 1 produces the frequency table, a mapping from each byte value to a count. Since byte values range from 0 to 255, they work perfectly as array indices, so the frequency table is most easily represented as a 256-element array. (C/C++: `int[256]`; Java: `int[]`; Python: `list`; Ruby: `Array`.)

- Phase 2 makes a Huffman forest, which is also probably best represented as an array of, in this case, Huffman trees. Then it builds the Huffman tree, whose representation I will discuss in more detail below (§2.1).

- Phase 3 produces the codeword table, a mapping from bytes to codewords. As with the frequency table, a 256-element array is both the easiest and most efficient representation for the codeword table, but how to represent the codewords is a more difficult question (see §2.2).

(Phases 4 and 5 do not produce data structures.)

As an aid in debugging, it's often useful to have a way to view the intermediate data structures. I suggest writing simple routines for printing out each of them, which will allow you to inspect and see what might be wrong. For the two tables this is straightforward, but producing a readable textual representation of the Huffman tree is a bit harder (see §2.1.3).

1.2 `puff`

Decompression is significantly simpler than compression. The phases are:

1. Read the file length and the serialized Huffman tree from the compressed file (§3.2).

2. Decode the file using the Huffman tree, using the file length from the previous step to know when to stop (§3.4).

You can (and probably should) reuse your Huffman tree data structure for decompression. Note, however, that this tree will not have weights if you serialize it in the way I suggest (§3.2)—but that's okay, because weights are unnecessary for decoding.

2 Data structures

2.1 The Huffman tree

There are two kinds of tree nodes:

- A leaf, which has a weight and a byte value.
- A branch, which has a weight and references to two nodes, its left and right subtrees.

How you represent this will depend on what language you are using and what programming style you prefer. Either style will work in any of the languages that we're using, but some choices may be easier or smoother than others.

2.1.1 Procedural: C, C++[1], and maybe Python

A straightforward non-OO representation is to unify leaves and branches into a single structure (or class) having the fields of both: a weight, a byte value, and references (C/C++: pointers) to left and right child nodes. For leaves, the left and right children are empty (C: `NULL`; C++: `nullptr`; Python: `None`), and only the weight and byte value have meaning. For internal nodes, the byte value isn't used, so it can be zero, but both the left and right subchild will always be valid references/pointers. That is, because branch nodes always have both children—there are no nodes with a child on one side and a nothing on the other—we can tell a leaf from a branch by checking whether, say, the left child is present[2].

It is probably wise to define predicate functions to check whether a node is a leaf or a branch, rather than open-coding the check each time.

Memory allocation (C and C++). Linked data structures such as trees (almost always) need to be allocated in the heap rather than on the stack. In Python all values are allocated on the heap (more or less), but in C and C++ you must explicitly allocate each node using `malloc` (C) or `new` (C++), and then refer to the via pointers. That is, if you define a `struct` (C++: or `class`) `treenode`, then *every place* you mention that type in your code—every variable, every field, and every parameter—must have a pointer type (C: `struct treenode*`; C++: `treenode*`). In particular, you cannot correctly allocate a tree node by taking the address (`&` operator) of a local variable or function argument, since the memory used for locals and arguments is deallocated upon function return.

[1] While C++ has some object-oriented features, it isn't primarily an object-oriented language, and unless you're very familiar with virtual dispatch, you are probably better off using a procedural style for this program.

[2] I've seen code that tries to deal with the case that one child is null and the other isn't, which can't happen but makes the code a lot more complicated.

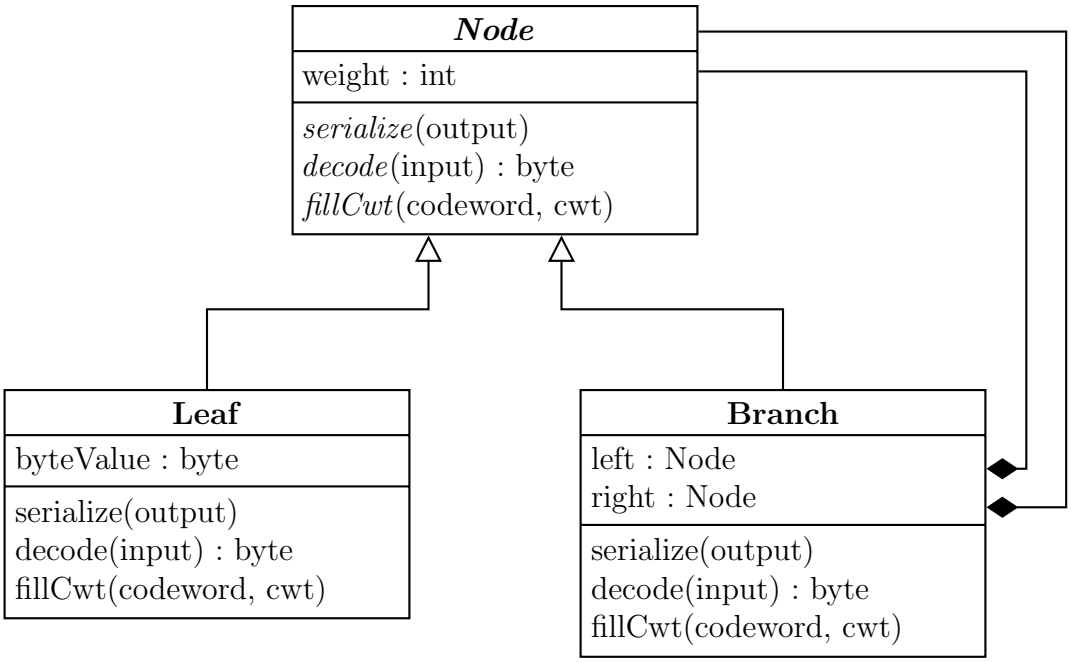

Figure 1: Class diagram for Huffman tree representation

You may have learned that every memory allocation must be paired with a deallocation (C: `free`; C++: `delete`). While this is good advice in general, you can avoid a whole host of bugs by breaking the rule in thise case. For long-running programs that allocate a lot, returning memory to the heap when you're done with it is important to avoid running out of memory. But when your program does at most 511 allocations (256 leaves and 255 branches) once, up front, running out of memory isn't an issue. Freeing memory introduces the risk of dangling pointer bugs, which cannot happen if you never free. One particular note of caution: C++ destructors that call `delete` are error-prone and difficult to debug, which is why they are seriously frowned upon in current C++ style. *Don't free—just let it be.*

2.1.2 OO: Java, Ruby, and maybe Python

The procedural approach from the previous section will work fine in Java and Ruby as well. But the Huffman tree also admits an elegant object oriented representation, if you'd prefer to go that way.

You would, of course, define a superclass `Node` that includes the commonalities, and two subclasses `Leaf` and `Branch` that include the differences. Furthermore, `Node` should define a common interface (Java: abstract methods; Ruby and Python: in your head/comments) for the operations that need to be performed on nodes. My design in a UML-like notation[3]

[3]In each box the second section lists the fields and the third lists the methods. Methods with names in

appears in figure 1.

Writing the operations as methods means that you avoid ever testing whether a node is a leaf or a branch—instead you just call the method and it does the right thing. For example, the suggested serialization strategy is to perform a pre-order traversal of the tree, handling leaves by outputting a 0 followed by the eight-bit byte value, and handling branches by outputting a 1 followed by recursivingly serializing the children. This can be done by having each class, Leaf and Branch, know how to perform its part of the job. In pseudocode:

Method Leaf::serialize(*out*)
  Write a 0 bit to *out*
  Write the 8 bits of *byteValue* to *out*[4]

Method Branch::serialize(*out*)
  Write a 1 bit to *out*
  left.serialize(*out*)
  right.serialize(*out*)

Yes, it really is that simple. You can write *decode* in similar fashion: when a leaf is asked to decode, it simply returns (or writes) its byte value; when a branch is asked to decode, it reads one bit and then recursively asks one of its child nodes to decode.

Similarly, *fillCwt* fills in the codeword table for all the leaf nodes below the current node, given a codeword that encodes the path to the current node in the tree, and the codeword table to fill in. See §2.2 for details on the algorithm and codeword data type.

2.1.3 Printing out the tree

If you want to bring out the Huffman tree for debugging, the easiest way is probably with a pre-order traversal. Here is pseudocode for the procedural version:

Function printHuff(*node*)
  If *node* is a leaf
    print("`Leaf[`", *node.weight*, "`, `", *node.byteValue*, "`]`")
  Else
    print("`Branch[`", *node.weight*, "`, `")
    printHuff(*node.left*)
    print("`, `")
    printHuff(*node.right*)
    print("`]`")

italic are *virtual*—in Java this means they are declared `abstract` and left undefined, whereas in Ruby and Python these means they aren't declared in that class at all but will be declared in its subclasses. Hollow arrows indicate inheritance: Leaf and Branch both subclass Node. Solid diamonds indicate ownership: a Branch owns two Nodes (its children).

Of course, the object-oriented version would split the cases between the Leaf and Branch classes.

2.2 Codewords

If we think of codewords as an abstract data type, they require three operations:

- Create a new, zero-length codeword.
- Append a bit (0 or 1) to the end of a codeword. (This can modify the codeword in place or return a new codeword—the latter will probably be less trouble.)
- Output a codeword to a bit IO stream.

Given an implementation of this ADT, building the codeword table is a straightforward tree walk. Here is pseudocode for a function that, given the root node of the Huffman tree, a zero-length codeword, and an empty codeword table, fills in the codeword values:

```
Function buildCwt(node, currentCodeword, table)
  If node is a leaf
    Store currentCodeword as the codeword for node.byte in table
  Else
    buildCwt(node.left, currentCodeword with 0 appended, table)
    buildCwt(node.right, currentCodeword with 1 appended, table)
```

The simplest representation of codewords is probably as a sequence (array, list, or vector) of boolean values[5]. Outputting this representation is easy—simply iterate over each bit and output it on the stream. One apparent downside is that this technique involves a lot of vector copying, but it isn't as bad as it looks, since 1) all the vectors are small and 2) you will allocate at most 511 of them if you traverse the Huffman tree just once to build the codeword table.

A perhaps more elegant solution is to represent a codeword as a pair (v, ℓ) of its value (if interpreted as a binary integer) and its length (in bits). For example, the codeword `01101` would be represented as the pair $(13, 5)$. Outputting this representation is easy because the bit IO library offers a *writeBits* operation that takes an integer and its length in bits. Given a codeword (v, ℓ), appending a 0 yields $(2v, \ell + 1)$, and appending a 1 yields $(2v + 1, \ell + 1)$.

[5] Or a string containing ASCII `'0'` and `'1'` characters, but please don't abuse strings that way.

3 Phases in detail

3.1 Constructing the Huffman tree

The Huffman tree construction algorithm is described in detail in the lecture notes. However, some lower-level suggestions may be helpful.

Once you have the frequency table, you should make a forest containing a leaf node for each byte having non-zero count. The easiest way to do this is probably to represent the forest as an array (I'll call it `forest`) with space 256 node references/pointers and an integer (`forestSize`) to keep track of how many actual nodes there are. (It may be worth writing a class to encapsulate these two things together.) Initially `forestSize` will be 0; then iterate over the frequency table, and for each non-zero byte, add a leaf node and increment `forestSize`.

The Huffman tree construction algorithm requires finding two minimal-weight nodes and combining them one tree as children of a new branch node. Code that attempts to find the two minimal-weight nodes directly is pretty complicated, but if you break the task down into these helper functions/methods then you can keep things simpler:

- Find the index of a minimal-weight node (by scanning the nodes).
- Remove a node given its index `i` (by assigning `forest[forestSize - 1]` to `forest[i]` and then decrementing `forestSize`.
- Add a node (by storing it at `forest[forestSize]` and then incrementing `forestSize`.

Using the first two operations to remove a minimal node, and then doing the same thing again, is much simpler than doing it in one pass. And then having removed them, the third operation can be used to add the new branch node back into the forest.

3.2 Serializing and deserializing the Huffman tree

Before serializing the tree, it's necessary to write the original file size to the encoded file, and before deserialization the tree it's necessary to read the original file size from the encoded file. But how should you store an integer in a file? There are several possibilities, but I suggest using the bit IO library's *writeBits* operation with the file size as the value and 32 as the length. (I promise not to test your code on anything large enough to overflow this.) Then it can be read using *readBits* and a length of 32.

Pseudocode for an object-oriented version of serialization appears in the section on OO representations of the Huffman tree (§2.1.2). Converting it to the procedural version is a matter of using a conditional statement to distinguish between leaf and branch cases.

Deserialization is serialization in reverse, and consequently the code has the same shape. In pseudocode:

```
Function deserialize(in)
  Read a bit from in
  If the bit is 0
    Read 8 bits from in
    Return a new leaf with the 8 bits as the byte value
  Else
    left  := deserialize(in)
    right := deserialize(in)
    Return a new branch with children left and right
```

In OO-land, unlike the other operations we've discussed, deserialization cannot be implemented as a method of Leaf and Branch, because no Leaf or Branch exists when we start deserializing. So even if you are using the OO approach, deserialization will probably look like the procedural code above (Java: static method; Python/Ruby: function).

3.3 Encoding

Once we're done counting, tree-building, and codeword-table-filling, encoding itself is anti-climactic: repeatedly read a byte from the input file, look it up in the codeword table, and output the corresponding codeward to the output file, until there are no more bytes to read.

But there's one catch: *eof* may not mean what you think it means. In particular, it typically does *not* return true right after the last byte has been read, even though no more bytes remain to be read. Instead, *eof* becomes true only after an attempt to read fails because it's reached the end of the file. Thus, in your encoding loop, the right time to check for end-of-file is not before reading a byte, but after. Once *eof* is indicated, break out of the loop without outputing the invalid byte that was just read.

3.4 Decoding

In order to avoid trouble with any padding bits added by the bit IO library, decoding should use a for-loop (or equivalent) that iterates exactly the correct number of times given the original file size read from the beginning of the encoded file. Then each iteration does one traversal from the root of the Huffman tree to some leaf, where the path is determined by reading bits from the decoder's input files, and then outputs the byte value of the leaf to the output file.