University of Cape Town
Department of Computer Science

CSC3022H/CSC3023F
Huffman Tree Encoding

March, 2019

You are asked to write a compression algorithm to compress English language text files. You decide to use a Huffman encoder, which is a fairly quick and easy way of turning a sequence of ASCII characters into a compressed bit stream. For this tutorial, you need only implement the compression side, and you do not need to pack the output into a stream of bytes. You can simply use the ASCII characters 0 and 1 to represent the compressed bit representation. Of course, your compression factor will be 8x less as a consequence! So for extra credit, you can do some bit operations to pack the 0s and 1s into an array of bytes.

1 Constraints

The constraints for this tutorial are as follows:

1. Your implementation must use at least a HuffmanTree class (which manages the tree and has methods to compress data, build a Huffman tree etc) and a HuffmanN-ode class, which represent the tree nodes in the underlying Huffman tree (which is a simple binary tree). These must be constructed and destructed appropriately. Your constructor and destructor will need to be defined to use the specified smart pointers to manage memory see point 2) below.

2. You must use `std::shared ptr<HuffmanNode>` to represent the left/right links for your tree nodes as well as the root of the tree in your HuffmanTree class.

3. your destructors must *not* explicitly delete the tree node representation: this should happen automatically when the root node is set to "nullptr" in the Huff-manTree destructor. You should not call delete at all in the HuffmanNode destruc-tor. That is the benefit of using a managed pointer. You may, however, allocated your HuffmanTree instance on the heap using new, in which case you would call delete to free that up and the end of your program.

4. You will need to implement: a default constructor, a move constructor, a copy constructor, a destructor, an assignment operator and a move assignment operator more details are provided below.

5. Your program will be driven from the command line, and should be invoked as follows: `huffencode <inputFile><output file>` where huffencode is the name

of your application, inputFile is an input English (ASCII) text file and output_file is the names of your output compressed "bitstream".

Note: if you do not do any bit packing, this file may be larger than the input (!). To compute the actual packed file size in bytes (approximately) you can compute: `(Nbits/8) + (Nbits%8 ? 1 : 0)` using integer arithmetic. This accommodates extra padding bits needed to deal with output bit stream sizes that are not a power of 2.

6. Prior to constructing the huffman encoding tree you have to count the number of occurrences of each letter in the file. To store these frequencies you can use a `std::unordered_map` with keys of type character and values of integer type (the unordered_map is very similar to Java's `HashMap`). When you build the Huffman tree, you will need this information to decide how to insert nodes for each letter. The relevant header file is <`unordered map`>. More information is available at `http://en.cppreference.com/w/cpp/container/unordered_map`.

7. You should use a `std::priority queue<HuffmanNode>` data structure to assist you in the building the tree. The relevant header file is <`queue`>
   A priority queue returns the next largest or smallest item, depending on how you have set it up. This is required to decide which tree nodes to aggregate as you build your tree. To ensure it works, you will need to provide a binary function predicate

```
bool compare(const HuffmanNode& a, const HuffmanNode& b)
{
  If (a < b) return true; // or > if the algorithm requires that ordering
  else return false;
}
```

   You will need to ensure the "<" operation is evaluated sensibly for the type HuffmanNode , as required by the tree construction algorithm. This function (which can be a free function) will be used by the priority queue to decide how to order its HuffmanNode elements.

   A priority queue supports a "push()" method (to insert elements), a "top()" method to copy the next element to be returned, and a "pop()" method to discard the next element to be returned. You can call "empty()" to see if the queue is empty or "size()" to see how many elements remain.

   To create an instance of the priority queue you would have code like the following: `std::priority_queue<HuffmanNode, compare>myQueue;` See `http://www.cplusplus.com/reference/queue/priority_queue/pop/` for a simple usage with plain old ints.

2 Huffman compression algorithm

You can build a Huffman tree as follows:

1. allocate a `shared_pointer<HuffmanNode>` for each letter, which contains the letter and frequency;

2. create a `std::priority_queue<HuffmanNode, compare>`, making sure that the right ordering (i.e. return smallest item) has been set in the compare function; Then, repeat until the queue has only one node, which will be the tree root:

3. choose the two nodes with the smallest letter frequency and create a new internal parent node which has these two nodes as its left and right children. The smaller nodes are inserted as the left child. These nodes should be popped out of the queue -you do not need to examine them again;

4. the new node (an internal node) has no letter associated with it, only a frequency. This frequency is the sum of the frequencies of its two children.

5. insert this new node in to the priority queue

At the end of this process you will have built your Huffman tree. If you have done this correctly the smart pointers will be sharing resources correctly and will auto destroy when the HuffmanTree goes out of scope (or is deleted) at the end of your program.

3 Code table

Next, you need to build the "code table": this is a mapping that gives a bit string code for each letter in your alphabet. The easiest way to do this (perhaps) is to recurse down the tree from the root building up a string as you go. When you hit a leaf node with a letter, you then output that letter with the corresponding bit string you have accumulated. When traversing the tree, add a "0" if branching left and add "1" if branching right. You can use an inorder walk and output when you get to a leaf node. It is up to you to decide how to store this code book. Ideally, finding symbol and its corresponding bit string should be fast.

4 Compression

To compress your ASCII text file, take each character, in turn, finding its bit string code and write this into the output buffer. This "buffer" can just be a `std::string` which you append characters/strings to. At the end, you can extract the `c_str()` (a pointer to a byte array) and write this out to file as a block of bytes. You should write out your code table to a separate file named "output file".hdr where output file is passed using command line argument. The header should have a field count at the top and a field for every character (consisting of the character and its code representation).

5 Mark allocation

Doing everything required above will get you 90%.

Extra credit:

- Convert to bit stream (5%): Convert your bits into an actual bit stream and store this in a byte array that is large enough; this should be written to disk as a binary file with a single int as header which lists the number of bits in the file. (you can work out number of bytes to read from this).

- Now write a method to read in and unpack the bitstream (5%). You should check that your unpacked bitstream matches the original encoded data.
- You may find the discussion at `http://www.cprogramming.com/tutorial/bitwise_operators.html` useful if you have never done this in Java. Take care to observe the behaviour of the shift operators for signed and unsigned integral types.

6 Unit tests

You must provide unit tests for each of the sections of your code. The unit tests should be specified in a separate file and compiled using a separate rule in your Makefile. Unit testing will require that you separate out the different steps into separate units of work, so that they can be tested individually. Therefore you should aim not write all of your code in one very long driver file for instance.

The unit tests will account up to 10% of your final mark (depending on whether each of the steps (frequency counting, tree construction, code table construction (tree traversal), encoding, etc.) is tested to satisfaction. The marks awarded for bit packing and unpacking includes the unit tests to show that these work as expected.

You should use the catch.hpp unit testing framework to test your code. A full tutorial is available at `https://github.com/philsquared/Catch/blob/master/docs/tutorial.md`.

7 Suggested work flow

- Week 1:
  - Define classes, default constructor/destructor
  - Build tree
  - Construct frequency table
  - Write unit tests
- Week 2:
  - Add in all extra constructors /move ops for tree and Node
  - create code book table
  - Encode string
  - Output bit string to file
  - Compute compression ratio (on "bit string" vs input chars)
  - *extra: pack bit string, unpack*
  - More unit tests

Please Note:

1. A working makefile must be submitted. If the tutor cannot compile your program in the lab by typing make, you will receive **50%** of your final mark.

2. You must use version control from the get-go. This means that there must be a .git folder alongside the code in your project folder. A **10%** penalty apply should you fail to include a local repository.

3. You must provide a README file explaining what each file submitted does and how it fits into the program as a whole. The README file should **not** explain any theory that you have used. These will be used by the tutors if they encounter any problems.

4. Do **not** hand in any binary files. Do **not** add binaries (.o files and your executable) to your local repository.

5. Please ensure that your tarball works and is not corrupt (you can check this by trying to extract the contents of your tarball - make this a habit!). Corrupt or non-working tarballs will not be marked - **no exceptions!**

6. A 10% penalty per day will be incurred for all late submissions. No hand-ins will be accepted if later than 5 days.

7. **DO NOT COPY. All code submitted must be your own.** ***Copying is punishable by 0 and can cause a blotch on your academic record.*** **Scripts will be used to check that code submitted is unique.**