# CSCI 230 — Data Structures and Algorithms
# Project 2 — File Compression, Inc.

February 12, 1999

**Preliminary Submission Deadline: Wednesday, Feb. 24 at 11:59:59 pm**
**Final Submission Deadline: Friday, March 5 at 11:59:59 pm**

Your successful completion of the "Meows and Mutts" project has resulted in a mild case of kennel cough and a referral to File Compression, Inc., a company specializing in software for compressing the size of files. Mr. Zvi Lepmel, Chief Technical Officer of FCI, has brought you in as a consultant to help with the implementation of a new product that performs a type of compression known as *Huffman coding*. (You can find some background information on this technique in your textbook in Section 10.1.2.)

Huffman coding requires the ability to manipulate binary trees, so the main deliverable for this project will be the implementation of a binary tree class, similar to the generic container classes found in the C++ Standard Library. You have already met with the FCI team and sketched out the following interface for the class:

```
template <class T>
class binary_tree
{
public:
  binary_tree();
  ~binary_tree();
  binary_tree(const T &);
  binary_tree(const binary_tree &);
  binary_tree & operator=(const binary_tree &);

  friend class iterator;
  class iterator
  {
  public:
    friend class binary_tree;
    iterator();
    ~iterator();
    T & operator*() const;
    bool operator!=(const iterator &) const;
    bool operator==(const iterator &) const;

    iterator left() const;            // go to left child node
    iterator right() const;           // go to right child node
    iterator parent() const;          // go to parent node
    bool is_root() const;             // true if at root
    bool is_leaf() const;             // true if at a leaf
  };

  iterator begin() const;
  iterator end() const;
  T & root();

  void combine(binary_tree & t1, binary_tree & t2);
};
```

In addition to the usual constructors and destructors, the binary tree class also contains an iterator class for traversing over the nodes in the tree. Binary tree iterators support some of the normal iterator operations, plus some special functions for traversing binary trees. Additional member functions of the binary tree class provide iterators to the "begin" (root) node and a special "end" node not in the tree, and fetch the value stored in the root node of the tree.

Here is an example of how the binary tree class and its iterator class is intended to be used:

```
// Print out contents of tree using an ''in-order'' traversal.
void inorder_traversal(const binary_tree<int>::iterator & i)
{
  if (i.is_leaf())
  {
    std::cout << *i << " ";
  }
  else
  {
    inorder_traversal(i.left());
    std::cout << *i << " ";
    inorder_traversal(i.right());
  }
}

void main()
{
   binary_tree<int> tree;

    ...   // put some stuff into the tree

   inorder_traversal(tree.begin());
}
```

One member function deserves special mention. Since a binary tree container will be made up of dynamically allocated nodes, it is possible to "combine" two binary trees as the children of the root node of a third binary tree in constant time (this is similar to how STL lists can be spliced together). A special function is provided in the header and should be implemented by you to do exactly this. The nodes in `t1` become the left subtree, and the nodes in `t2` become the right subtree of the third tree (represented by `*this`). The "root" tree must consist of only a single node, and both `t1` and `t2` are empty afterwards. This sounds complicated, but remember it will be done in constant time!

With the help of this binary tree class (and some useful standard library components), a class for performing Huffman encoding and decoding can be implemented. The interface for this class is quite simple it provides two member functions, one to encode and one to decode. Both use standard library streams for their input and output.

```
class huffman_codec
{
public:
  void encode(std::istream &, std::ostream &);
  void decode(std::istream &, std::ostream &);
};
```

## Encoding Algorithm

The Huffman encoding algorithm is as follows:

1. Construct the encoding tree:
   (a) Count the frequency of each byte value in the input (note that there are $2^8 = 256$ possible byte values).
   (b) Construct an initial binary tree for each byte value, weighted by its frequency.
   (c) Repeat until there is only a single tree remaining: combine the two trees with the smallest weights into a new tree, with weight equal to the sum of that of the two combined trees.
2. Construct the encoding table: for each byte value in the input, determine its code by tracing from the leaf node representing that value to the root of the tree. (Note that this will give you the code *backwards*, and you will need to reverse it.)
3. Write out the compressed data:
   (a) Write out the encoding tree (it is necessary to decode the file later). See below for a technique to do this.
   (b) Write out the count of characters in the input.
   (c) Write out the compressed data.

## Decoding Algorithm

To decode a file:

1. Read in and reconstruct the encoding tree (see below).
2. Read in the character count.
3. While there are characters left to decode:
   (a) Start at the root of the encoding tree.
   (b) While not at a leaf node, read a bit from the input and go to the left or right child as appropriate.
   (c) Output the character in the leaf node.

## Encoding Tree Output

Writing out the encoding tree is fairly straightforward. First, we can ignore the weight information, since it is only used in the initial construction of the tree from the frequency data. Also, we can take advantage of the fact that every node in the tree is either a leaf or has exactly two children.

The algorithm to write out the tree is recursive:

1. If the root node of the tree is itself a leaf:
   (a) Write out the byte value stored in the leaf, preceded by the character "." (period).
   (b) Return.
2. Otherwise:
   (a) Recursively write out the left subtree, surrounded by parentheses.
   (b) Recursively write out the right subtree, surrounded by parentheses.

## Encoding Tree Input

Reading in and reconstructing the tree is also recursive:

1. Read a character byte.
2. If the character is "." (period), then we have a leaf:
   (a) Read the next byte, and create a new tree with that value in its node.
   (b) Return the new tree.
3. Otherwise, the character must be a left parentheses, indicating we have two subtrees:
   (a) Recursively read in the left subtree.
   (b) Read in and skip the closing right parentheses and the next left parentheses.
   (c) Recursively read in the right subtree.
   (d) Read in and skip the closing right parentheses.
   (e) Create a new tree combining the left and right subtrees, and return it.

## Program synopsis and input format

The program will be run twice with the command-lines

```
huffman encode input-file compressed-file
huffman decode compressed-file output
```

The decoded output will be compared to (and should be identical with) the input file. You can use any file for sample input (even binary files).

## Notes, hints, and assumptions

- Engineers at FCI have already implemented a class for reading and writing individual bits, as well as a main program to test your classes. In addition they have completed most of the Huffman encoding and decoding routines. The code can be found in:

  Project 2 Auxiliary Files Directory[1]

  This directory is also accessible directly on the RCS file system

  ```
  /dept/cs/cs230/public/projects/project2
  ```

  Be sure to carefully look over the code and make sure you understand it.

  You will need to complete certain parts of the Huffman encoding/decoding algorithms in the file `huffman_codec.cpp`. Search for the string "COMPLETE" in the source code and follow the comments.

- The `bit_unpacker` class provides a function that returns individual bits from a stream:

  ```
  bool read(std::istream & in);
  ```

  The `bit_packer` class provides two functions, one that writes a string of bits to a stream, and a second that should be used after you have processed the last bit to ensure all bits are actually written (this is necessary since the system can only write bytes, or groups of 8 bits, at a time):

  ```
  void write(std::ostream &, const std::basic_string<bool> &);
  void flush(std::ostream &);
  ```

[1] http://www.rpi.edu/dept/cs/cs230/public/projects/project2/

- It is easiest to deal with strings of bits using the following type:

  ```
  std::basic_string<bool>
  ```

  This type works just like the `std::string` class you have used before (including working with standard library iterators and algorithms).

- You will need some way to keep track of the "forest" of binary trees you are combining to form the final encoding tree. To do this, use the `priority_queue` class in the standard library. Most of this has been done for you already by the FCI engineers.

- Remember that standard library containers copy the values that they hold. It will be very expensive to copy an entire binary tree each time it is added to or removed from a container, so store a pointer to the tree instead.

- The standard library component `pair` will be useful for holding two items (the weight and the byte value) in a binary tree node.

- What happens if the file being compressed is empty? What if it contains only a single byte (perhaps repeated over and over)? What should the compressed file look like for these cases? Does your final program handle them correctly?

- If you are using Microsoft Visual C++, you may get some warnings like the following:

  ```
  c:\program files\devstudio\vc\include\iosfwd(83) : warning
  C4804: '<' : unsafe use of type 'bool' in operation
  c:\program files\devstudio\vc\include\iosfwd(127) : warning
  C4305: 'return' : truncation from 'const int' to 'bool'
  ```

  These are in the standard library and can be safely ignored.

## Implementation environment

You may use either the Visual C++ environment or the GNU (g++ compiler) environment to implement your project. Please be very specific in indicating which one we should use in grading! Regardless of which you submit you **must submit a makefile**. You will need to write your own if you are working with the GNU compiler. Visual Studio will create one for you (look under the Project menu).

## Submission dates and guidelines

The final submission deadline is March 5 at 11:59:59 pm. Projects submitted within 24 hours after this deadline will have 10 points taken off. Projects submitted within 24-48 hours after this deadline will have 20 points taken off. Projects submitted later than this will NOT be accepted for credit.

A preliminary submission is required, however. By Wednesday, Feb. 25 at 11:59:59 pm, you must submit a brief outline of the design of the binary tree class and algorithm implementation for the project. This outline should indicate what private instance variables and member functions the class will have, as well as an outline of the implementation. Each submission will be briefly critiqued and returned (electronically). Students not submitting an outline or submitting an obviously thoughtless one by the indicated date will have a 10 point penalty taken from their grade. During final implementation of the project, you should feel free to revise the design (in part based on the feedback).

Detailed project submission guidelines will be posted on the course web site.

## Grading criteria

The program will be graded 25% for a clean compilation, 40% for correctness of the program, and 35% for program structure. Criteria for program correctness may be discerned from the foregoing description and will solely be based on the output from your program. Program structure includes class organization, member function design, code readability (indentation, variable names, etc.), and commentary. There should be comments describing the purpose of each class and the member functions and member variables. The comments about each class **must** also include a discussion of the flexibility of the class design and how well it will support future changes. Keep other comments, especially within the body of each function, short and concise. Avoid comments that just state in English what is obvious from the code. One bad example is

```
i ++;   //  increment i
```

## Academic integrity

Students may discuss the class and algorithm design. All code must be implemented by the student alone. Sharing of code among students is expressly forbidden and will be detected by code comparison tools. Projects from all six sections will be graded together by the TAs.