CSE 100:
STREAM I/O, BITWISE OPERATIONS,
BIT STREAM I/O
Today’s Class

- Build Huffman Tree
- C++ I/O
- I/O buffering
- Bit-by-bit I/O
Building Huffman Tree

Symbols:

A 20
C 8
S 10
V 2
W 13
Z 73
X 10
V 2
C 8
S 10
Building the tree: Huffman’s algorithm

0. Determine the count of each symbol in the input message.

1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.

2. Loop while there is more than 1 tree in the forest:
   2a. Remove the two lowest count trees
   2b. Combine these two trees into a new tree (summing their counts).
   2c. Insert this new tree in the forest, and go to 2.

3. Return the one tree in the forest as the Huffman code tree.

You know how to create a tree. But how do you maintain the forest?
Implement Huffman’s algorithm in C++

Sorted Linked list

\[
O(1) \quad O(N) \\
\text{forest} \quad \frac{O(k)}{O(k^2)} \quad \frac{O(k)}{O(k^2)} \\
\text{remove lowest count} \quad \frac{O(1)}{O(k^2)} \\
\text{insert} \quad \frac{O(k)}{O(k^2)} \\
\text{for each} \quad \frac{O(1)}{O(k^2)} \\
\text{= total} \quad \frac{O(N + k^2)}{O(N + k^2)}
\]
Implement Huffman’s algorithm in C++

Heap

K - Symbols
M - Size of text

0 - O(N)
1 - O(k log k) = O(k)

2 \log k

3 - O(M + k log k)

= O(M + k log k)

= O(\log N)
template<typename T> void print_queue(T& q) {
    while(!q.empty()) {
        std::cout << q.top() << " ";
        q.pop();
    }
    std::cout << '\n';
}

int main() {
    std::priority_queue<int> q;
    for(int n : {1, 8, 5, 6, 3, 4, 0, 9, 3, 2})
        q.push(n);
    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int> > q2;
    for(int n : {1, 8, 5, 6, 3, 4, 0, 9, 3, 2})
        q2.push(n);
    print_queue(q2);
}
#include HCNODE_HPP

/** Compare this HCNode and other for priority ordering.
  * Smaller count means higher priority.
  * Use node symbol for deterministic tiebreaking
  */

bool HCNode::operator<(HCNode const & other) const {
    // if counts are different, just compare counts
    if(count != other.count) return count > other.count;
    // counts are equal. use symbol value to break tie.
    // (for this to work, internal HCNodes
    // must have symb set.)
    return symb < other.symb;
}

#endif
Using < to compare nodes

• Consider this context:

```java
HCNode n1, n2, n3, n4;
n1.count = 100; n1.symbol = 'A';
n2.count = 200; n2.symbol = 'B';
n3.count = 100; n3.symbol = 'C';
n4.count = 100; n4.symbol = 'A';
```

• Now what is the value of these expressions?

- `n1 < n2`  
- `n2 < n1`  
- `n2 < n3`  
- `n1 < n3`  
- `n3 < n1`  
- `n1 < n4`

A. true
B. false
Using std::priority_queue in Huffman’s algorithm

- If you create an STL container such as priority_queue to hold HCNode objects:

```cpp
#include <queue>
std::priority_queue<HCNode> pq;
```

- ... then adding an HCNode object to the priority_queue:

```cpp
HCNode n;
pq.push(n);
```

- ... actually creates a copy of the HCNode, and adds the copy to the queue. You probably don’t want that. Instead, set up the container to hold pointers to HCNode objects:

```cpp
std::priority_queue<HCNode*> pq;
HCNode* p = new HCNode();
pq.push(p);
```
Using std::priority_queue in Huffman’s

Instead, set up the container to hold pointers to HCNode objects:

```cpp
std::priority_queue<HCNode*> pq;
HCNode* p = new HCNode();
pq.push(p);
```

What is the problem with the above approach?

A. Since the priority queue is storing copies of HCNode objects, we have a memory leak

B. The nodes in the priority queue cannot be correctly compared

C. Adds a copy of the pointer to the node into the priority queue

D. The node is created on the run time stack rather than the heap
std::priority_queue template arguments

- The template for priority_queue takes 3 arguments:
  - The first is the type of the elements contained in the queue.
  - If it is the only template argument used, the remaining 2 get their default values:
    - a `vector<T>` is used as the internal store for the queue,
    - `less` a class that provides priority comparisons
  - Okay to use vector container, but we want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply operator<
  - How to do that? We need to provide the priority_queue with a Compare class

```cpp
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
class priority_queue;
```
Defining a “comparison class”

- The documentation says of the third template argument:
- Compare: Comparison class: A class such that the expression \( \text{comp}(a,b) \), where \( \text{comp} \) is an object of this class and \( a \) and \( b \) are elements of the container, returns true if \( a \) is to be placed earlier than \( b \) in a strict weak ordering operation. This can be a class implementing a function call operator...

Here’s how to define a class implementing the function call operator \( \text{operator}() \) that performs the required comparison:

```cpp
class HCNodePtrComp {
    bool operator()(HCNode* & lhs, HCNode* & rhs) const {
        // dereference the pointers and use operator<
        return *lhs < *rhs;
    }
};
```

Now, create the priority_queue as:
```cpp
std::priority_queue<HCNode*, std::vector<HCNode*>, HCNodePtrComp> pq;
```
and priority comparisons will be done as appropriate.
One more piece

• When you create a new node in the forest, what character should it hold (beyond holding the sum of their children)?

A. The character ‘0’
B. The alphabetically smaller character of its children
C. The alphabetically larger character of its children
D. It doesn’t matter
PA3: encoding/decoding

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:
1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree

These require file I/O. We’ll talk about that today.
PA 3 Implementation strategy

- Implement Huffman tree `build()` method
  - `HCNode.cpp` and `HCTree.cpp`
- Write verification code to check that you can construct simple Huffman trees correctly
  - Use small inputs that you can verify by hand
  - Output codes as strings of 1s and 0s (char)
- Write the encoder method and driver program
  - Test with simple inputs that you can verify by hand and output the encoded input as character strings of 1s and 0s
- Add binary I/O
  - Write implementations of `BitInputStream` and `BitOutputStream` that write/read the compressed file as a text file (retain the ability to output in ASCII, it may come in handy)
- Compress/decompress a small file (100 bytes)
- Decompression should map the encoded input back to the original input
The `istream` class introduces member functions common to all input streams (that is, streams used for input into your program)

Some important ones are:

- **`istream& operator>>(type & val);`**
  - This is the stream extraction operator, overloaded for many primitive types `type`
  - Performs an input operation on an istream generally involving some sort of interpretation of the data (like translating a sequence of numerical characters to a value of a given numerical type)
  - Returns a reference to the istream, so extractions can be ‘chained’

```cpp
std::cin >> i >> j;
```

- **`int get();`**
  - Perform basic unformatted input. Extracts a single byte from the stream and returns its value (cast to an `int`)

```cpp
int k = cin.get();
```

- **`istream& read(char* s, streamsize n);`**
  - Perform unformatted input on a block of data. Reads a block of data of `n` bytes and stores it in the array pointed to by `s`

```cpp
char buff[40];
cin.read(buff,30);
```
C++ ostream

- The `ostream` class introduces member functions common to all output streams (streams used for output from your program).
- Some important ones are:

  `ostream & operator<<(type & val);`
  - This is the stream insertion operator. It is overloaded for many primitive types `type`. It performs an output operation on an ostream generally involving some formatting of the data (like for example writing a numerical value as a sequence of characters). It returns a reference to the ostream, so insertions can be ‘chained’.

  ```cpp
  std::cout << a << " and " << b << std::endl;
  ```

  `ostream & put(char c);`
  - Perform basic unformatted output. Writes a single byte to the stream and returns a reference to the stream

  `ostream & write(const char* s, streamsize n);`
  - Perform unformatted output on a block of data. Write a block of data of `n` bytes starting at address `s`.

  `ostream & flush();`
  - Any unwritten characters in the ostream’s buffer are written to its output destination as soon as possible ("flushed").

`cout` and `cerr` are instances of `ostream`
**C++ ifstream and ofstream**

- The **ifstream** class inherits from istream, and introduces functions specialized for doing input from files:

  ```cpp
  void open ( const char * filename,
              ios_base::openmode mode = ios_base::in );
  ```
  
  - Opens a file whose name is `filename`.

  ```cpp
  void close ( );
  ```
  
  - Closes the file associated with the stream. The stream is flushed first.

- The **ofstream** class inherits from ostream and introduces functions specialized for doing output to files:

  ```cpp
  void open ( const char * filename,
              ios_base::openmode mode = ios_base::out );
  ```
  
  - Opens a file whose name is `filename`.

  ```cpp
  void close ( );
  ```
  
  - Closes the file associated with the stream.
Reading bytes from a file

What should go in the blank so that we read a character at a time from a text file?

\[
\text{#include } \langle \text{iostream} \rangle \\
\text{#include } \langle \text{fstream} \rangle \\
\text{using namespace std; }
\]

\[
\text{int main( int argc, char** argv ) }
\{
\text{ifstream theFile;}
\text{char nextChar;}
\text{theFile.open( "testerFile.txt" );}
\text{while ( 1 ) }
\{
\text{nextChar = ___________________________;}
\text{if (theFile.eof()) break;}
\text{cout } \ll \text{ nextChar;}
\}
\text{theFile.close();}
\}
Writing bytes to a file

- In your Huffman code program you will write the encoded text from the infile to an outfile by writing out the code (a sequence of 0s and 1s) for each character in sequence.
- What is wrong with using with the following method for writing these codes to the file for actual compression?

```cpp
// assume that outStream is an ofstream, n is an HCNode
// and HCNode has a boolean field isZeroChild
...
if (n->isZeroChild) {
    outStream << '0';
} else {
    outStream << '1';
}
```

A. Nothing
B. You cannot use the `<<` operator to write to a file in C++
C. The ‘compressed’ file will be larger than the uncompressed file
D. The bits in the code will be written in the wrong order
Writing the compressed file

In the reference solution, the header is just 256 ints in a row, one for each byte value.

This header takes 1024 bytes of space. (4*256)
Reading and writing numbers

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile << num;
    numFile.close();
}
```

Assuming ints are represented with 4 bytes, how large is numfile after this program runs?
A. 1 byte
B. 4 bytes
C. 5 bytes
D. 20 bytes
Reading and writing numbers

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile << num;
    numFile.close();
}

You’ll need to include delimiters between the numbers
```
Writing raw numbers

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile.write( (char*)&num, sizeof(num) );
    numFile.close();
}

This is the method you'll use for the final submission
Let's look at the file after we run this code…
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "numfile" );
    numFile.write( (char*)&num, sizeof(num) ) ;
    numFile.close();

    // Getting the number back!
    ifstream numFileIn;
    numFileIn.open( "numfile" );
    int readN;
    numFileIn.read((char*)&readN, sizeof(readn));
    cout << readN << endl;
    numFileIn.close();
}
Opening a binary file

```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    unsigned char nextChar;
    theFile.open( "testerFile", ios::binary );
    while ( 1 ) {
        nextChar = theFile.get();
        if (theFile.eof()) break;
        cout << nextChar;
    }
    theFile.close();
}
```
Binary and nonbinary file streams

- Ultimately, all streams are sequences of bytes: input streams, output streams... text streams, multimedia streams, TCP/IP socket streams...

- However, for some purposes, on some operating systems, text files are handled differently from binary files
  - Line termination characters for a particular platform may be inserted or removed automatically
  - Conversion to or from a Unicode encoding scheme might be performed

- If you don’t want those extra manipulations to occur, use the flag `ios::binary` when you open it, to specify that the file stream is a binary stream
- To test your implementation on small strings, use formatted I/O

- Then add the binary I/O capability
  - But there is one small detail: binary I/O operates on units of information such as whole bytes, or a string of bytes
  - We need variable strings of bits
Reading binary data from a file: an example

```cpp
#include <fstream>
using namespace std;
/** Count and output the number of times char 'a' occurs in
 * a file named by the first command line argument. */
int main(int argc, char** argv) {
    ifstream in;
in.open(argv[1], ios::binary);
    int count = 0;
    unsigned char ch;
    while(1) {
        ch = in.get(); // or: in.read(&ch,1);
        if(! in.good() ) break; // failure, or eof
        if(ch == 'a') count++; // read an 'a', count it
    }
    if(! in.eof() ) { // loop stopped for some bad reason...
        cerr << "There was a problem, sorry." << endl;
        return -1;
    }
    cerr << "There were " << count << " 'a' chars." << endl;
    return 0;
}
```
Writing the compressed file

Now let's talk about how to write the bits…

Header (some way to reconstruct the HCTree)

Encoded data (bits)
Today’s Class

- C++ I/O
- I/O buffering
- Bit-by-bit I/O
Buffering

- The C++ I/O classes `ofstream`, `ifstream`, and `fstream` use buffering.

- I/O buffering is the use of an intermediate data structure, called the buffer, usually an array used with FIFO behavior, to hold data items.
  - Output buffering: the buffer holds items destined for output until there are enough of them to send to the destination; then they are sent in one large chunk.
  - Input buffering: the buffer holds items that have been received from the source in one large chunk, until the user needs them.

- The reason for buffering is that it is often much faster per byte to receive data from a source, or to send data to a destination, in large chunks, instead of one byte at a time.

- This is true, for example, of disk files and internet sockets; even small buffers (512 or 1K bytes), can make a big difference in performance.

- Also, operating system I/O calls and disk drives themselves typically perform buffering.
Streams and Buffers

BitOutputStream:

BitInputStream:

You can also manually flush this buffer
Buffering and bit-by-bit I/O

- The standard C++ I/O classes do not have any methods for doing I/O a *bit* at a time

- The smallest unit of input or output is one *byte* (8 bits)

- This is standard not only in C++, but in just about every other language in the world

- If you want to do bit-by-bit I/O, you need to write your own methods for it

- Basic idea: use a byte as an 8-bit buffer!
  - Use bitwise shift and or operators to write individual bits into the byte, or read individual bits from it;
  - flush the byte when it is full, or done with I/O

- For a nice object-oriented design, you can define a class that extends an existing iostream class, or that delegates to an object of an existing iostream class, and adds `writeBit` or `readBit` methods (and a `flush` method which flushes the 8-bit buffer)
Today’s Class

• C++ I/O
• I/O buffering
• Bit-by-bit I/O
C++ bitwise operators

- C++ has bitwise logical operators &, |, ^, ~ and shift operators <<, >>

- Operands to these operators can be of any integral type; the type of the result will be the same as the type of the left operand

  & does bitwise logical **and** of its arguments;

  | does logical bitwise **or** of its arguments;

  ^ does logical bitwise **xor** of its arguments;

  ~ does bitwise logical **complement** of its one argument

  << shifts its left argument left by number of bit positions given by its right argument, shifting in 0 on the right;

  >>> shifts its left argument right by number of bit positions given by its right argument, shifting in the sign bit on the left if the left argument is a signed type, else shifts in 0
C++ bitwise operators: examples

unsigned char a = 5, b = 67;

| a:   | 0 0 0 0 0 0 1 0 1 |
| b:   | 0 1 0 0 0 0 0 1 1 |

What is the result of a & b
A. 01000111
B. 00000001
C. 01000110
D. Something else
C++ bitwise operators: examples

unsigned char a = 5, b = 67;

\begin{align*}
a: & \quad 0000101 \\
b: & \quad 0100011
\end{align*}

What is the result of \( b \gg 5 \)?

A. 0000010
B. 0000011
C. 01100000
D. Something else
C++ bitwise operators: examples

unsigned char a = 5, b = 67;

a: \[00000010101\]
b: \[010000011\]

a \& b: \[0000000001\]
a | b: \[01000111\]
~a: \[11111110100\]
a << 4: \[01010000\]
b >> 1: \[001000001\]
(b >> 1) & 1: \[000000001\]
a | (1 << 5): \[00100101\]
C++ bitwise operators: an exercise

- Selecting a bit: Suppose we want to return the value --- 1 or 0 --- of the nth bit from the right of a byte argument, and return the result. How to do that?
  
  ```cpp
  byte bitVal(char b, int n) {
      return
  }
  ```

- Setting a bit: Suppose we want to set the value --- 1 or 0 --- of the nth bit from the right of a byte argument, leaving other bits unchanged, and return the result. How to do that?
  
  ```cpp
  byte setBit(char b, int bit, int n) {
      return
  }
  ```
Defining classes for bitwise I/O

• For a nice object-oriented design, let’s define a class `BitOutputStream` that delegates to an object of an existing iostream class, and that adds a `writeBit` method (and a `flush` method which flushes the 8-bit buffer)

• If instead `BitOutputStream` subclassed an existing class, it would inherit all the existing methods of its parent class, and so they become part of the subclass’s interface also
  • some of these methods might be useful, but...
  • in general it will complicate the interface

• Otherwise the two design approaches are very similar to implement, except that:
  • with inheritance, BitOutputStream uses superclass methods to perform operations
  • with delegation, BitOutputStream uses methods of a contained object to perform operations

• We will also consider a `BitInputStream` class, for bitwise input
#include <iostream>

class BitOutputStream {
private:
    char buf;     // one byte buffer of bits
    int nbits;    // how many bits have been written to buf
    std::ostream & out; // reference to the output stream to use

public:
    /** Initialize a BitOutputStream that will use
        * the given ostream for output. 
        */
    BitOutputStream(std::ostream & os) : out(os), buf(0), nbits(0) {
        // clear buffer and bit counter
    }

    /** Send the buffer to the output, and clear it */
    void flush()
    {
        os.put(buf);
        os.flush();
        buf = nbits = 0;
    }
}
Outline of a BitOutputStream class using delegation, cont

/** Write the least significant bit of the argument to
  * the bit buffer, and increment the bit buffer index.
  * But flush the buffer first, if it is full.
  */
void writeBit(int i)  {
    // Is the bit buffer full? Then flush it

    // Write the least significant bit of i into the buffer
    // at the current index

    // Increment the index
}
Outline of a BitInputStream class, using delegation

```cpp
#include <iostream>
class BitInputStream { 
private:
    char buf; // one byte buffer of bits
    int nbits; // how many bits have been read from buf
    std::istream & in; // the input stream to use

public:

    /** Initialize a BitInputStream that will use
     * the given istream for input.
     */
    BitInputStream(std::istream & is) : in(is) {
        buf = 0; // clear buffer
        nbits = ?? // initialize bit index
    }

    /** Fill the buffer from the input */
    void fill() {
        buf = in.get();
        nbits = 0;
    }

    What should we initialize nbits to?
    A. 0
    B. 1
    C. 7
    D. 8
    E. Other
```
/** Read the next bit from the bit buffer.
 * Fill the buffer from the input stream first if needed.
 * Return 1 if the bit read is 1;
 * return 0 if the bit read is 0.
 *
 */

int readBit() {
    // If all bits in the buffer are read, fill the buffer first

    // Get the bit at the appropriate location in the bit buffer, and return the appropriate int

    // Increment the index
}
Shannon’s entropy provides a lower bound on the average code length purely as a function of symbol frequencies and independent of ANY encoding scheme.

L_{ave} = 0.6 \times -\log(0.6) + 0.2 \times -\log(0.2) + 0.1 \times -\log(0.1) + 0.1 \times -\log(0.1)
= 0.6 \times \log(5/3) + 0.2 \times \log(5) + 0.1 \times \log(10) + 0.1 \times \log(10)
= 1.57
What is the best possible average length of a coded symbol with these frequencies?

A. 0  
B. 0.67  
C. 1.0  
D. 1.57  
E. 2.15

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>1.0</td>
</tr>
<tr>
<td>P</td>
<td>0.0</td>
</tr>
<tr>
<td>A</td>
<td>0.0</td>
</tr>
<tr>
<td>M</td>
<td>0.0</td>
</tr>
</tbody>
</table>
What is the best possible average length of a coded symbol with this frequency distribution? (why?)

A. 1  
B. 2  
C. 3  
D. $\log(2)$

Symbol Frequency  
S  0.25  
P  0.25  
A  0.25  
M  0.25

Calculate the entropy – what does it tell you? 
Calculate the average code length of Code C

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>110</td>
</tr>
<tr>
<td>m</td>
<td>111</td>
</tr>
</tbody>
</table>