CSE 100: HUFFMAN CODES AND C++ IO
Huffman’s algorithm: Building the Huffman Tree

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.
What is a good data structure to use to hold the forest of trees?
A. BST
B. Sorted array
C. Linked list
D. Something else
What is a good data structure to use to hold the forest of trees?
A. BST: Supports min, insert and delete in O(log N)
B. Sorted array: Not good for dynamic data
C. Linked list: If unordered then good for insert (constant time) but min would be O(N). If ordered then delete, min are constant time but insert would be O(N)
D. Something else: Heap (new data structure?)
What is a Heap?

Think of a Heap as a binary tree that is as complete as possible and satisfies the following property:

At every node x
  \[ \text{Key}[x] \leq \text{Key}[\text{children of } x] \]

So the root has the ______ value
## Heap vs. BST vs. Sorted Array

<table>
<thead>
<tr>
<th>Operations</th>
<th>BST (Balanced)</th>
<th>Sorted Array</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>O(log N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Min and Max</td>
<td>O(log N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Min or Max</td>
<td>O(log N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Predecessor/Successor</td>
<td>O(log N)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td>Rank</td>
<td>O(log N)</td>
<td>O(log N)</td>
<td></td>
</tr>
<tr>
<td>Output in sorted order</td>
<td>O(N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Insert</td>
<td>O(log N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>O(log N)</td>
<td>O(N)</td>
<td></td>
</tr>
<tr>
<td>Extract min or extract max</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ref: Tim Roughgarden (Stanford)
The suitability of Heap for our problem

- In the Huffman problem we are doing repeated inserts and extract-min!
- Perfect setting to use a Heap data structure.
- The C++ STL container class: `priority_queue` has a Heap implementation.
- Priority Queue and Heap are synonymous
Priority Queues in C++

A C++ `priority_queue` is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example `HCNode`'s, or pointers to `HCNode`'s, etc.

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

- You can extract object of highest priority in $O(\log N)$
Priority Queues in C++

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

- You can extract object of highest priority in \(O(\log N)\)
- To determine priority: objects in a priority queue must be comparable to each other
- By default, a `priority_queue<T>` uses `operator<` defined for objects of type `T`:
  - if `a < b`, `b` is taken to have higher priority than `a`
 ifndef HCNODE_HPP
#define HCNODE_HPP

class HCNode {

public:
  HCNode* parent; // pointer to parent; null if root
  HCNode* child0; // pointer to "0" child; null if leaf
  HCNode* child1; // pointer to "1" child; null if leaf
  unsigned char symb; // symbol
  int count; // count/frequency of symbols in subtree

  // for less-than comparisons between HCNodes
  bool operator<(HCNode const &) const;
};

#endif
In HCNode.cpp:

```cpp
#include HCNODE_HPP
/** Compare this HCNode and other for priority ordering.
 * Smaller count means higher priority.
 */
bool HCNode::operator<(HCNode const & other) const {
    // if counts are different, just compare counts
    return count > other.count;
}
#endif
```

What is wrong with this implementation?
A. Nothing
B. It is non-deterministic (in our algorithm)
C. It returns the opposite of the desired value for our purpose
In HCNode.cpp:

```cpp
#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;
}
```

```cpp
#endif
```
Using < to compare nodes

• Consider this context:

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

• Now what is the value of these expressions?

```java
n1 < n2
n2 < n1
n2 < n3
n1 < n3
n3 < n1
n1 < n4
```

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

• 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. The node is created on the run time stack rather than the heap
Using std::priority_queue in Huffman’s algo

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

What is the problem with the above approach?
- our operator< is a member function of the HCNode class. It is not defined for pointers to HCNodes. What to do?
std::priority_queue template arguments

- The template for priority_queue takes 3 arguments:

```cpp
1 template < class T, class Container = vector<T>,
2     class Compare = less<typename Container::value_type> > class priority_queue;
```

- 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 need to provide the priority_queue with a Compare class
Defining a "comparison class"

• The prototype for `priority_queue`:

```cpp
1 template < class T, class Container = vector<T>,
2     class Compare = less<typename Container::value_type> > class priority_queue;
```

• The documentation says of the third template argument
• Compare: Comparison class: A class such that the expression `comp(a,b)`, returns true if `a` is to be placed earlier than `b`, where `comp` is an object of this class and `a` and `b` are elements of the container. This can be a class implementing a function call operator... Called a “functor”

```cpp
std::priority_queue<HCNode*> pq;
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```
Defining a "comparison class"

• The prototype for priority_queue:

```cpp
template < class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> >
class priority_queue;
```

• Here’s how to define a class implementing the function call operator `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, we create the priority_queue, and priority comparisons will be done as needed

```cpp
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```

• Here is how a comparison will be done inside `priority_queue`

```cpp
HCNodePtrComp nodeComparator;
If (nodeComparator(pnode1, pnode2)) { .... }
```

• We have defined `operator <` on `HCNode`, to perform the comparison
Encoding a symbol

A very bad way is to start at the root and search down the tree until you find the symbol you are trying to encode.
A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code... or not?).
Traversing a list

class LNode {
    int data;
    LNode* next;
}

Assume you have created the following list:

void traverse(LNode* n) {
    while(n) {
        std::cout << n->data << std::endl;
        n = n->next;
    }
}

What does traverse(first) print?
A. 1 2 3
B. 3 2 1
C. Something else

```
first:  
    data: 1  
        next:  
    data: 2  
        next:  
    data: 3  
        next: /
```
Traversing a list, with recursion

class LNode {
    int data;
    LNode* next;
}

    first:
       data: 1
         next:
       data: 2
         next:
       data: 3
         next:

void traverse(LNode* n) {
    // 1
    if (n == 0) return;
    // 2
    traverse(n->next);
    // 3
}

Where should I put the line to print \texttt{n->data} to print the list in reverse order?
\texttt{std::cout \ll n->data \ll std::endl;}
A. 1  B. 2  C. 3
Encoding a symbol

A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code... or not?).

Use recursion to easily write a symbol’s code in the correct order!

```cpp
vector<HCNode*> leaves;
...
leaves = vector<HCNode*>(256, (HCNode*)0);
```
PA 2 Implementation strategy

• Implement Huffman tree \texttt{build()} method
  • \texttt{HCNode.cpp} and \texttt{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 (\texttt{char})
• Write the encode and decode method
  • Test with simple inputs that you can verify by hand and output the encoded input as character strings of 1s and 0s

CHECKPOINT HERE!

• Add binary I/O
  • Write implementations of \texttt{BitInputStream} and \texttt{BitOutputStream} that write/read the compressed file as a binary files
• Compress/decompress a small file (100 bytes)
• Decompression should map the encoded input back to the original input
Huffman: Encode & Decode File I/O

- Encode (compress)
- Decode (uncompress)
• **C++ I/O streams**  
  What is a stream?

• **I/O buffering**  
  Streams are essentially sequences of bytes of infinite length that are buffered.

• **Bit-by-bit I/O**
C++ istream

- The **istream** class introduces member functions common to all input streams (that is, streams used for input into your program)
- Some important ones are:

  ```
  cin is an instance of **istream**
  ```

  ```
  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’
    ```
    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**)
    ```
    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**
    ```
    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:

  ```cpp
  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;
  ```

  ```cpp
  ostream & put(char c);
  ```

  Perform basic unformatted output. Writes a single byte to the stream and returns a reference to the stream.

  ```cpp
  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`.

  ```cpp
  ostream & flush ( );
  ```

  Any unwritten characters in the ostream’s buffer are written to its output destination as soon as possible ("flushed").
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.
```cpp
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    string nextWord;
    theFile.open( "testerFile.txt" );
    while ( 1 ) {
        theFile >> nextWord;
        // Also if (!theFile.good()) break
        cout << nextWord << " ";
    }
    theFile.close();
}
```

Identify the C++ operator that is allowing us to read from the file!
# Reading from a file

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

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  string nextWord;
  theFile.open( "testerFile.txt" );
  while ( 1 ) {
    theFile >> nextWord;
    if (theFile.eof()) break;  // Also if (!theFile.good()) break
    cout << nextWord << " ";
  }
  theFile.close();
}
```

Notice that this code will strip formatting and read whole strings! (Not what you should do for your internal checkpoint…)

Identify the C++ operator that is allowing us to read from the file!
# Reading bytes from a file

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

using namespace std;

int main( int argc, char** argv )
{
    ifstream theFile;
    char nextChar;
    theFile.open( "testerFile.txt" );
    while ( 1 )
    {
        nextChar = ___________________________;
        if (theFile.eof()) break;
        cout << nextChar;
    }
    theFile.close();
}
```

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

A. `theFile.get();`
B. `(char)theFile.get();`
C. `(int)theFile.get();`
D. All of the above

What is the difference between this approach and using the `>>` operator?

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

```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
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?

```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

But this is exactly what you will do for the internal Checkpoint deadline! (We’ll talk about how to write one bit at a time later)
Reading and writing numbers

#include <iostream>
#include <fstream>

using namespace std;

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

What is in number.txt after this program is executed?
Is the binary data written into the file same as that representing the number 12345?
Writing raw numbers

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

using namespace std;

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

This is the method you’ll use for the final submission (and the checkpoint too if you want)

Recap: Different ways of writing to file
# Reading raw numbers

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

using namespace std;

int main( int argc, char** argv )
{
    ofstream numFile;
    int num = 12345;
    numFile.open( "number.txt" );  // .txt isn't really correct anymore...
    numFile.write( (char*)&n, sizeof(n) );
    numFile.close();
    // Getting the number back!
    ifstream numFileIn;
    numFileIn.open( "number.txt" );
    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
• 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:
- encoder
  - Buffer
  - ostream
  - bits
  - bytes

BitInputStream:
- istream
  - Buffer
  - bytes
  - bits
  - decoder
  - DATA OUT

- ostream
  - Buffer
  - disk
  - bytes
  - 4KB

- disk
  - Buffer
  - istream
  - bytes
  - 4KB

DATA IN

Scott B. Baden / CSE 100-A / Spring 2013
Buffers, Latency and Throughput

- **Latency**: the time between a new input and the corresponding output.
  - Lower is better.
- **Throughput**: the number of input bytes processed per second.
  - Higher is better.
- Buffers hurt (increase) latency but improve (increase) throughput.
  - Consider traveling SD->SF using a train vs. a private car.
  - Consider express lane vs. regular lane in the super-market.
Why Buffer?

Q: Why should we implement buffering in our BitOutputStream and BitInputStream class for the Huffman coding problem?

A. To improve latency
B. To improve throughput
C. To use the standard file IO stream interfaces provided by C++
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)
• 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: \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0 \\
1 \\
0 \\
1
\end{array}

b: \begin{array}{c}
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
1 \\
1
\end{array}

a & b \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{array}

a | b \begin{array}{c}
0 \\
1 \\
0 \\
0 \\
0 \\
1 \\
1 \\
1
\end{array}

~a \begin{array}{c}
1 \\
1 \\
1 \\
1 \\
1 \\
0 \\
1 \\
0
\end{array}

a << 4 \begin{array}{c}
0 \\
1 \\
0 \\
1 \\
0 \\
0 \\
0 \\
0
\end{array}

a >> 1 \begin{array}{c}
0 \\
0 \\
1 \\
0 \\
0 \\
0 \\
0 \\
1
\end{array}

(b >> 1) & 1 \begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
1
\end{array}

a | (1 << 5) \begin{array}{c}
0 \\
0 \\
1 \\
0 \\
0 \\
1 \\
0 \\
1
\end{array}
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
# Outline of a BitOutputStream class using delegation

```cpp
#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’d)

```java
/** 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 = 0; // initialize bit index
        }

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

};
```
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
Outline of a BitInputStream class, using delegation

/** 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
}