CSE 143
Lecture 23

Priority Queues and Huffman Encoding

slides created by Daniel Otero and Marty Stepp
http://www.cs.washington.edu/143/

Assignment #8

- You're going to make a Winzip clone except
  - without a GUI (graphical user interface)
  - it only works with a weird proprietary format (not ".zip")

- Your program should be able to compress/decompress files
  - "Compression" refers to size (bytes); compressed files are smaller

Why use compression?

- Reduce the cost of storing a file
  - …but isn't disk space cheap?
- Compression applies to many more things:
  - Store all personal photos without exhausting disk
  - Reduce the size of an e-mail attachment to meet size limit
  - Make web pages and images smaller so they load fast
  - Reduce raw media to reasonable sizes (MP3, DivX, FLAC, etc.)
  - …and on…
- Don't want to use your 8th assignment? Real-world apps:
  - Winzip or WinRAR for Windows
  - StuffitExpander for Mac
  - Linux guys…you know what to do

What you'll need

- A new data structure: the Priority Queue.
  - so it's, like, a queue…but with, like…priorities?

- A sweet new algorithm: Huffman Encoding
  - Makes a file more space-efficient by
    - Using less bits to encode common characters
    - Using more bits to encode rarer characters
  - But how do we know which characters are common/rare?

Problems we can't solve (yet)

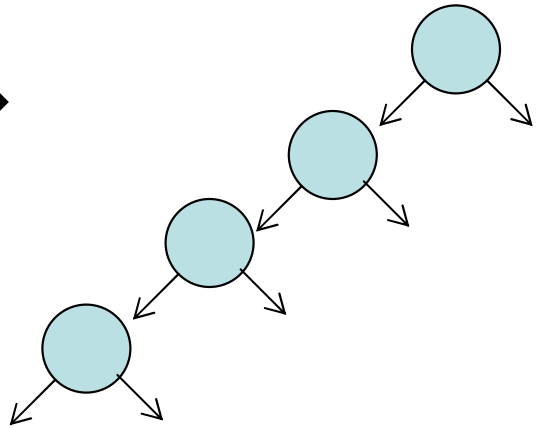
- The CSE lab printers constantly accept and complete jobs from all over the building. Suppose we want them to print faculty jobs before student jobs, and grad before undergrad?

- You are in charge of scheduling patients for treatment in the ER. A gunshot victim should probably get treatment sooner than that one guy with a sore shoulder, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?

- Why can't we solve these problems efficiently with the data structures we have (list, sorted list, map, set, BST, etc.)?

Some bad "fixes" (opt.)

- *list*: store all customers/jobs in an unordered list, remove min/max one by searching for it
  - problem: expensive to search
- *sorted list*: store all in a sorted list, then search it in O(log *n*) time with binary search
  - problem: expensive to add/remove
- *binary search tree*: store in a BST, search it in O(log *n*) time for the min (leftmost) element
  - problem: tree could be unbalanced ➔



- *auto-balancing BST*
  - problem: extra work must be done to constantly re-balance the tree

Priority queue

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
  - a mix between a queue and a BST
  - usually implemented using a tree structure called a *heap*

- priority queue operations:
  - `add` adds in order; O(1) average, O(log *n*) worst
  - `peek` returns **minimum** element; O(1)
  - `remove` removes/returns **minimum** element; O(log *n*) worst
  - `isEmpty`, `clear`, `size`, `iterator` O(1)

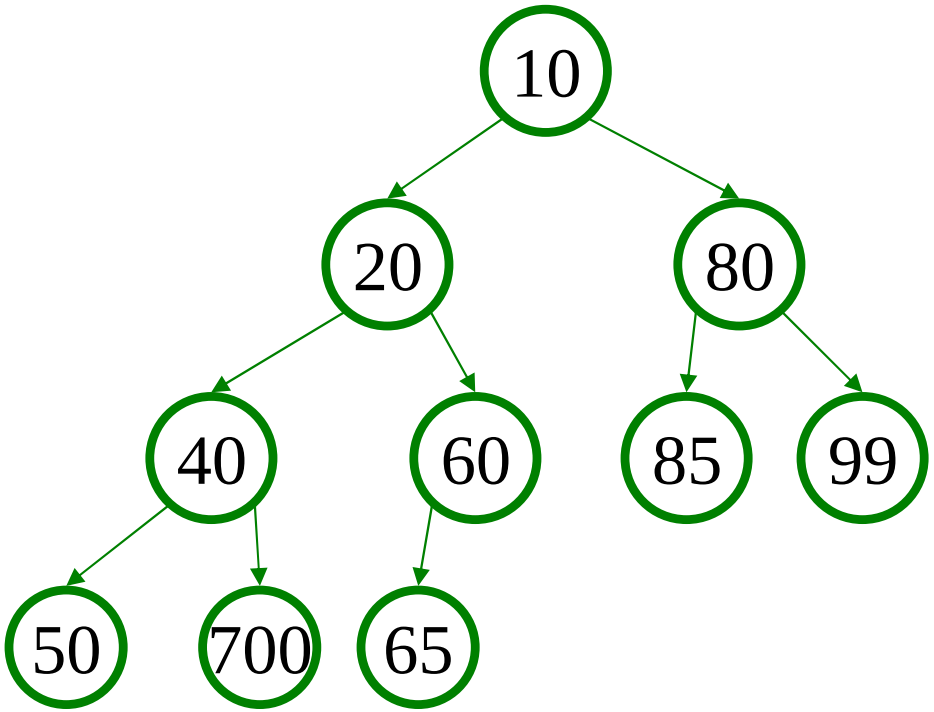
Java's `PriorityQueue` class

```
public class PriorityQueue<E> implements Queue<E>
```

Method/Constructor	Description
`public PriorityQueue<E>()`	constructs new empty queue
`public void add(E value)`	adds given value in sorted order
`public void clear()`	removes all elements
`public Iterator<E> iterator()`	returns iterator over elements
`public E peek()`	returns minimum element
`public E remove()`	removes/returns minimum element

Inside a priority queue

- Usually implemented as a "heap": a sort of tree.
- Instead of being sorted left->right, it's sorted up->down
  - Only guarantee: children are lower-priority than ancestors



Exercise: Firing Squad

- Marty has decided that TA performance is unacceptably low.
- We are given the task of firing all TAs with < 2 qtrs

- Write a class `FiringSquad`. Its `main` method should read a list of TAs from a file, find all with sub-par experience, and replace them. Print the final list of TAs to the console.
- Input format:

  taName numQuarters
  taName numQuarters
  taName numQuarters
  `… etc.`

NOTE: No guarantees about input order

The caveat: ordering

- For a priority queue to work, elements must have an ordering
- In Java, this means using the `Comparable<E>` interface

- Reminder:

```
public class Foo implements Comparable<Foo> {
  …
  public int compareTo(Foo other) {
    // Return positive, zero, or negative number if this object
    // is bigger, equal, or smaller than other, respectively.
    …
  }
}
```

Let's fix it…

ASCII

- At the machine level, everything is binary (1s and 0s)
- Somehow, we must "encode" all other data as binary
- One of the most common character encodings is ASCII
  - Maps every possible character to a number (`'A'` → 65)
- ASCII uses one *byte* (or eight *bits*) for each character:

Char	ASCII value	ASCII (binary)
`' '`	32	00100000
`'a'`	97	01100001
`'b'`	98	01100010
`'c'`	99	01100011
`'e'`	101	01100101
`'z'`	122	01111010

For fun and profit: http://www.asciitable.com/

Huffman Encoding

- ASCII is fine in general case, but we know letter frequencies.
- Common characters account for more of a file's size, rare characters for less.
- Idea: use fewer bits for high-frequency characters.

Char	ASCII value	ASCII (binary)	Hypothetical Huffman
' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	01110100
'c'	99	01100011	001100
'e'	101	01100101	1100
'z'	122	01111010	00100011110

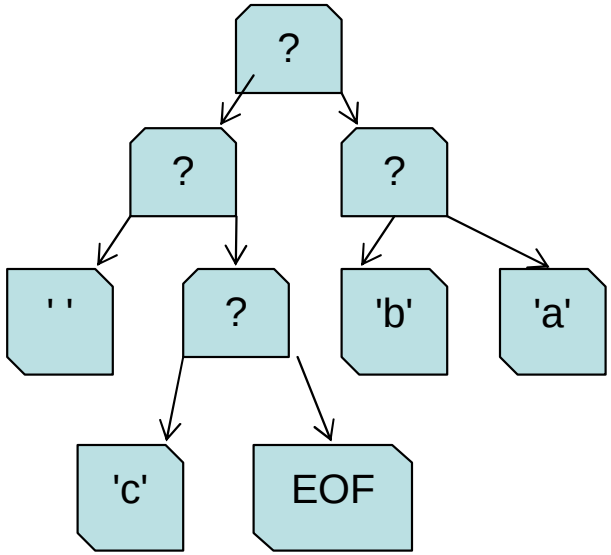
Compressing a file

To compress a file, we follow these steps:

- Count occurrences of each character in the file
  - Using: ?
- Place each character into priority queue using frequency comparison
  - Using: a priority queue
- Convert priority queue to another binary tree via mystery algorithm X
  - Using: binary tree
- Traverse the tree to generate binary encodings of each character
  - Using: ?
- Iterate over the source file again, outputting one of our binary encodings for each character we find.

"Mystery Algorithm X"

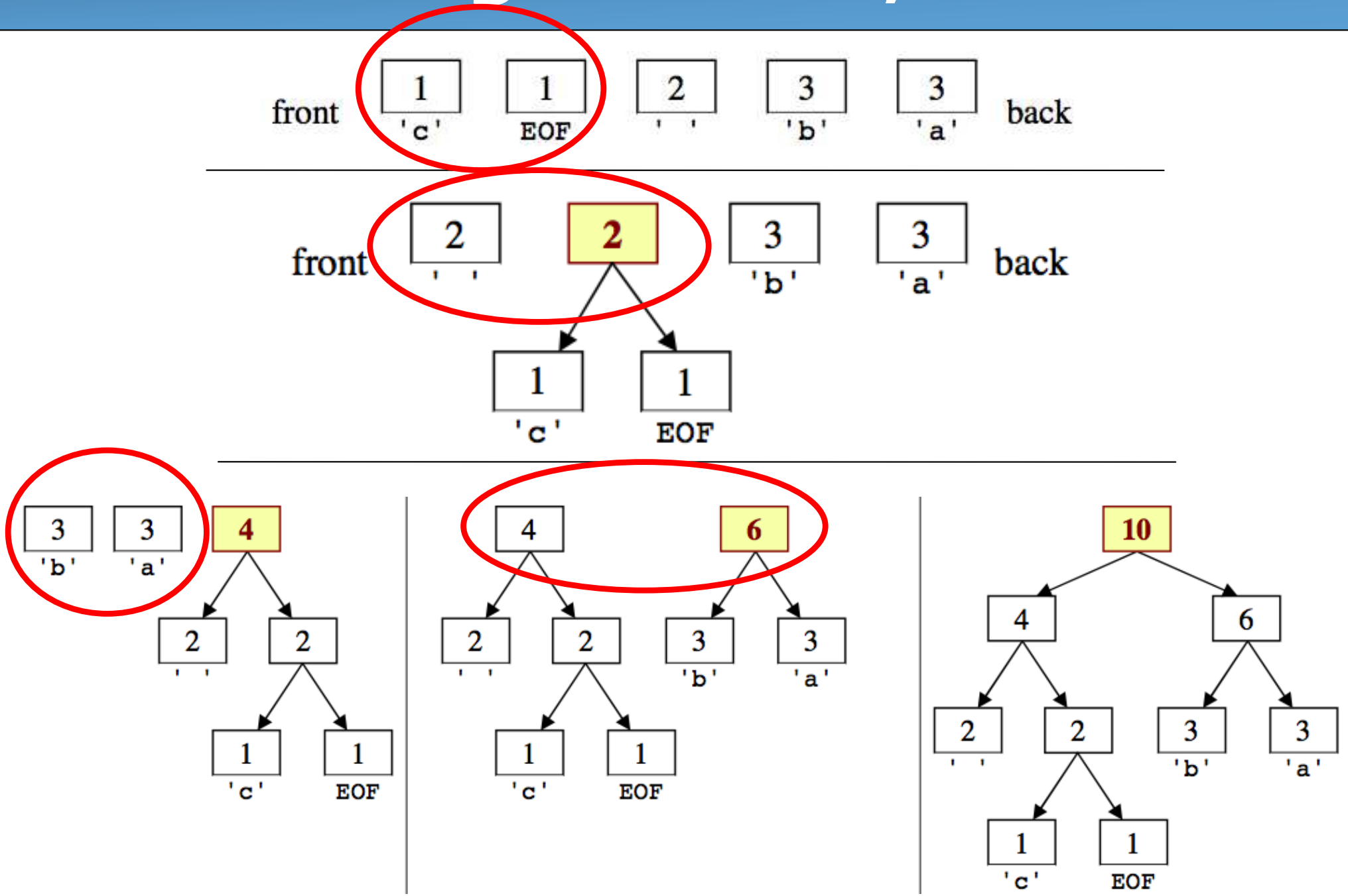
- The secret:
  - We'll build a tree with common chars on top
  - It takes fewer links to get to a common char
  - If we represent each link (left or right) with one bit (0 or 1), we automagically use fewer bits for common characters
- Tree for the example file containing text "ab ab cab":



Building the Huffman tree

- Create a binary tree node for each character containing:
  - The character
  - # occurences of that character
- Shove them all into a priority queue.
- While the queue has more than one element:
  - Remove the two smallest nodes from the priority queue
  - Join them together by making them children of a new node
  - Set the new node's frequency as the sum of the children
  - Reinsert the new node into the priority queue
- Observation: each iteration reduces the size of the queue by 1.

Building the tree, cont'd



HuffmanTree: Part I

- Class for HW#8 is called `HuffmanTree`
  - Does both compression and decompression
- Compression portion:

- **`public HuffmanTree(Map<Character, Integer> counts)`**
  - Given a Map containing counts per character in an file, create its Huffman tree.

- **`public Map<Character, String> createEncodings()`**
  - Traverse your Huffman tree and produce a mapping from each character in the tree to its encoded binary representation as a String. For the previous example, the map is the following: `{' '=010, 'a'=11, 'b'=00, 'd'=011, 'n'=10}`

- **`public void compress(InputStream in, BitOutputStream out) throws IOException`**
  - Read the text data from the given input file stream and use your Huffman encodings to write a Huffman-compressed version of this data to the given output file stream

Bit Input/Output Streams

- Filesystems have a lowest size denomination of 1 byte.
  - We want to read/write one *bit* at a time (1/8th of a byte)
- `BitInputStream`: like any other stream, but allows you to read one bit at a time from input until it is exhausted.
- `BitOutputStream`: same, but allows you to *write* one bit at a time.

`public BitInputStream(InputStream in)`	Creates stream to read bits from given input
`public int readBit()`	Reads a single 1 or 0; returns -1 at end of file
`public boolean hasNextBit()`	Returns `true` iff another bit can be read
`public void close()`	Stops reading from the stream

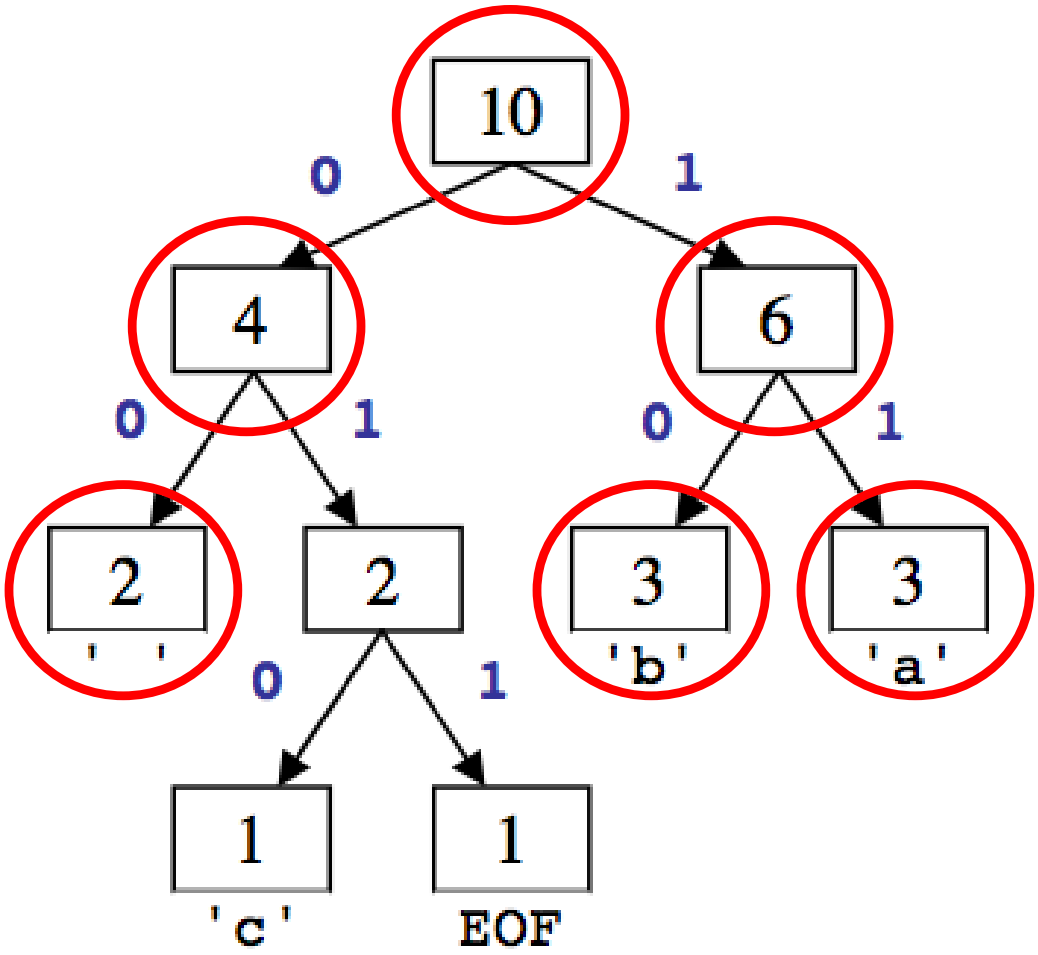
`public BitOutputStream(OutputStream out)`	Creates stream to write bits to given output
`public void writeBit(int bit)`	Writes a single bit
`public void writeBits(String bits)`	Treats each character of the given string as a bit (`'0'` or `'1'`) and writes each of those bits to the output
`public void close()`	Stops reading from the stream

HuffmanTree: Part II

- Given a bunch of bits, how do we decompress them?
- Hint: HuffmanTrees have an encoding "prefix property."
  - No encoding A is the prefix of another encoding B
  - I.e. never will $x \rightarrow$ `011` and $y \rightarrow$ **`011`**`100110` be true for any two characters x and y
- Tree structure tells how many bits represent "next" character
- While there are more bits in the input stream:
  - Read a bit
  - If zero, go left in the tree; if one, go right
  - If at a leaf node, output the character at that leaf and go back to the tree root

HuffmanTree: Part II cont'd.

HuffmanTree for "ab ab cab"



Sample encoding

111000…

→ "ab "

HuffmanTree: Part II cont'd.

- The decompression functionality of `HuffmanTree` is handled by a single method:

- **`public void decompress(BitInputStream in, OutputStream out) throws IOException`**
  - Read the compressed binary data from the given input file stream and use your Huffman tree to write a decompressed text version of this data to the given output file stream.
  - You may assume that all characters in the input file were represented in the map of counts passed to your tree's constructor.

EOF?

- When reading from files, end is marked by special character: EOF ("End Of File")
  - NOT an ASCII character
  - Special code used by each particular OS / language / runtime

- Do you need to worry about it?
  - No, it doesn't affect you at all.
  - You may however notice it in your character maps, so don't get confused or worried.
  - FYI: EOF prints as a ? on the console or in jGRASP. (binary 256)

Checked Exceptions

- **Unchecked exceptions** can occur without being explicitly handled in your code
- Any subclass of `RuntimeException` or `Error` is unchecked:
  - `IllegalArgumentException`
  - `IllegalStateException`
  - `NoSuchElementException`

- **Checked exceptions** must be handled explicitly
- Checked exceptions are considered more dangerous/important:
  - `FileNotFoundException`
  - Its parent, `IOException`

The `throws` clause

- What does the following mean:

```
public int foo() throws FileNotFoundException { …
```

- ***Not*** a replacement for commenting your exceptions
- A `throws` clause makes clear a checked exception could occur
- Passes the buck to the caller to handle the exception

- In HW#8's `compress` and `decompress` methods, we say `throws IOException` to avoid having to handle IOExceptions