CSE 143
Lecture 24

Priority Queues; Huffman Encoding

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

Prioritization problems

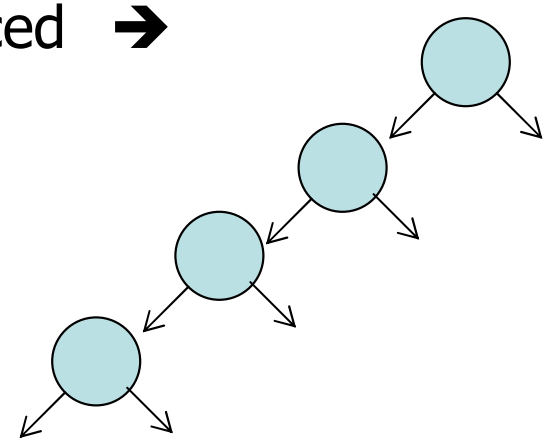
- The CSE lab printers constantly accept and complete jobs from all over the building. Suppose we want them to print faculty jobs before staff before student jobs, and grad students before undergraduate students, etc.?

- 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 neck, 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 poor choices

- *list* : store customers/jobs in a list; remove min/max by searching (O(*N*))
  - problem: expensive to search
- *sorted list* : store in sorted list; binary search it in O(log *N*) time
  - problem: expensive to add/remove (O(*N*))
- *binary search tree* : store in BST, search in O(log *N*) time for min element
  - problem: tree could be unbalanced ➔



Priority queue ADT

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

- priority queue operations:

– `add`	adds in order;	O(log *N*) worst
– `peek`	returns **minimum** value;	O(1) always
– `remove`	removes/returns **minimum** value;	O(log *N*) worst
– `isEmpty`, `clear`, `size`, `iterator`		O(1) always

Java's `PriorityQueue` class

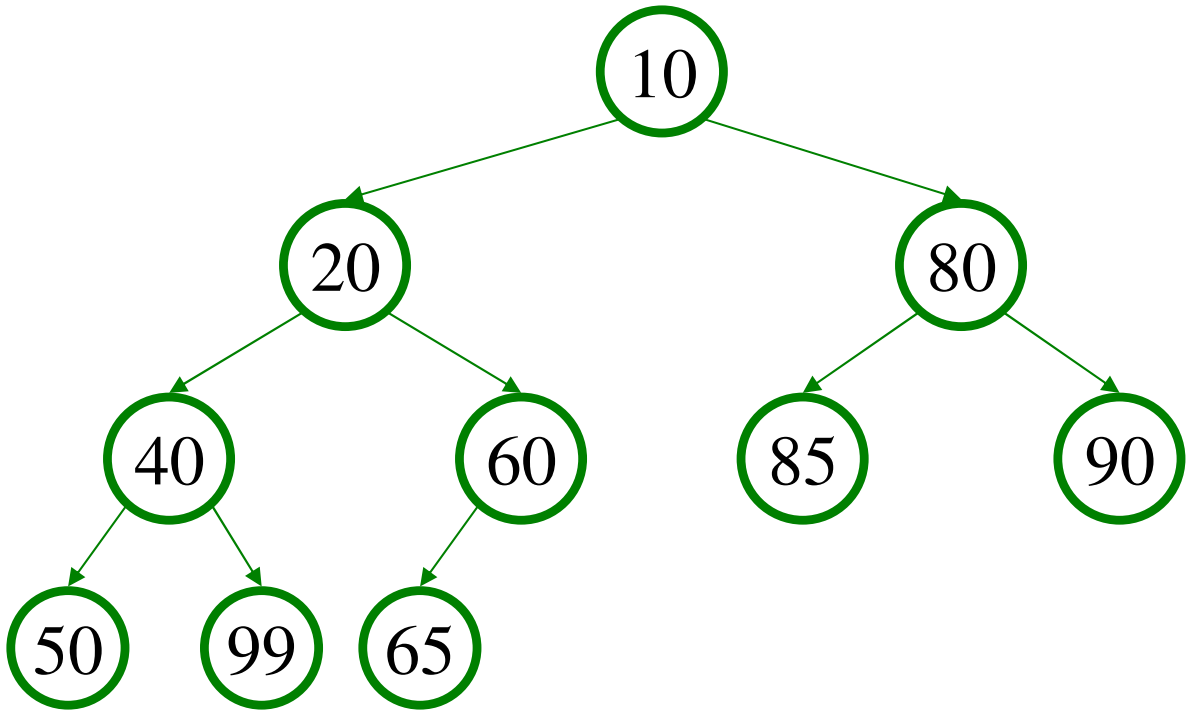
```
public class PriorityQueue<E> implements Queue<E>
```

Method/Constructor	Description	Runtime
`PriorityQueue<E>()`	constructs new empty queue	O(1)
`add(E value)`	adds value in sorted order	O(log *N*)
`clear()`	removes all elements	O(1)
`iterator()`	returns iterator over elements	O(1)
`peek()`	returns minimum element	O(1)
`remove()`	removes/returns min element	O(log *N*)

```
Queue<String> pq = new PriorityQueue<String>();
pq.add("Stuart");
pq.add("Marty");
...
```

Inside a priority queue

- Usually implemented as a **heap**: a kind of binary tree.
- Instead of sorted left $\rightarrow$ right, it's sorted top $\rightarrow$ bottom
  - guarantee: each child is greater (lower priority) than its ancestors
  - add/remove causes elements to "bubble" up/down the tree
  - (take CSE 332 or 373 to learn about implementing heaps!)



Exercise: Firing Squad

- We have decided that TA performance is unacceptably low.
  - We must fire all TAs with ≤ 2 quarters of experience.

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

  - Input format:

name quarters
```
Lisa 0
```
name quarters
```
Kasey 5
```
name quarters
```
Stephanie 2
```

Priority queue ordering

- For a priority queue to work, elements must have an ordering
  - in Java, this means implementing the `Comparable` interface

- Reminder:

```
public class Foo implements Comparable<Foo> {
    …
    public int compareTo(Foo other) {
        // Return positive, zero, or negative number...
    }
}
```

Homework 8
(Huffman Tree)

File compression

- **compression**: Process of encoding information in fewer bits.
  - But isn't disk space cheap?

- Compression applies to many things:
  - store photos without exhausting disk space
  - reduce the size of an e-mail attachment
  - make web pages smaller so they load faster
  - reduce media sizes (MP3, DVD, Blu-Ray)
  - make voice calls over a low-bandwidth connection (cell, Skype)

- Common compression programs:
  - WinZip or WinRAR for Windows
  - Stuffit Expander for Mac

ASCII encoding

- **ASCII**: Mapping from characters to integers (binary bits).
  - Maps every possible character to a number (`'A'` → 65)
  - uses one *byte* (8 *bits*) for each character
  - most text files on your computer are in ASCII format

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

Huffman Encoding

- **Huffman encoding**: Uses variable lengths for different characters to take advantage of their relative frequencies.
  - Some characters occur more often than others.
    If those characters used < 8 bits each, the file would be smaller.
  - Other characters need > 8, but that's OK; they're rare.

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

Huffman compression

- The following major steps can be used to compress a file:

  1. Count the occurrences of each character in the file.

  2. Place the characters and their counts into a priority queue.

  3. Use the priority queue to create a special binary tree called a **Huffman tree**.

  4. Traverse the Huffman tree to find a (character -> binary) mapping from each character to its binary encoding.

  5. For each character in the original file, convert it into its compressed binary encoding.

1) Counting characters

- **step 1**: count occurrences of characters into a map (provided)
  - example input file contents:

```
ab ab cab
```

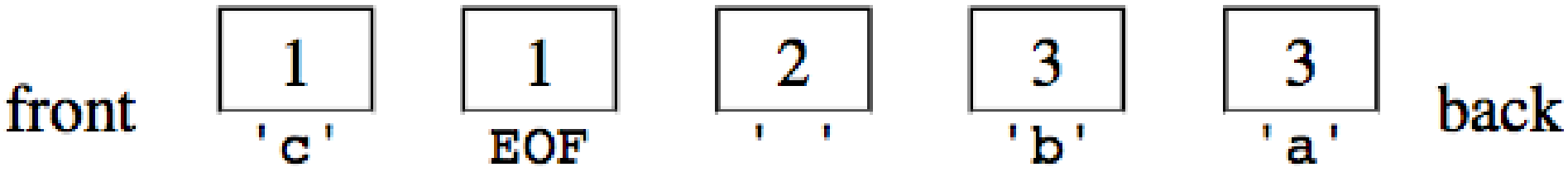
byte	1	2	3	4	5	6	7	8	9	10
char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A

```
counts: {' '=2, 'a'=3, 'b'=3, 'c'=1, EOF=1}
```

  - every file actually ends with a single invisible character indicating the end-of-file (**EOF**)

2) Create priority queue

- **step 2**: place characters and counts into a priority queue
  - store a single character and its count as a Huffman node object
  - the priority queue will organize them into ascending order

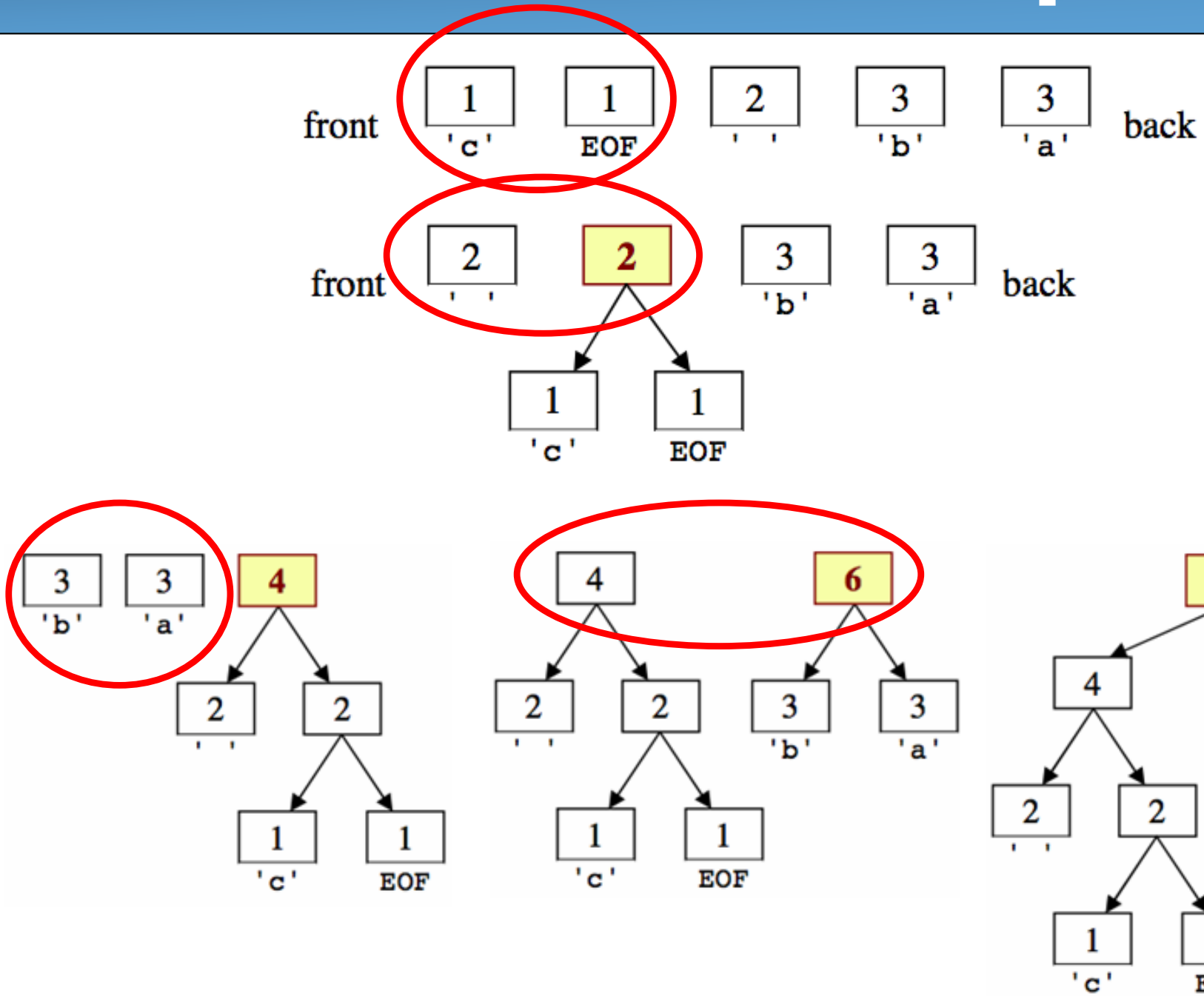


3) Build Huffman tree

- **step 2**: create "Huffman tree" from the node counts

  algorithm:
- Put all node counts into a **priority queue**.
- while P.Q. size > 1:
  - Remove two rarest characters.
  - Combine into a single node with these two as its children.

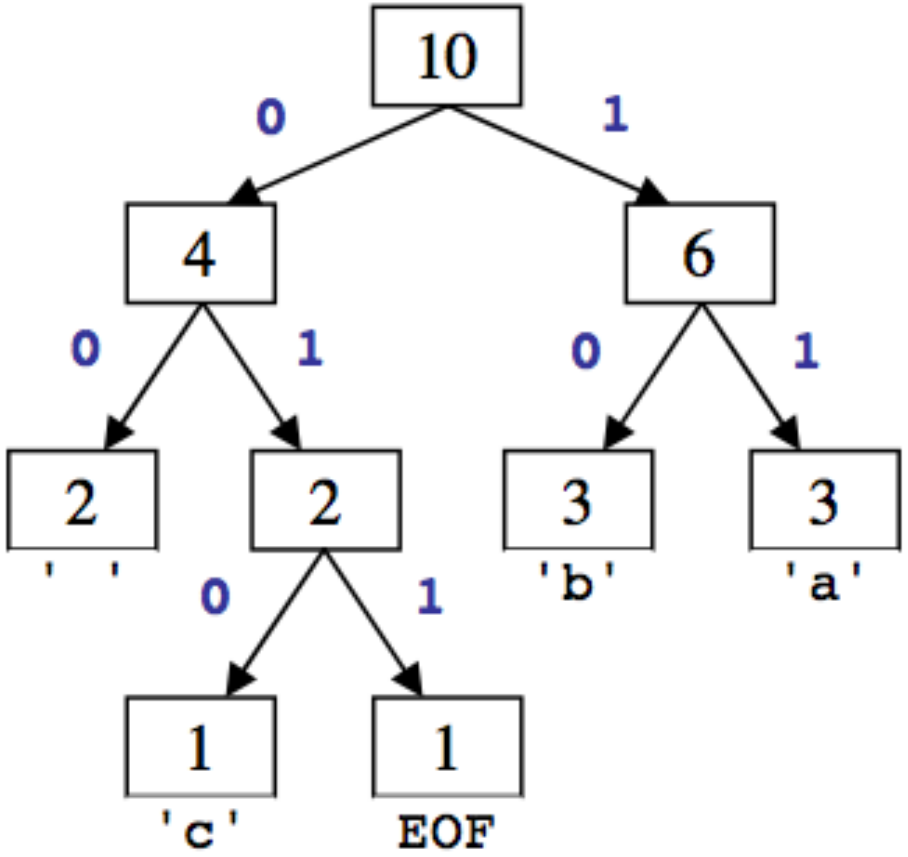
Build tree example



4) Tree -> binary encodings

- The Huffman tree you have created tells you the compressed binary encoding to use for every character in the file.
  - left branches mean `0`, right branches mean `1`
  - example: `'b'` is `10`
  - What are the binary encodings of:

    EOF,
    `' '`,
    `'b'`,
    `'a'`?



5) compress the actual file

- Based on the preceding tree, we have the following encodings:

  `{' '=00, 'a'=11, 'b'=10, 'c'=010, EOF=011}`

  - Using this map, we can encode the file into a shorter binary representation. The text ab ab cab would be encoded as:

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary	11	10	00	11	10	00	010	11	10	011

  - Overall: `1110001110000101110011`, (22 bits, ~3 bytes)

byte	1	2	3
char	a b a	b c a	b EOF
binary	11 10 00 11	10 00 010 1	1 10 011 00

Decompressing

- How do we decompress a file of Huffman-compressed bits?

- useful "prefix property"
  - No encoding A is the prefix of another encoding B
  - I.e. never will have $x \rightarrow$ `011` and $y \rightarrow$ **`011`**`100110`

- the algorithm:
  - Read each bit one at a time from the input.
  - If the bit is 0, go left in the tree; if it is 1, go right.
  - If you reach a leaf node, output the character at that leaf and go back to the tree root.

Decompressing

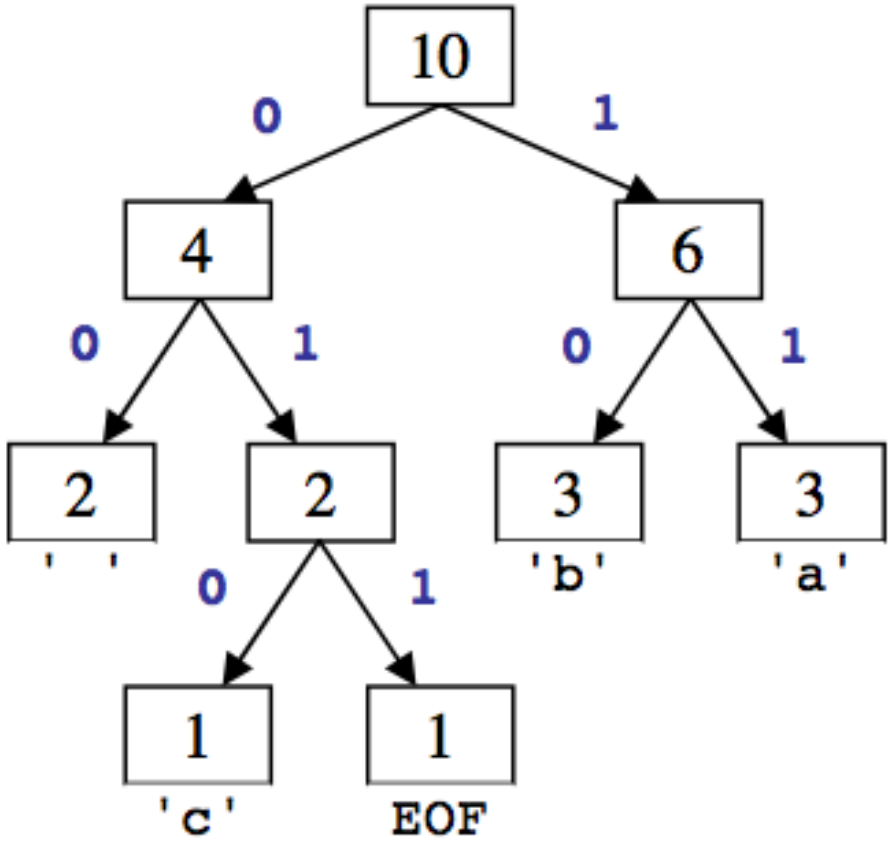
- Use the tree to decompress a compressed file with these bits:

  `1011010000110111011`

- the algorithm:
  - Read each bit one at a time.
  - If it is 0, go left; if 1, go right.
  - If you reach a leaf, output the character there and go back to the tree root.

- Output:

  `bac aca`



Public methods to write

- **`HuffmanTree`**`(Map<Character, Integer> counts)`
  - Given a Map of counts/char in a file, creates its Huffman tree.
- `public Map<Character, String>` **`createEncodings`**`()`
  - Traverses Huffman tree to produce a mapping from each character in the tree to its binary representation. Example:
    {' '=010, 'a'=11, 'b'=00, 'd'=011, 'n'=10}
- `public void` **`compress`**`(InputStream in,`
  `BitOutputStream out) throws IOException`
  - Reads text data from given input file and uses encodings to write a compressed version of the data to the given output file
- `public public void` **`decompress`**`(BitInputStream in,`
  `OutputStream out) throws IOException`
  - Reads compressed binary data from given input and uses Huffman tree to write decompressed version to the output.

Provided Bit I/O Streams

- Java's input/output streams read/write 1 *byte* (8 bits) at a time.
  - We want to read/write one single *bit* at a time.

- `BitInputStream`: Reads one bit at a time from input .

`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

- `BitOutputStream`: Writes one bit at a time to output.

`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