CSE 143

Lecture 24: Priority Queues and Huffman Encoding

"It's our new method for determining who we should treat first. We take people in order of how loud they scream."

Prioritization problems

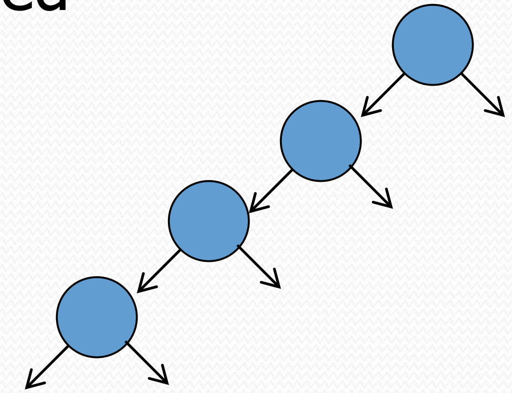
- **ER scheduling:** You are in charge of scheduling patients for treatment in the ER. A gunshot victim should probably get treatment sooner than that one guy with a sore neck, regardless of arrival time. How do we always choose the most urgent case when new patients continue to arrive?

- **print jobs:** 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.?

- *What would be the runtime of solutions to these problems using the data structures we know (list, sorted list, map, set, BST, etc.)?*

Inefficient structures

- *list* : store 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, go right for max in O(log *N*)
  - problem: tree becomes unbalanced



Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element

- 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

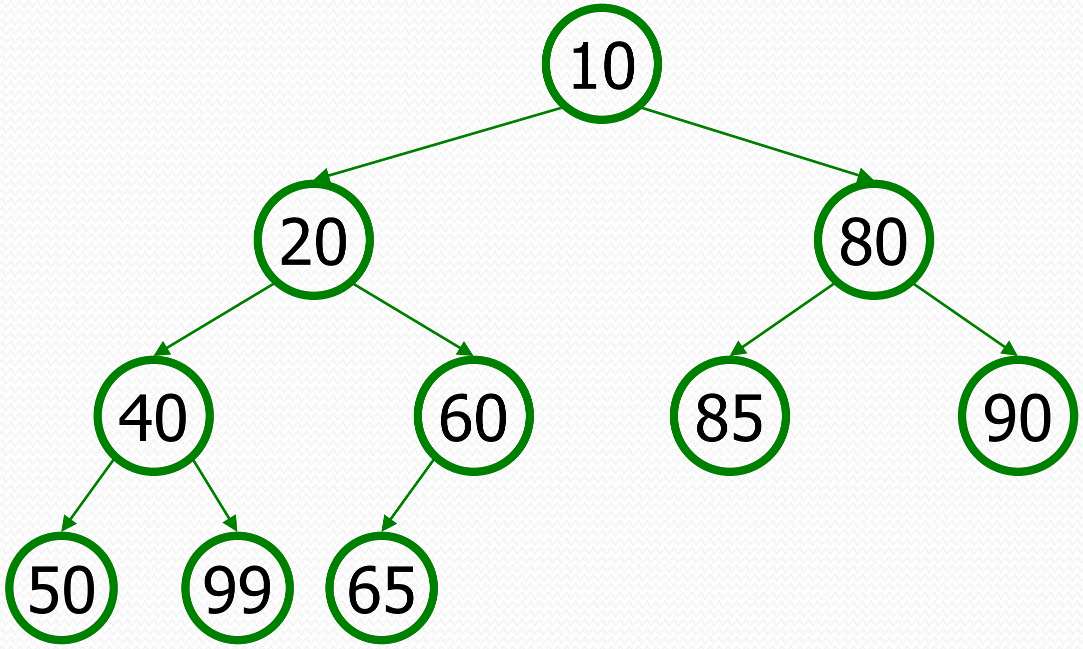
```
public class PriorityQueue<E> implements Queue<E>
```

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*)
`size()`	number of elements in queue	O(1)

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

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: Fire the TAs

- We have decided that novice Tas should all be fired.
  - Write a class `TAManager` that reads a list of TAs from a file.
  - Find all with ≤ 2 quarters experience, and replace them.
  - Print the final list of TAs to the console, sorted by experience.

  - Input format:

name quarters	`Will 3`
name quarters	`Ying 2`
name quarters	`Andrew 1`

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 integer
    }
}
```

Homework 8
(Huffman Coding)

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'` $\rightarrow$ 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

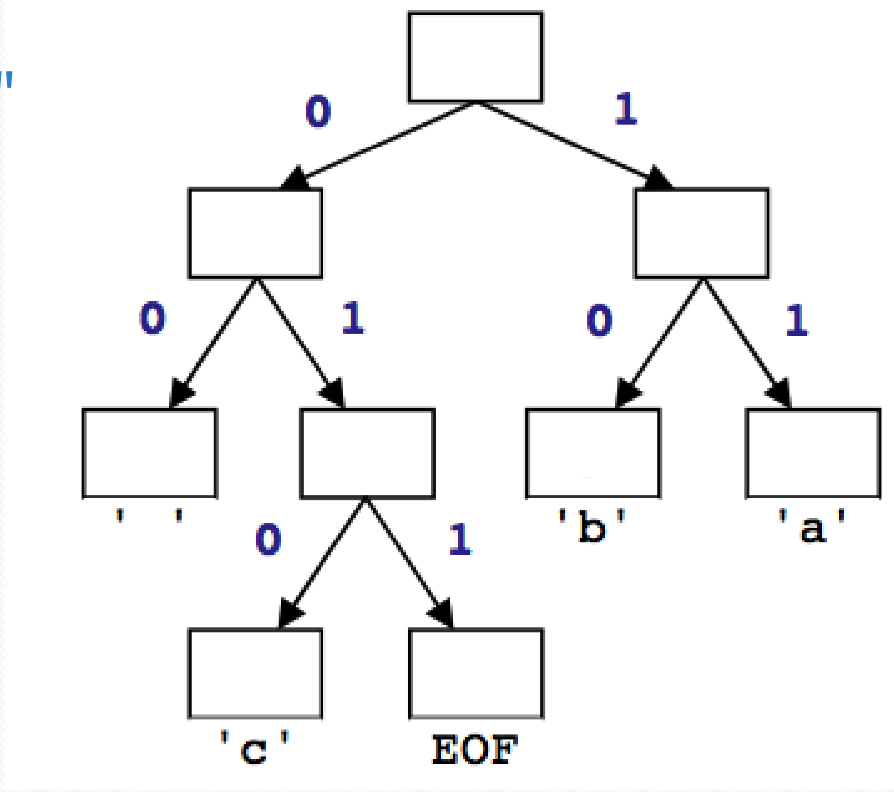
Huffman encoding

- **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 use < 8 bits each, the file will 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's algorithm

- *The idea:* Create a "Huffman Tree" that will tell us a good binary representation for each character.
  - Left means 0, right means 1.
    - example: 'b' is 10
  - More frequent characters will be "higher" in the tree (have a shorter binary value).



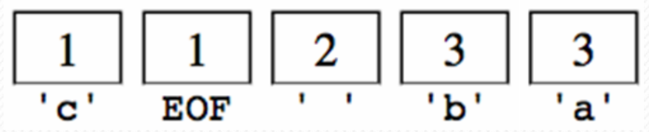
- To build this tree, we must do a few steps first:
  - **Count occurrences** of each unique character in the file.
  - Use a **priority queue** to order them from least to most frequent.

Huffman compression

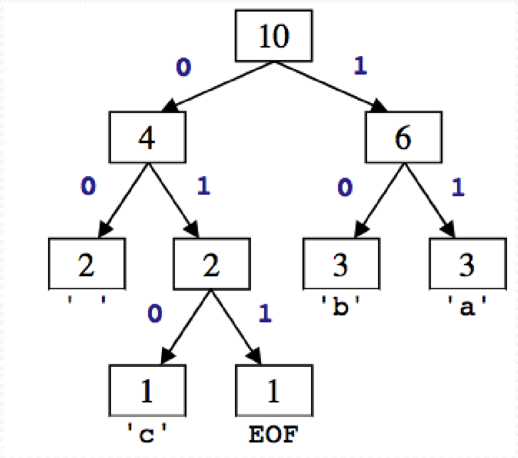
1. Count the occurrences of each character in file

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

2. Place characters and counts into **priority queue**



3. Use priority queue to create **Huffman tree** →



4. Traverse tree to find (char → binary) map

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

5. For each char in file, **convert** to compressed binary version

```
11 10 00 11 10 00 010 1 1 10 011 00
```

1) Count characters

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

```
ab ab cab
```

byte	1	2	3	4	5	6	7	8	9
char	`'a'`	`'b'`	`' '`	`'a'`	`'b'`	`' '`	`'c'`	`'a'`	`'b'`
ASCII	97	98	32	97	98	32	99	97	98
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010

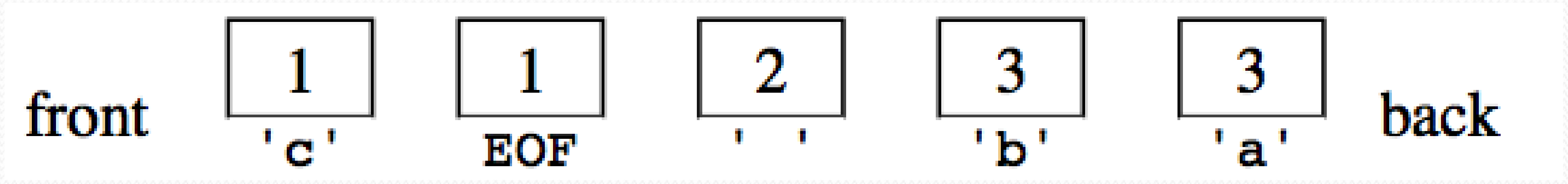
counts array:

index	0	1	...	32	...	97	98	99	100	...
value	0	0		2		3	3	1	0	

- *(in HW8, we do this part for you)*

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

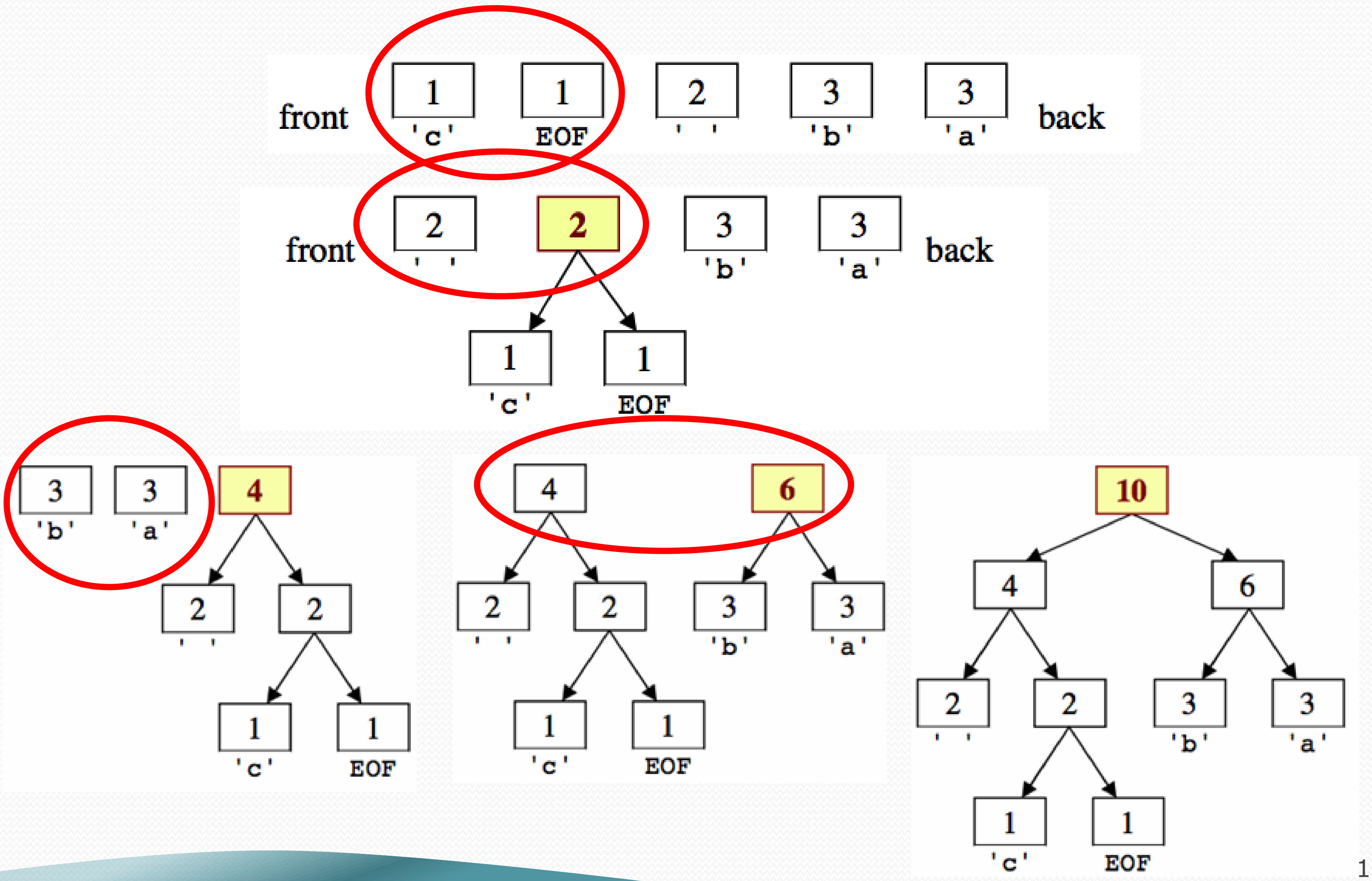


3) Build Huffman tree

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

  <u>algorithm:</u>
- 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.

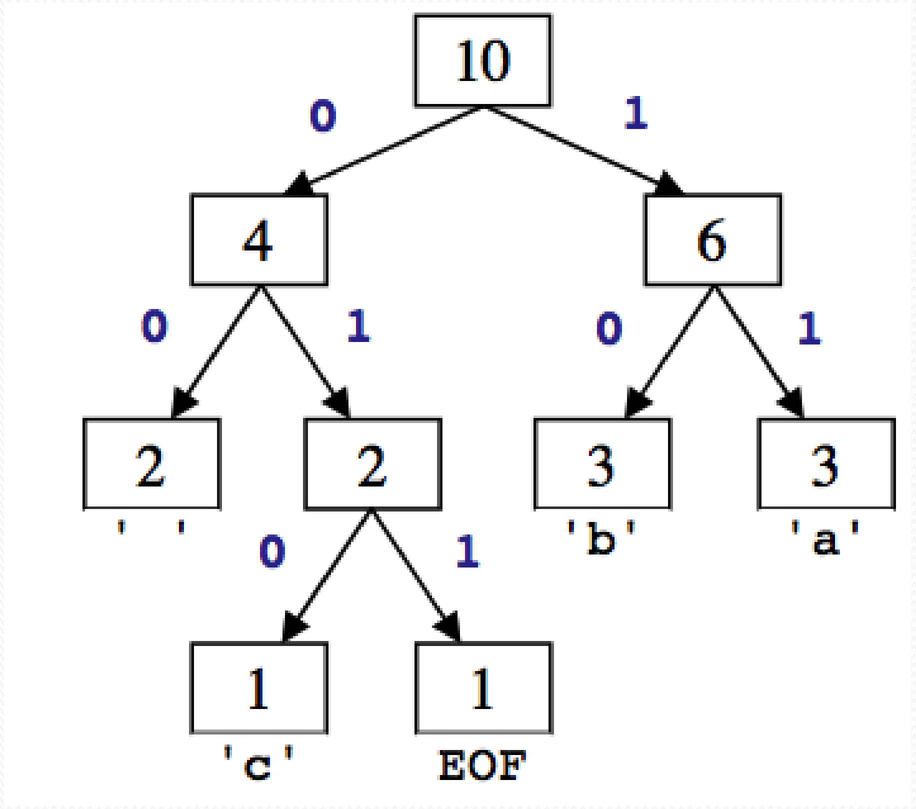
Build tree example



4) Tree to binary encodings

- The Huffman tree tells you the binary encodings to use.
  - left means `0`, right means `1`
  - example: `'b'` is `10`
  - What are the binary encodings of:

    EOF,
    `' '`,
    `'c'`,
    `'a'`?



  - *What is the relationship between tree branch height, binary representation length, character frequency, etc.?*

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

  - `Encode.java` does this for us using our codes file.
  - *How would we go back in the opposite direction (decompress)?*

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

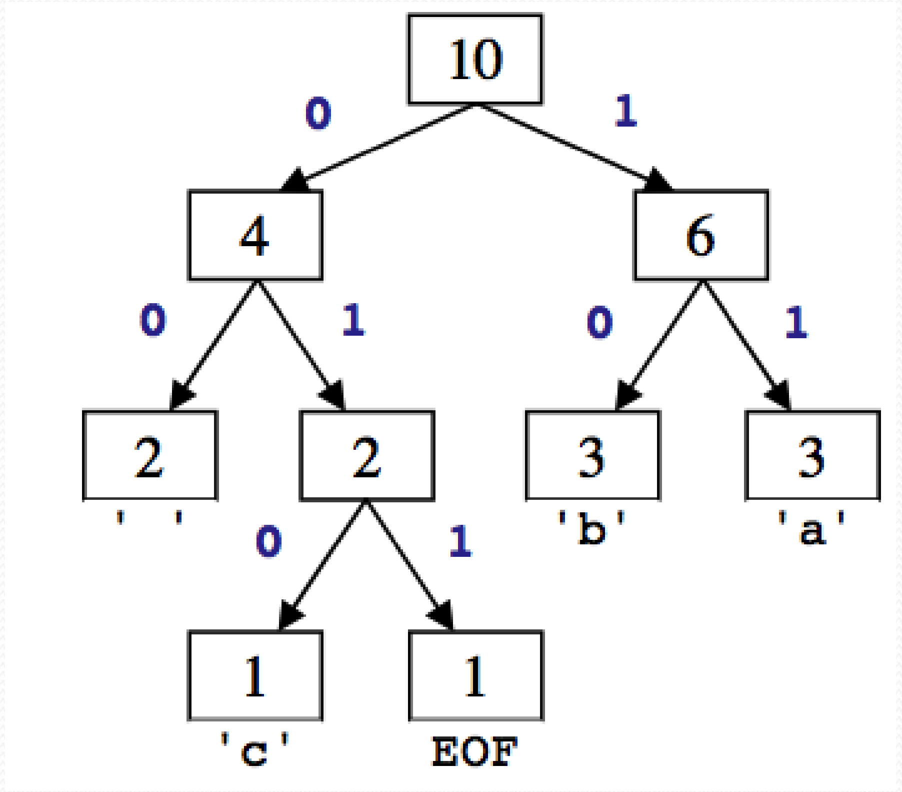
- Use the tree to decompress a compressed file with these bits:

```
1011010001101011011
 b a  c _ a  c a
```

  - 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

- `public HuffmanTree(int[] counts)`
  - Given character frequencies for a file, create Huffman tree *(Steps 2-3)*
- `public void write(PrintStream output)`
  - Write mappings between characters and binary to a `.code` file *(Step 4)*
- `public HuffmanTree(Scanner input)`
  - Reconstruct the tree from a `.code` file
- `public void decode(BitInputStream in, PrintStream out, int eof)`
  - Use the Huffman tree to decode characters

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`**`(String file)`	Creates stream to read bits from given file
`public int` **`readBit`**`()`	Reads a single 1 or 0
`public void` **`close`**`()`	Stops reading from the stream

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

`public` **`BitOutputStream`**`(String file)`	Creates stream to write bits to given file
`public void` **`writeBit`**`(int bit)`	Writes a single bit
`public void` **`close`**`()`	Stops reading from the stream