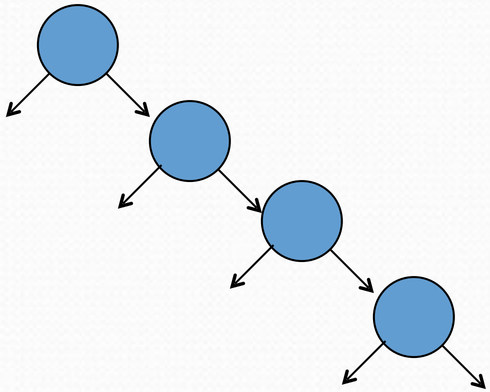
Priority Queues & Huffman Encoding

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?

Structure Options

- list : store people in a list; remove min/max by searching (O(N))
  - problem: expensive to search
- sorted list : store in sorted list; remove in O(1) time
  - problem: expensive to add (O(N))
- binary search tree : store in BST, go right for min in O(log N)
  - problem: tree becomes unbalanced



Java's `PriorityQueue` class

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

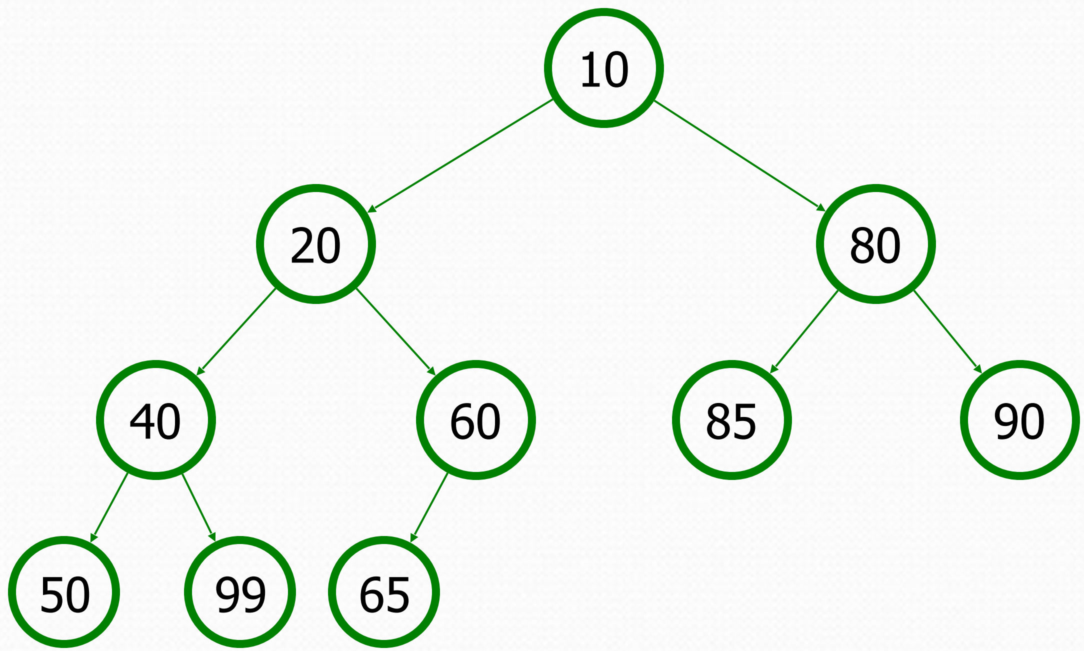
```
public class PriorityQueue<E> implements Queue<E>
```

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("Rasika");
pq.add("Radu");
...
```

Inside a priority queue

- Usually implemented as a **heap**, a kind of binary tree.
- Instead of sorted left → right, it's sorted top → 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!)

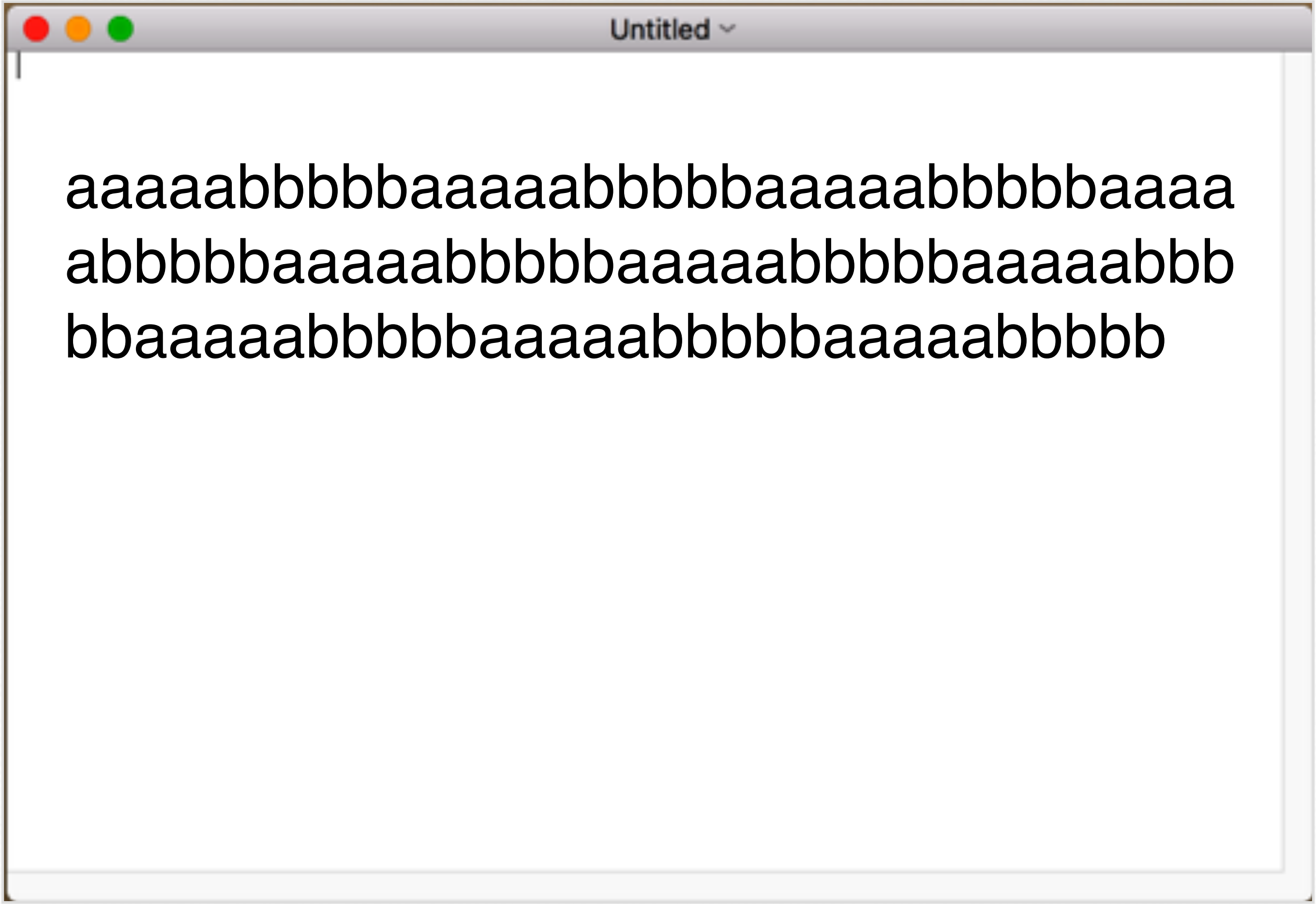


Homework 11
(Huffman Coding)

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



100 characters, 50 a's, 50 b's

Huffman encoding

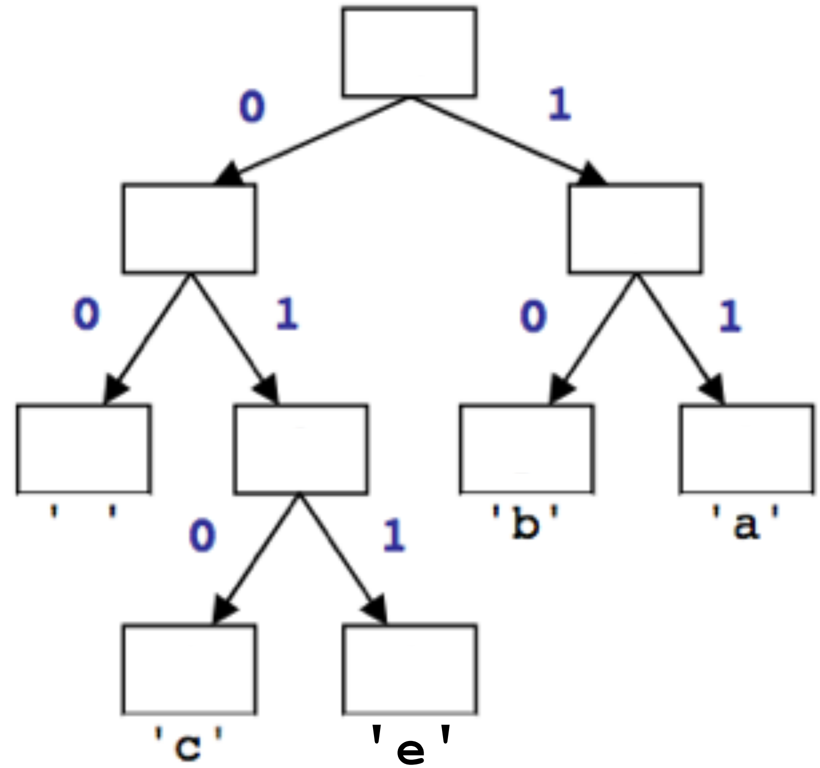
- **Huffman encoding**: Uses variable lengths for different characters to take advantage of their relative frequencies.
  - Some characters occur more often than others.
    If those characters use < 8 bits each, the file will be smaller.
  - Other characters need > 8, but that's OK; they're rare.

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
  - Example: 0001010

    ' cb'

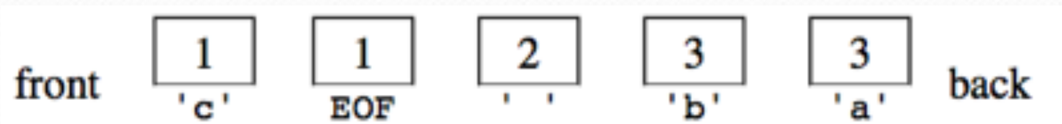


Huffman compression

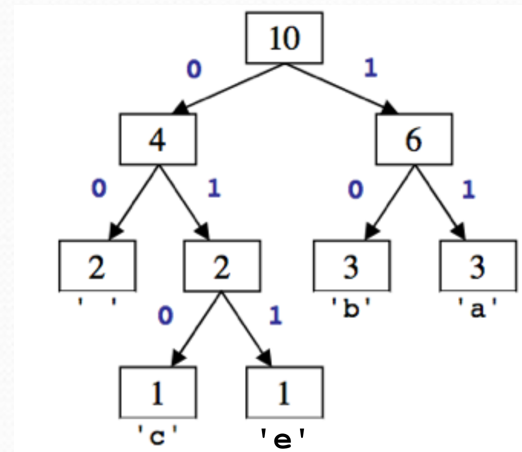
1. Count the occurrences of each character in file

`{' '=2, 'a'=3, 'b'=3, 'c'=1, 'e'=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, 'e'=011}`

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

```
 a  b        a  b          c    a  b    e
11 10 00 11 10 00 010 11 10 011
```

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

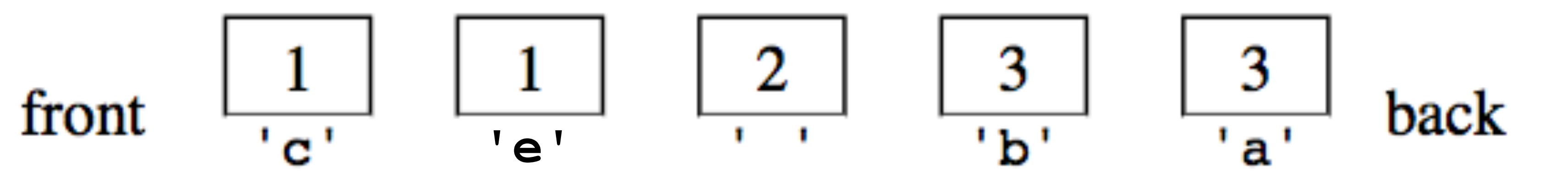
counts array:

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

  - (in HW11, 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

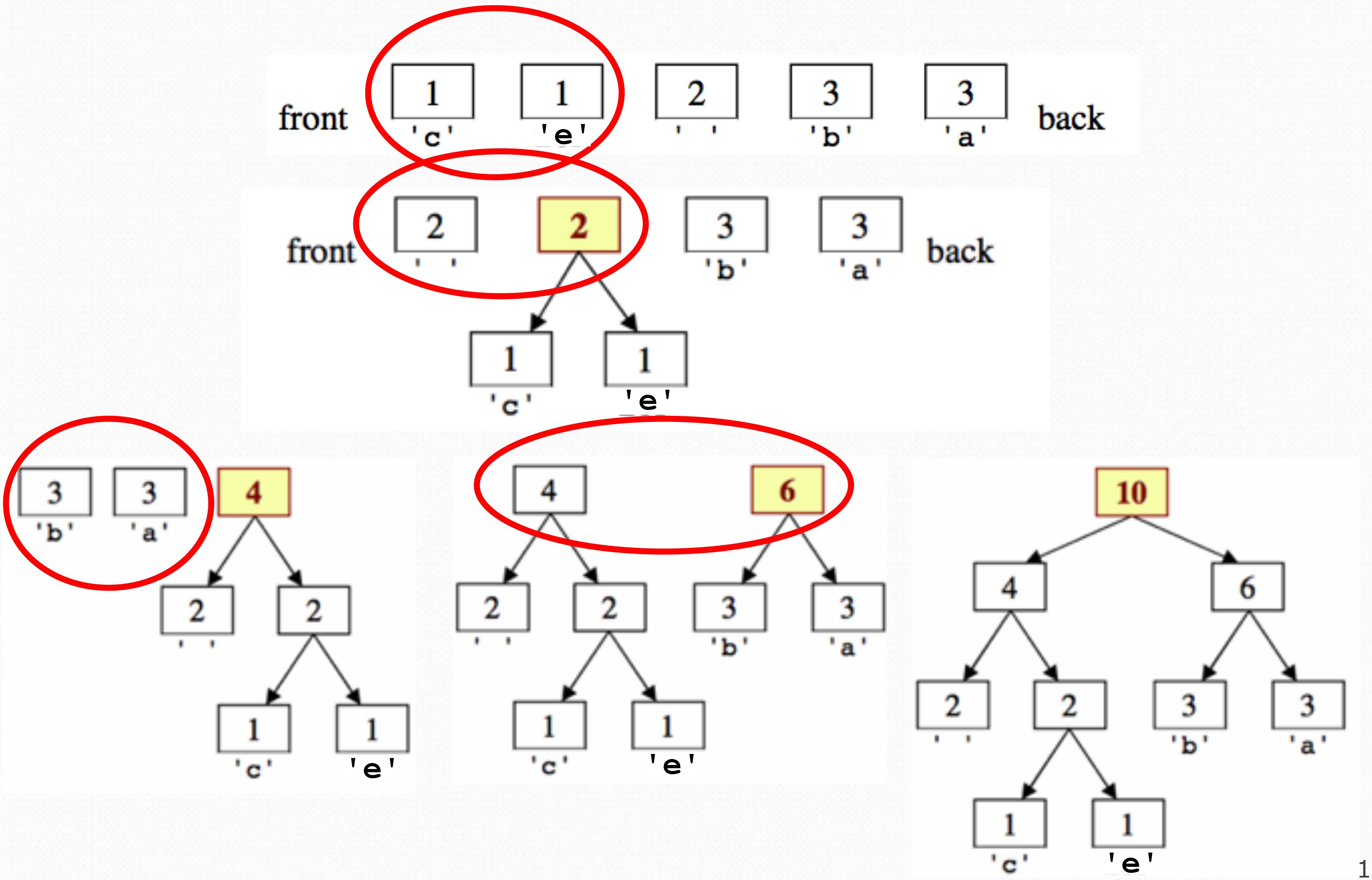


3) Build Huffman tree

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

  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.

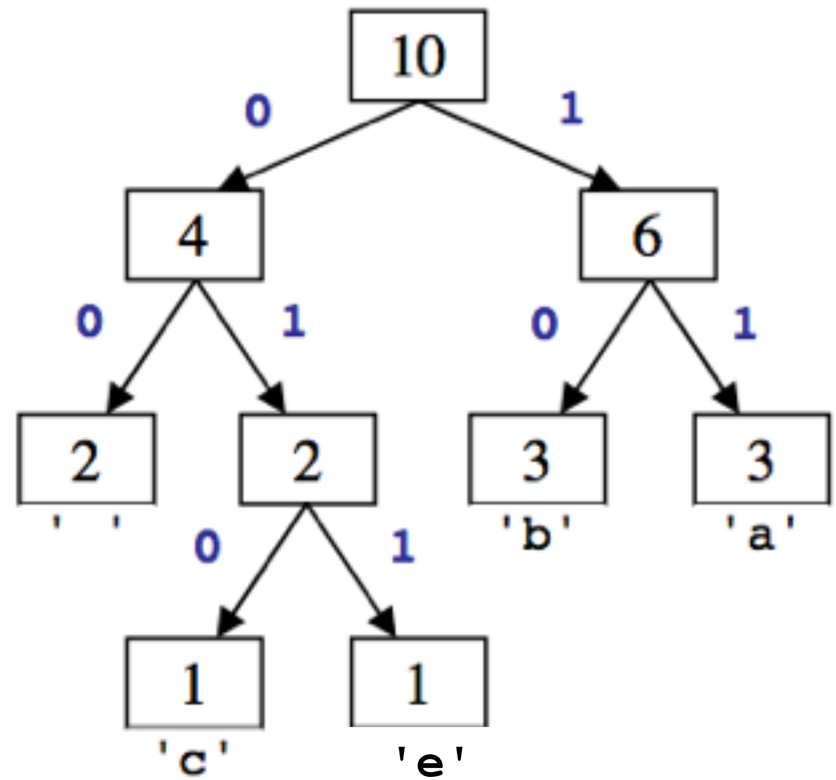
Build tree example



4) Tree 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:

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



5) compress the actual file

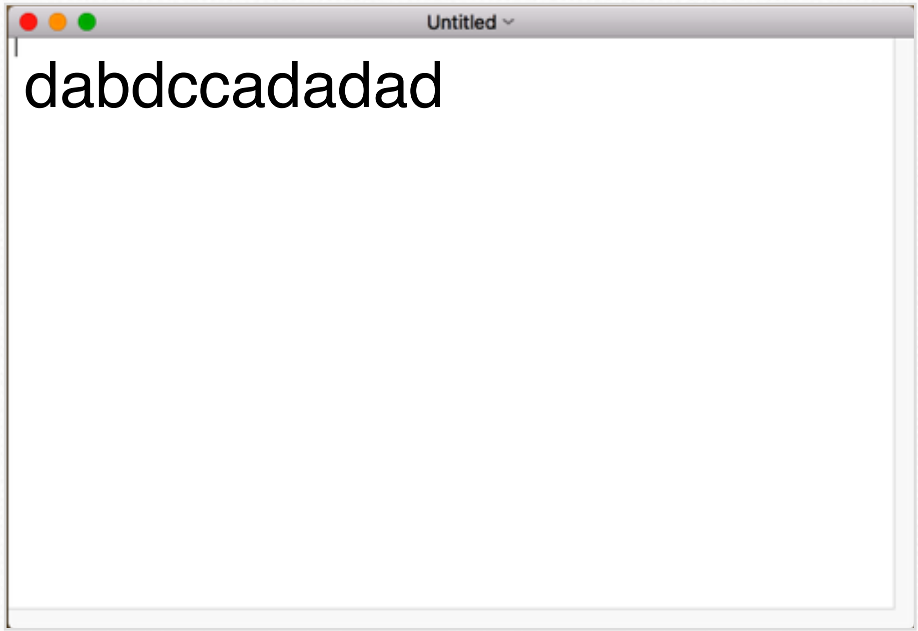
- Based on the preceding tree, we have the following encodings:
  `{' '=00, 'a'=11, 'b'=10, 'c'=010, 'e'=011}`
  - Using this map, we can encode the file into a shorter binary representation. The text `ab ab cab` would be encoded as:

char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	'e'
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 e
binary	11 10 00 11	10 00 010 1	1 10 011

Compression example



Compressed binary:

0 11 100 0 101 101 11 0 11 0 11 0

Decompressing

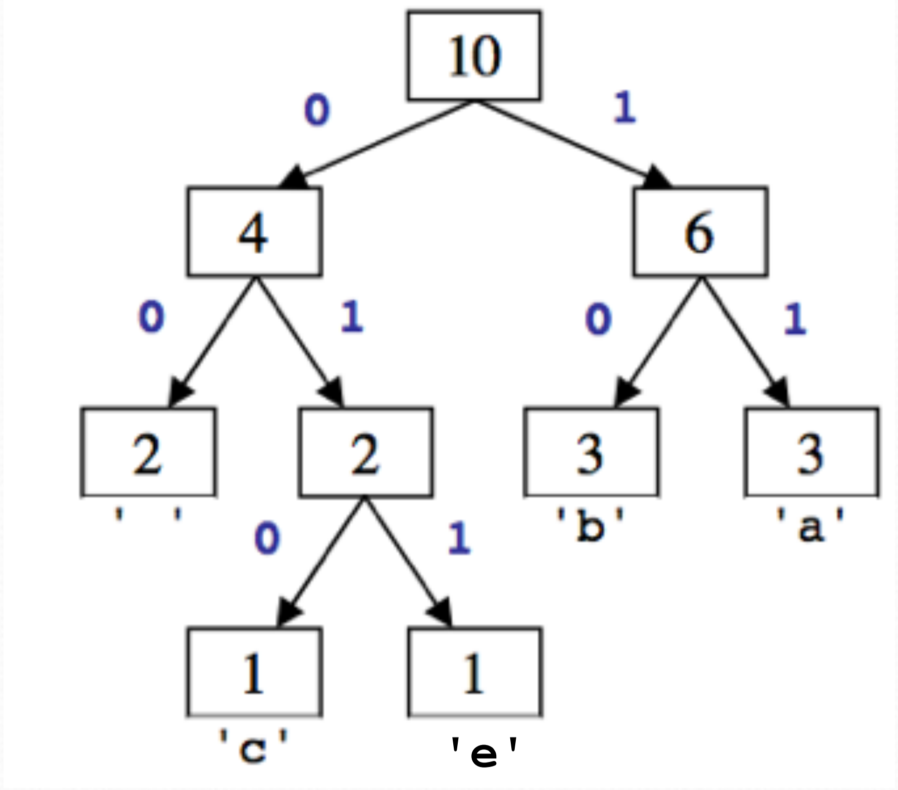
How do we decompress a file of Huffman-compressed bits?

- Useful "prefix property"
  - No encoding A is the prefix of another encoding B
  - I.e. never will have x → `011` and y → **`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 th
    root.

Decompressing

- Use the tree to decompress a compressed file with these bits:
  `1011010001101011011`
  - 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 HuffmanCode(int[] frequencies)`
  - Given character frequencies for a file, create Huffman code (Steps 2-3)

- `public void save(PrintStream output)`
  - Write mappings between characters and binary to a output stream (Step 4)

- `public HuffmanCode(Scanner input)`
  - Reconstruct the tree from a `.code` file

- ```
  public void translate(BitInputStream input,
                         PrintStream output)
  ```
  - Use the Huffman code to decode characters
  ```

Bit input stream

- Java's input stream reads 1 byte (8 bits) at a time.
  - We want to read 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 hasNextBit()`	Checks to see if stream still has input

That's it!