Written by Alec Silverstein - 03/31/2017

Applications of Priority Queues: Huffman Coding

Text Reference: Weiss Section 10.1.2 (page 433)

Here is a great, live visualization tool for encoding and decoding a Huffman tree:
https://opendsa-server.cs.vt.edu/ODSA/AV/Binary/huffmanBuildAV.html

Huffman coding allows for efficiently encoding messages and data before they are sent. It is an essential tool in **file compression**. We send a computer message in binary *1*s and *0*s and must decide a code consisting of a binary string mapping to each character in the message. ASCII representation assigns a 7-bit string to each character in the language, but is inefficient if we are sending short messages that only have a handful of characters. Using a more efficient encoding algorithm can save as much as 50% space.

We can choose which characters get longer or shorter bit strings based on their frequencies in the message. Clearly, we want to assign longer bit strings to characters with lower frequency, as the resultant message size will be shorter.

A **prefixless** bit string is one in which no pattern has the same beginning bits. We want all prefixless strings in the Huffman encoding to be able to clearly distinguish one character from another. One bit string cannot be a prefix to another.

Ex. Consider the encoding of the string *aaaaaaabca*. Say we assign the code:

a ==> 0
b ==> 10
c ==> 11

The assignment is *prefixless* because we cannot confuse the values of any two characters when evaluating left to right. If we see a *1*, we must check if the next character is *0* or *1*, signifying *b* or *c*, respectively. *01110* must be *acb*.

The following code is *not* prefixless:

a ==> 0
b ==> 1
c ==> 01

We do not know if *01* corresponds to *ab* or *c*.

A Huffman tree encapsulating an encoding may look something like:

```
      O
    /   \
   a     O
        / \
       b   c
```

The circles represent empty nodes with no symbol in them. Only the *leaf nodes* contain actual values, which allows the tree to be prefixless. We can assign movement to the left as a *0*, and movement to the right as a *1*.

Formally, the **Huffman algorithm for binary encoding** is an optimal encoding method for a message using a prefixless binary tree depending on the frequency of the corresponding characters. Infrequent characters have longer codes and are hence lower down in the tree. The Huffman algorithm is a **greedy algorithm**, meaning at each step it will always choose the best option at the moment, in this case the lowest cost character combination.

The algorithm has three steps using a priority queue/min binary heap:

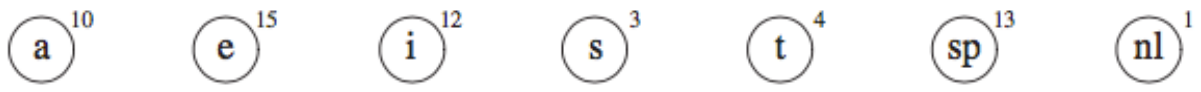
Step 1: Pick the two least frequency nodes (deleteMin).

Step 2: Push the two nodes together as children of a temporary/blank node. Push this new node back onto the heap.

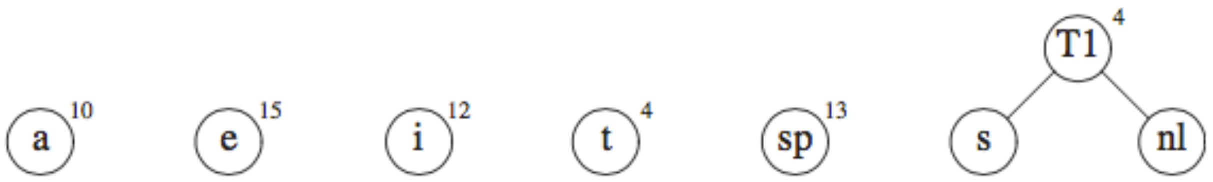
Step 3: Return to Step 1 and repeat until the heap contains only one element, the entire Huffman tree (the top node having frequency equal to the total number of characters/total frequency of the message).

Encoding and Decoding a Huffman Tree by Example (see Weiss chapter)

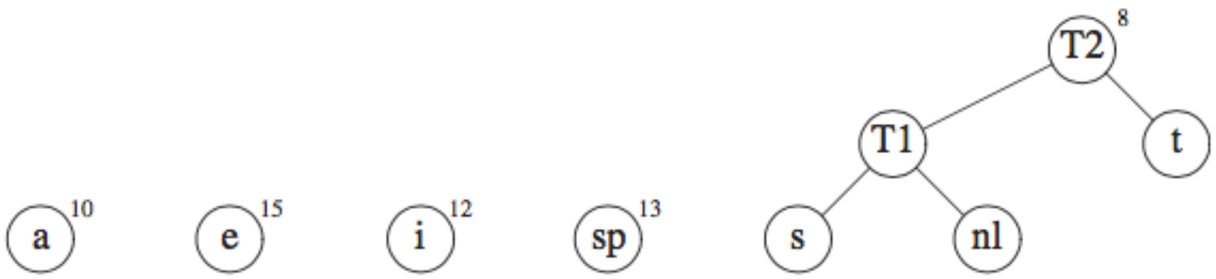
Given a message as text, we can form a forest of nodes for each character. Here, the subscript number indicates the total frequency/number of times that character appears in the given message:



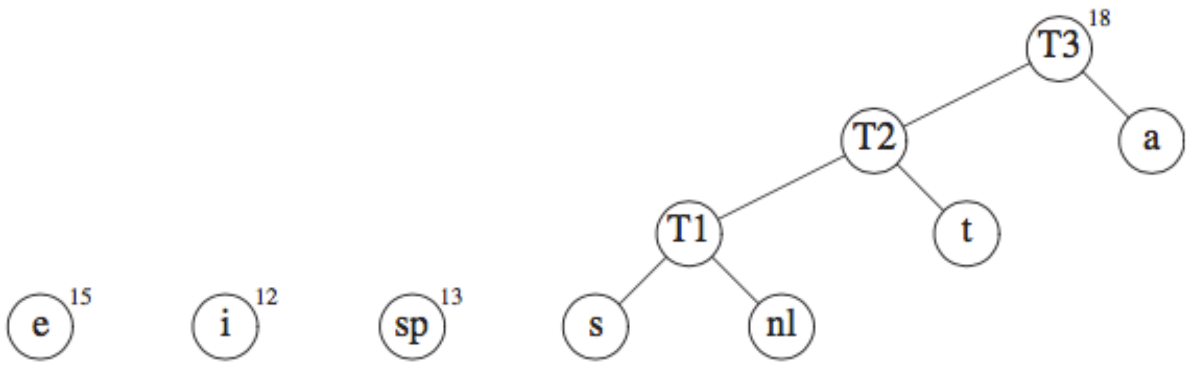
The smallest frequency nodes are “nl” with frequency 1 and “s” with frequency 3. We delete the two mins and combine them under a new parent node with arbitrary label “T1”. It does not matter which is the right or left child, they will lead to the same result.



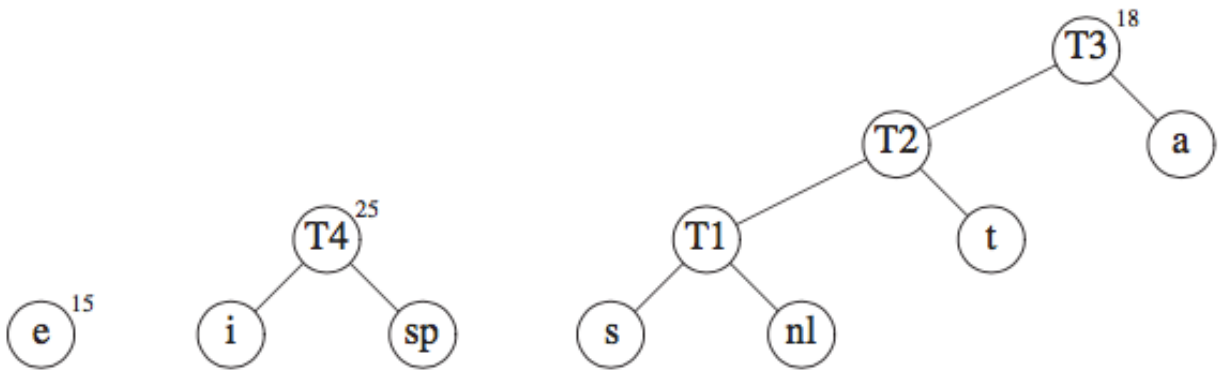
Smallest nodes are T1 and t



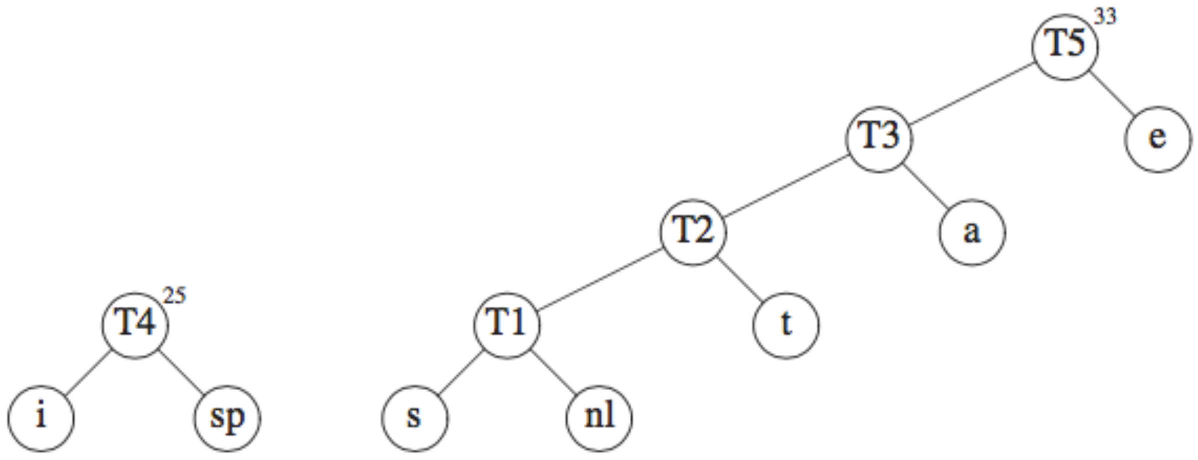
Smallest nodes are T2 and a



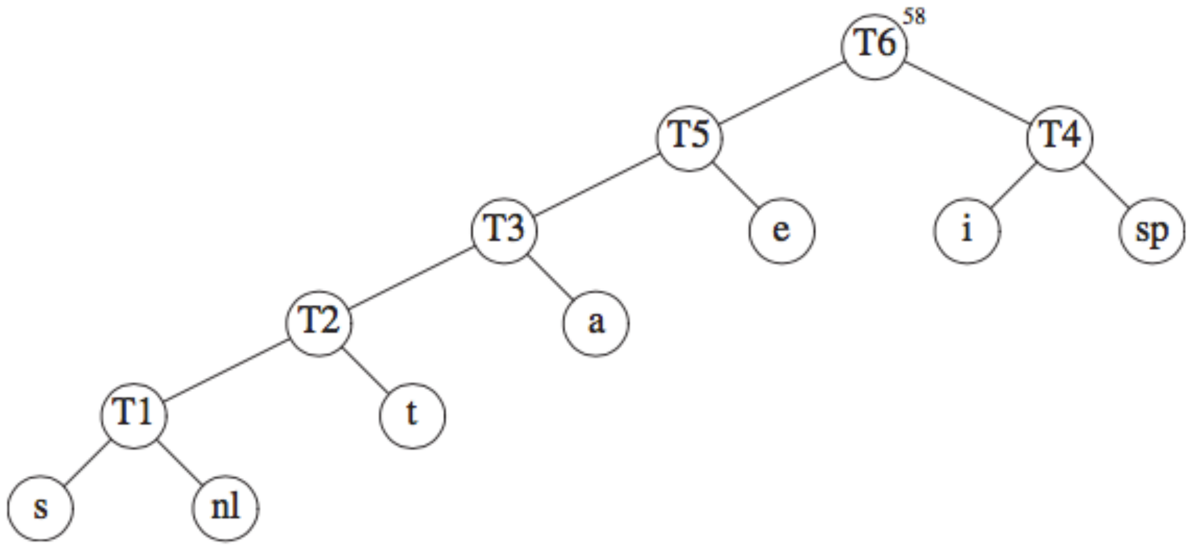
Smallest nodes are i and sp - create a new node T4



Smallest nodes are e and T3



Smallest nodes are T4 and T5



This is the final optimal Huffman encoding, as all characters are contained in leaf nodes. T6 contains the entire encoding. The following table summarizes the encoding:

Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5
Total			146

Notice the improvement over a non-optimal encoding:

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	13	39
newline	110	1	3
Total			174

An important note is that the encoding is not unique, as we arbitrarily choose to put a node as the right or left child. Only the *length* of the encoding matters for optimality.

To decode the tree, we can use tree traversal: for each movement left, add a *0* to the string, and for each movement right, add a *1*. We can also use a hash map to map each character to its bit string.

You need both the message *and* the original tree to decode it. The **transmission cost** is the length of the optimal encoding plus the size of the tree. The cost is **minimal** and efficient if the tree is small compared to the relative size of the message. Note that the algorithm is a waste of time and resources for very small messages.

Homework

For Homework 4, Programming 2 you will build a Huffman encoding for a message.
Some hints:

- You can use a Java HashMap to keep track of character frequencies
- Create a HuffmanNode object to keep track of each key and its frequency
- Use a priority queue
- The last element will be the root node of the Huffman tree