Huffman Trees

Heaps

- **The Huffman Code:**
- Huffman algorithm uses a binary tree to compress data. It is called the Huffman code, after David Huffman who discovered it in 1952. Data compression is important in many situations. An example is sending data over the Internet, where especially over a dial-up connection, transmission can take a long time.

Heaps

- **Character Codes:**
- Each character in a normal uncompressed text file is represented in computer by one byte (for the ASCII Code) or by two bytes (for Unicode). In these schemes, every character requires the same number of bits.
- There are several approaches to compressing data. For text, the most common approach is to reduce the number of bits that represent the most-used characters. In this approach we must be careful that no character is represented by the same bit combination that appears at the beginning of a longer code used for some other character. For example, if E is 01, and X is 01011000, then anyone decoding 01011000 would not know if the initial 01 represented an E or the beginning of an X. This leads to a rule: No code can be the prefix of any other code.

Heaps

- For each message, we make up a new code tailored to that particular message. Suppose we want to send the message SUSIE SAYS IT IS EASY. The letter S appears a lot, and so does the space character. We might want to make up a table showing how many times each letter appears. This is called a frequency table:
- Character Count
- A 2
- E 2
- I 3
- S 6
- T 1
- U 1
- Y 2
- Space 4
- Linefeed 1

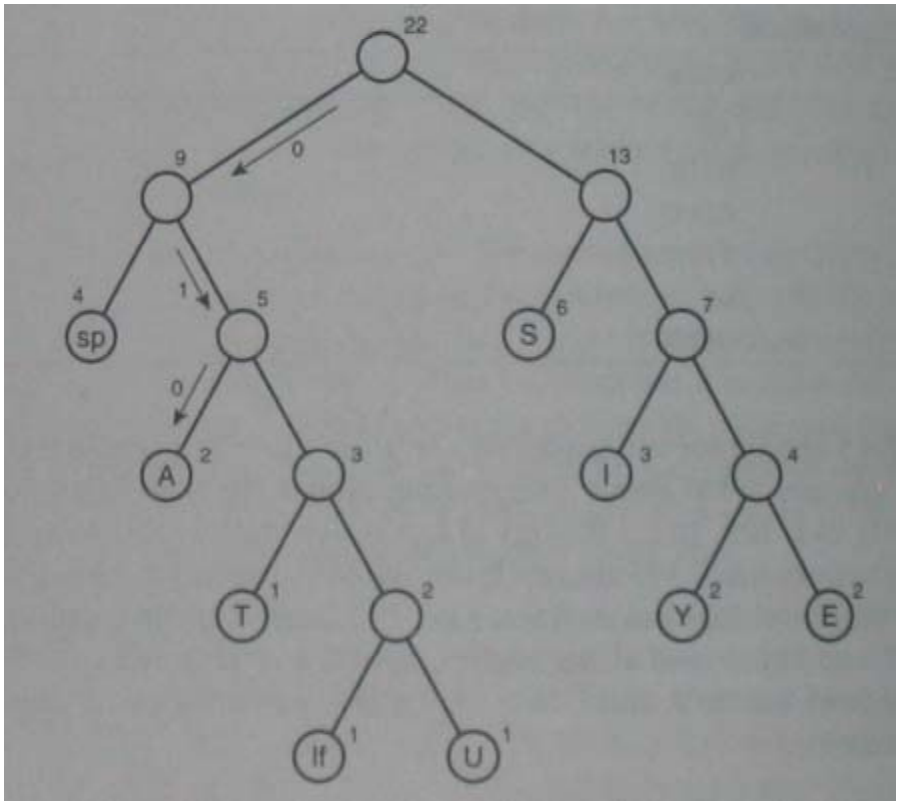
Heaps

- **The** characters with the highest counts should be coded with a small number of bits. The following table shows how we might encode the characters in the above message:
- Character Code
- A 010
- E 1111
- I 110
- S 10
- T 0110
- U 01111
- Y 1110
- Space 00
- Linefeed 01110
- Thus, the entire message is coded as:
- 10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00
- 1111 010 10 1110 01110

Heaps

- For sanity reasons we show this message broken into the codes for individual characters. Of course, in reality all the bits would run together; there is no space character in a binary message, only 0s and 1s.
- The following figure shows the Huffman Tree for the above message:

Heaps



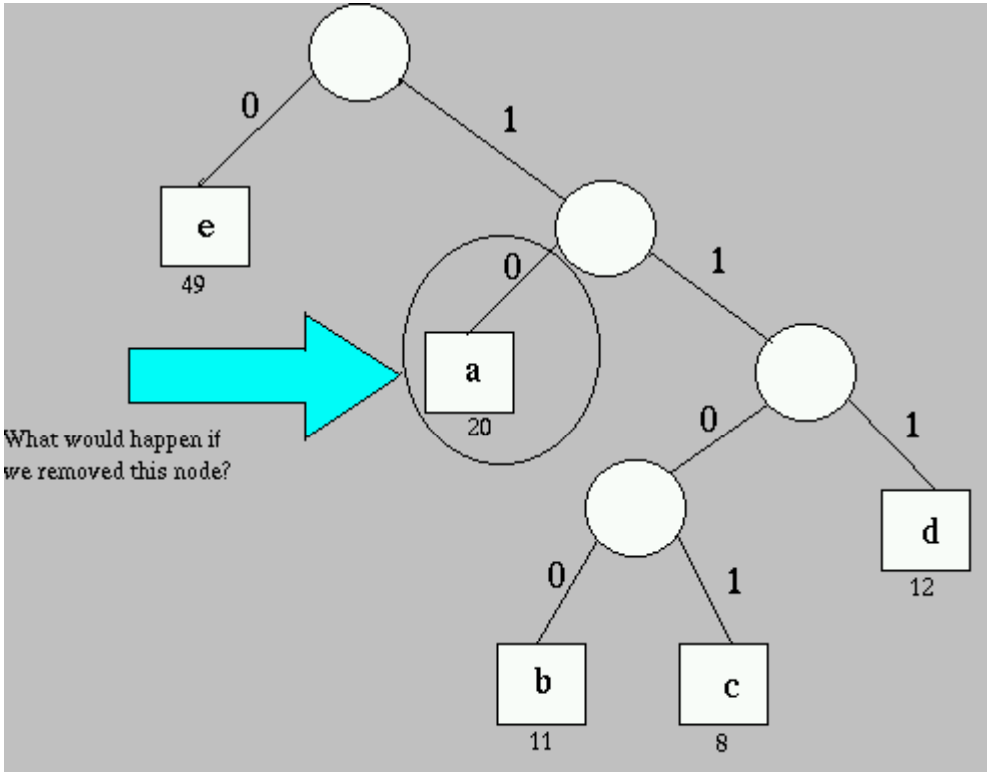
Heaps

- **Example-2:**
- Let us assume an alphabet containing the letters "a", "b", "c", "d" and "e" which are the names of leaves shown in the following table, with their corresponding frequencies:

Heaps

Character	Frequency
a	20
b	11
c	8
d	12
e	49

Heaps


0
1
e
49
0
1
a
20
What would happen if
we removed this node?
0
1
d
12
0
1
b
11
c
8

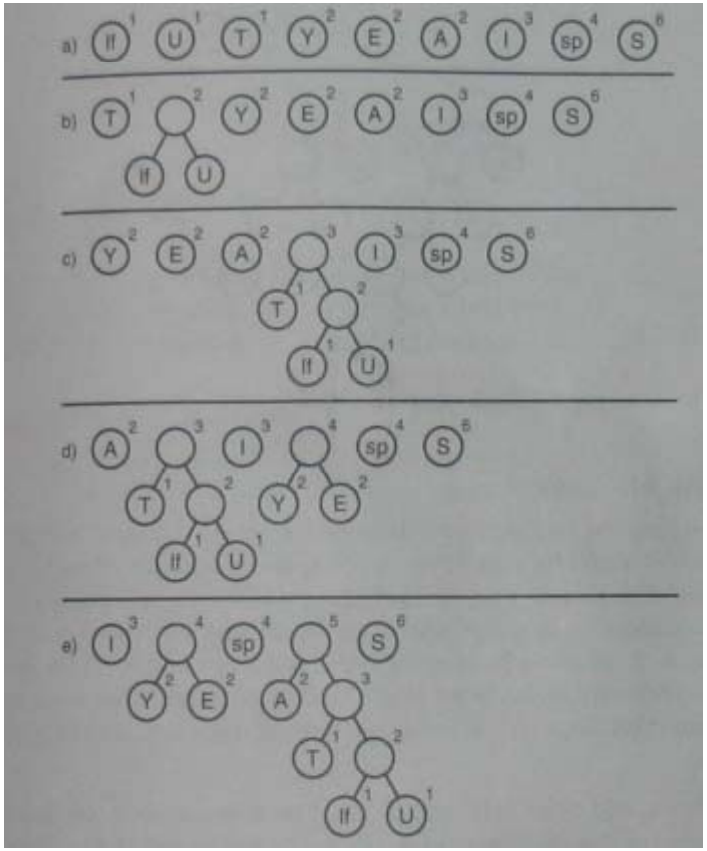
Heaps

- **Properties of Huffman trees**
- Huffman trees have the following properties:
- Every internal node has 2 children.
- Smaller frequencies are further away from the root.
- The 2 smallest frequencies are siblings.
- **Decoding with the Huffman Tree:**
- Suppose we received the string of bits shown in the preceding section. How would we transform it back into characters? We can use a kind of binary tree called a Huffman tree.
- The characters in the message appear in the tree as leaf nodes. The higher their frequency in the message, the higher up they appear in the tree. The number outside each node is the frequency. The numbers outside non-leaf nodes are the sums of the frequencies of their children. For each character you start at the root. If you see a 0 bit, you go left to the next node, and if you see a 1 bit, you go to right. Try it with the code for A, which is 010. You go left, then right, then left again, and you find the A node.

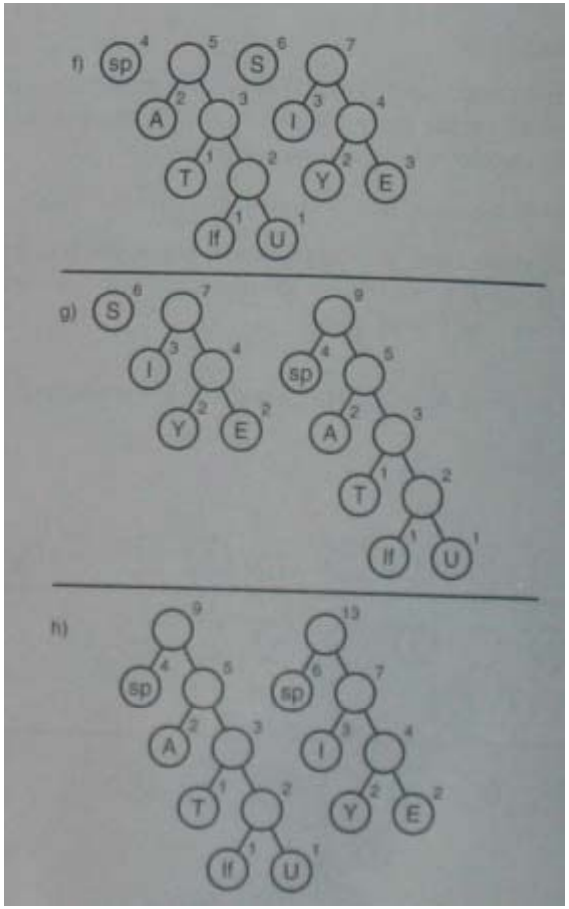
Heaps

- **Creating the Huffman Tree:**
- Here is the algorithm for constructing the tree:
- Make a node object (as seen in tree) for each character used in the message. For the message in the example, we will have nine nodes. Each node has two data items: the character and that character's frequency in the message.
- Make a tree object for each of these nodes. The node become the root of the tree.
- Insert these trees in a priority queue. They are ordered by frequency, with the smallest frequency having the highest priority.
- Now do the following:
- 1- Remove two trees from the priority queue, and make them into children of a new node. The new node has a frequency that is the sum of the children's frequencies; its character field can be left blank.
- 2- Insert this new three-node tree back into the priority queue.
- 3- Keep repeating steps 1 and 2. The trees will get larger and larger, and there will be fewer and fewer of them. When there is only one tree left in the queue, it is the Huffman tree and you are done.
- The following figures show how the Huffman tree is constructed from the example message:

Heaps



Heaps



Heaps

- **Coding the Message:**
- Now that we have the Huffman tree, how do we code a message? We start by creating a code table, which lists the Huffman code alongside each character. To simplify the discussion, let us assume that, instead of the ASCII code, our computer uses a simplified alphabet that has only uppercase letters with 28 characters. A is 0, B is 1, and so on up to z, which is 25. A space is 26, and a linefeed is 27. We number these characters so their numerical code run from 0 to 27. Our code table would be an array of 28 cells. The index of each cell would be the Huffman code for the corresponding character. Not every cell contains a code; only those that appear in the message. The following table shows the table for the example message.
- Such a code table makes it easy to generate the coded message. For each character in the original message, we use its code as an index into the code table.

Heaps

- **Creating the Huffman Code:**
- How do we create the Huffman code to put into the code table? The process is like decoding a message. We start at the root of the Huffman tree and follow every possible path to a leaf node. As we go along the path, we remember the sequence of left and right choices, recording a 0 for a left edge and a 1 for a right edge. When we arrive at the leaf node for a character, the sequence of 0s and 1s is the Huffman code for that character. We put this code into the code table at the appropriate index number. This process can be handled by calling a method that starts at the root and then calls itself recursively for each child. Eventually, the paths to all the leaf nodes will be explored and the code table will be completed.