Lecture 16

Huffman Coding:
A way to compress data using binary tree

Last time:

Binary search tree is useful because it can combine the advantages of two other data structures.

- Ordered array: Search on a binary search tree is relatively quick using binary search. After all, it is ***"binary search"*** tree.
- Linked list: Insert and delete items quickly (No need to shift other items). Binary search tree is very similar to linked list. The major difference is the fact that it has two self-referenced fields, right and left. (Linked List has next or next/prev.)

From the previous statement, we would expect most operations of binary search tree should have a good running time complexity.

But, in the last lecture, we looked at the case that binary search tree can degenerate to linked list when all of the input values are already sorted or inversely sorted, etc.

For that reason, binary search tree's worst-case running time complexity for major operations are **NOT O(log n).** ***It is O(n).***

To remedy this issue, there have been many efforts to have other tree data structures called self-balanced trees. Red-Black tree and AVL tree are two major examples of self-balanced trees.

But, due to the time limit, we cannot look at those self-balanced trees. I strongly encourage you to look into them on your own. Taking this course, you are ready to explore other advanced data structures.

Today, we will look at Huffman Coding, another usage of binary trees.

Binary trees are not always search trees!

Today, we will discuss an algorithm that uses binary tree to compress data.

Since what we are trying to do is to show an application of a data structure, we will mainly focus on understanding its concept.

Let's take a look at the standard ASCII code again but a little bit closer this time!

Character	Decimal	Binary
A	65	1000001
B	66	1000010
C	67	1000011
…	…	…
X	88	1011000
Y	89	1011001
Z	90	1011010

As you can see each character takes 7 bits.

Fixed Width Encoding
Standard ASCII is a fixed width encoding and each character is encoded with 7 bits.

How many different codes can we have with 7 bits? 128

From a different perspective, to encode alphabet A-Z (26 only uppercase letters) using a fixed width encoding, how many bits would we need?

$2^5 = 32$

Assuming that we are using a fixed width encoding scheme where we use *n bits* for a character set and we want to store or transmit *m characters*, we need *m*n bits for the entire file*.

The question is *"can we do better?"*

In 1952, David Huffman thought about:

> Some characters are used more often than other characters. If I can find a way to use as few bits as possible for most common letters, I can reduce the amount of bits to store or transmit data eventually.

There is a basic rule though: No code can be the prefix of any other code.

Being very specific, let's take an example sentence.
SUSIE SAYS IT IS EASY

Frequency Table

Character	Count (Frequency)
S	6
Space	4
I	3
A	2
E	2
Y	2
T	1
U	1
Linefeed	1

From this, we can generate a Huffman tree with the following steps:

1) Create nodes that have two data items of each character and its frequency.
2) Combine lowest two frequency nodes into a tree with a new parent with the sum of their frequencies.
3) Combine lowest two frequency nodes, including the new node we just created, into a tree with a new parent with the sum of their frequencies.
4) Repeat the previous step until we have one tree with all nodes are linked.
5) Label all left branches (edges) with 0 and all right branches (edges) with 1.

Build a Huffman tree using the frequency table of the previous page.

<Step 1>
lf(1) U(1) T(1) Y(2) E(2) A(2) I(3) SP(4) S(6)

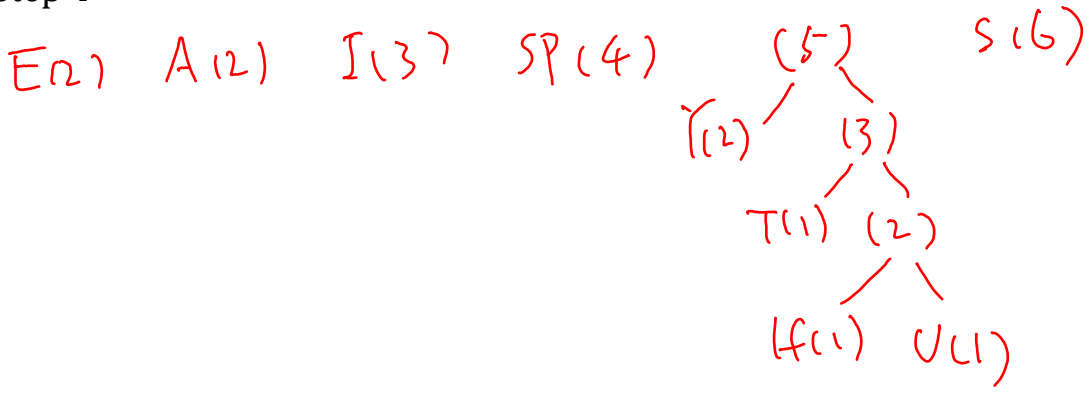
<Step 2>
T(1) (2) Y(2) E(2) A(2) I(3) SP(4) S(6)

```
      /   \
   lf(1)  U(1)
```

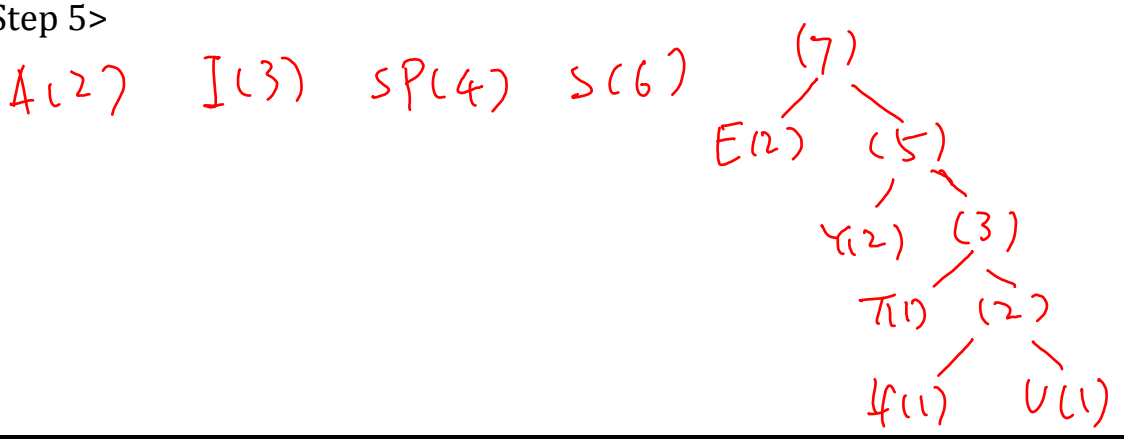
<Step 3>
Y(2) E(2) A(2) (3) I(3) SP(4) S(6)

```
          /   \
       T(1)   (2)
             /   \
          lf(1)  U(1)
```

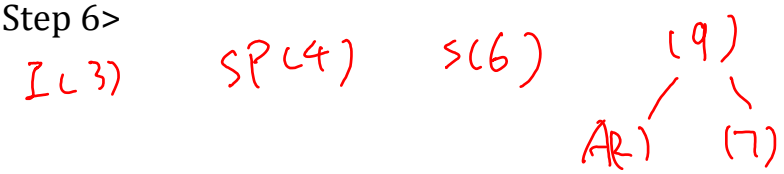
<Step 4>



<Step 5>



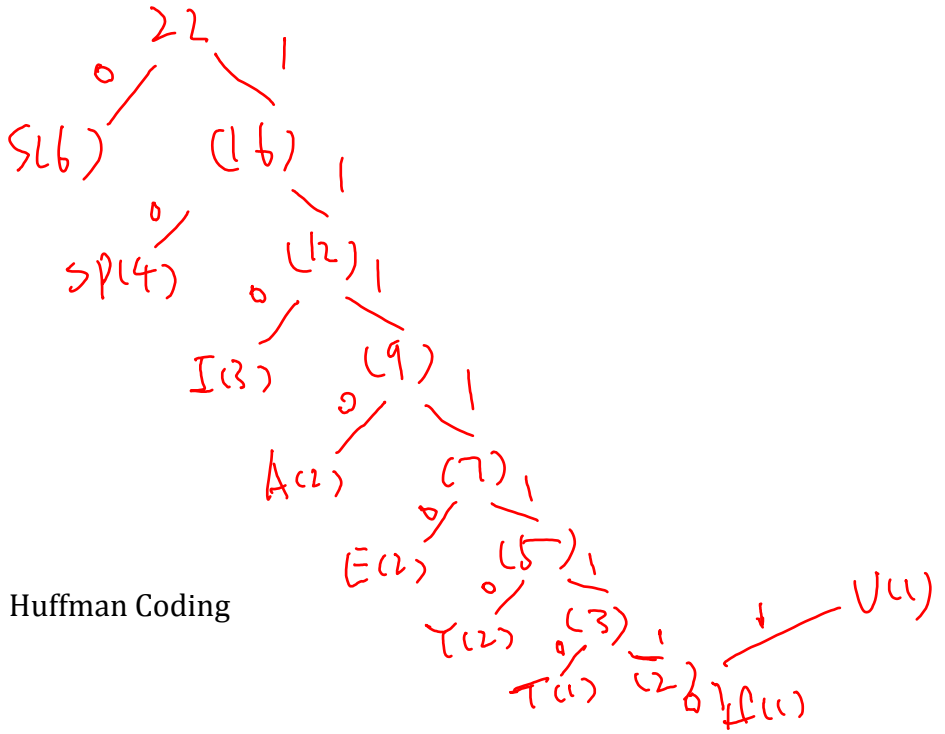
<Step 6>



<Step 7>

<Step 8>

<Step 9: The final Huffman tree>



Now, what is the Huffman code for each character?

Character	Huffman Code
S	
Space	
I	
A	
E	
Y	
T	
U	
Linefeed	

Hufman code for "SUSIE SAYS IT IS EASY"

Character	Huffman Code	Number of bits
S		
U		
S		
I		
E		
SP		
S		
A		
Y		
S		
SP		
I		
T		
SP		
I		
S		
SP		
E		
A		
S		
Y		
lf		

Comparison with fixed width encoding for "SUSIE SAYS IT IS EASY"

	Total number of bits
Fixed width encoding	
Huffman coding	

A brief history in relation to Huffman Coding

David Huffman developed the coding when he was a Ph.D student at MIT taking a course on information theory taught by Robert Fano. The professor had given his students a choice: a final exam or a term paper on something related to information theory. He suggested a number of possible topics, one of which was finding the most efficient binary encodings: while Fano himself had worked on the subject with his colleague Claude Shannon, it was not known at the time how to efficiently construct optimal encodings.

Huffman struggled for some time to make headway on the problem and was about to give up and start studying for the final when he hit upon a key insight and invented the algorithm that bears his name. Huffman outdid his professor, making history. In addition, he got an "A" for the course.

Today, the coding is used in producing ZIP files and as part of several multimedia codecs like JPEG and MP3.