ICSI 403
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 09/10 – Huffman Encoding and Huffman Trees

J Marques de Carvalho

A Completely New Tree Application

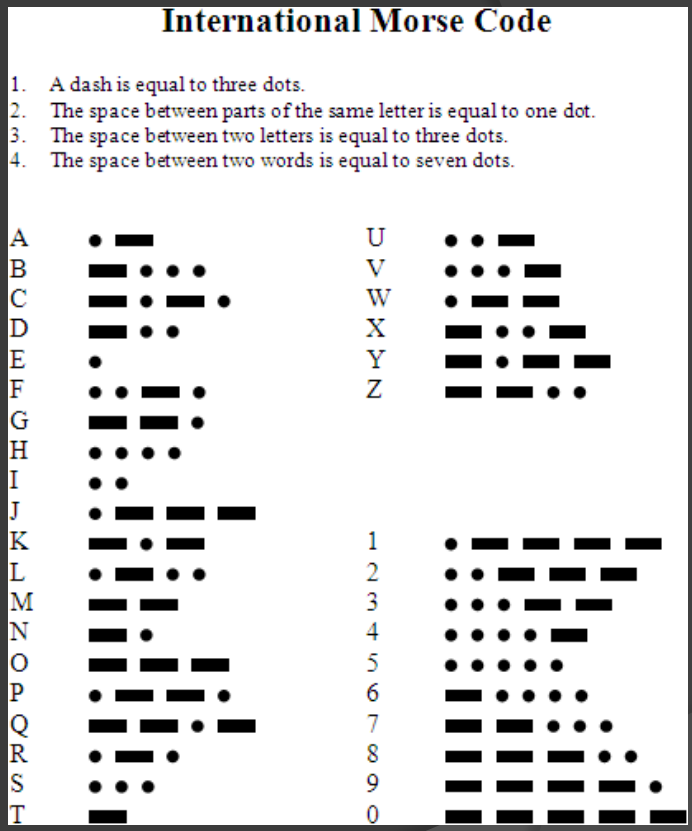
- Sometimes it's not just the *data* in the node that's important, but the *PATH* to the data.
- Why does Manhattan have area code 212?
- Los Angeles (just *part* of the city) has 213.
- Chicago (just *part* of the city) has 312.
- *All* of Alaska has 907.
- When area codes were devised, we had rotary dial phones, and ALL area codes had either 0 or 1 as the middle digit.

Rotary Telephone Dial

- No pushbuttons
- It takes longer to dial 907 (Alaska) than to dial 212 (Manhattan)
- 212 is the fastest 3-digit number (with a 1 or 0 as the middle digit) you can dial

A Related System – Morse code

- Used by telegraphers in mid-1800's before long-distance telephone lines existed
- Letters were transmitted using sequences of dots (short pulses) and dashes (long pulses)
- Why was "E" assigned the one-dot code?



Frequency Of Occurrence (1)

- We encode data in 8-bit bytes (using one byte per character), because computers work well with fixed-size characters
- What if we could encode data with variable-length “bytes”?
- If we assigned shorter bit strings to the characters that occur most often (and longer bit strings to the ones that occur least often), could we save space overall?
  - Does the "savings" outweigh the "cost" ?

Frequency Of Occurrence (2)

- There are 26 letters in the English alphabet, but 12 of them comprise about 80% of the total usage
  - Obviously, we have to consider huge bodies of text to get the averages
  - Specific documents, or text related to a specific area would have different frequency of occurrence patterns (as would different languages, of course)
    - Documents from the Oxford University website probably have many more “X” and “F” characters than we would normally expect in English prose.

Huffman Coding

- Huffman Coding (David Huffman, MIT, 1952)
- Uses variable-length bit codes to represent characters
- Requires that we know (up-front) the relative frequency of occurrence for each character.
- Once we build the “Huffman tree”, we can encode and decode the characters/bit strings
- A way to implement lossless compression (but not as much as ZIP, RAR, or 7-zip).
- Used in JPEG images

So How Does It work?

- Start with the table of relative frequencies
- Let's simplify things a bit -- rather than encoding the entire alphabet, let's consider something shorter
  - "THIS IS AN EXAMPLE OF A HUFFMAN TREE"
  - This message consists of 16 different characters:

    {T, H, I, S, space, A, N, E, X, M, P, L, O, F, U, R}
  - Next, we need to count how many times each appears (compute the frequency of occurrence) in our body of text.

Compute Frequency Of Occurrence

- “THIS IS AN EXAMPLE OF A HUFFMAN TREE”
- {T, H, I, S, space, A, N, E, X, M, P, L, O, F, U, R}

Character	Occurs		Character	Occurs
T	2		X	1
H	2		M	2
I	2		P	1
S	2		L	1
space	7		O	1
A	4		F	3
N	2		U	1
E	4		R	1

Sort The Table (Descending)

Character	Occurs		Character	Occurs
space	7		S	2
A	4		T	2
E	4		L	1
F	3		O	1
H	2		P	1
I	2		R	1
M	2		U	1
N	2		X	1

- The remaining question:
  - Do we save enough bits by representing the space, "A", and "E" with something short to pay for having to represent "R", "U", and "X" with something longer?

Huffman Trees

- Huffman proved (mathematically) that, on a single-character basis, this method was optimal (ZIP, RAR, 7-Zip all consider more than one character at a time)
- The Huffman process is based on building a tree from the frequency of occurrence information
  - This IS a binary tree, but it is NOT a Binary Search Tree – the BST property is not required to hold at any particular point in the tree!

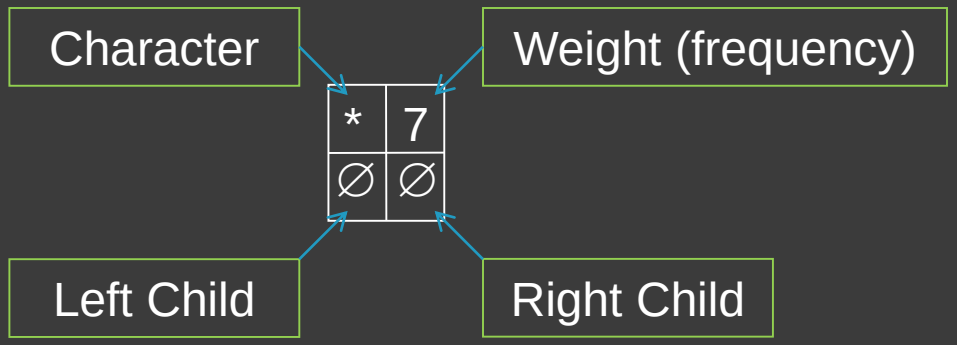
Use This Data To Build A Binary Tree

- This is a special type of binary tree
  - The *leaves* contain letters *and* “weights”
  - The *non-leaf* nodes contain only “weights”
  - How do we tell a leaf from an internal node?

```
struct treenode {
    char symb;
    int  weight;
    treenode *LCH;
    treenode *RCH;
}
```

Build The Binary Tree (2)

- Make a leaf node for every character
- The frequency of occurrence becomes the weight for a leaf
- We'll use "*" to represent "space"

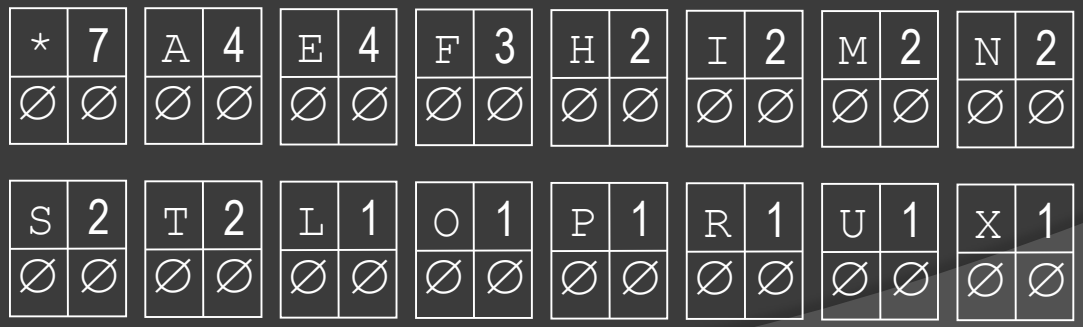


- Repeat (make a node) for *every* character
  - This gives us an *array of nodes*

Make A Leaf Node For Every Character

Character	Occurs		Character	Occurs
space	7		S	2
A	4		T	2
E	4		L	1
F	3		O	1
H	2		P	1
I	2		R	1
M	2		U	1
N	2		X	1

Hint: A single node is also a tree which just happens to have no children

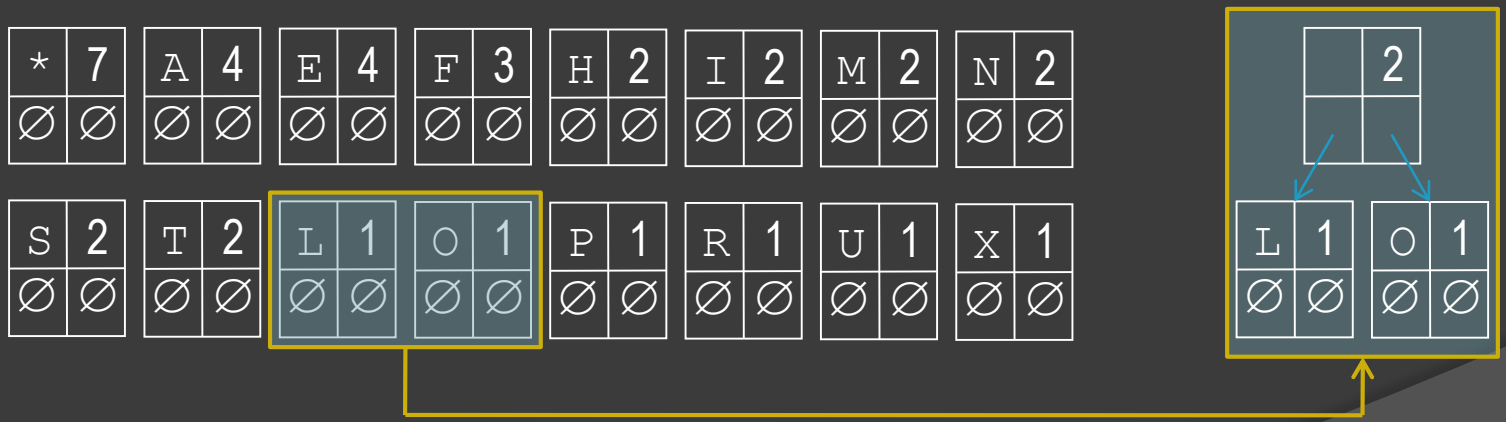


Build The Huffman Tree: Overview

- Next, we take our leaf nodes, and start building the tree from the bottom-up
- We built the BST from the top-down
- How? We said that in a binary tree, all accesses start at the root
  - How can we build a tree starting with a leaf?
  - Simple – gather two leaves, create them a parent node, and then we have a tree – the parent node is the root of the subtree with the two leaves
  - Then repeat

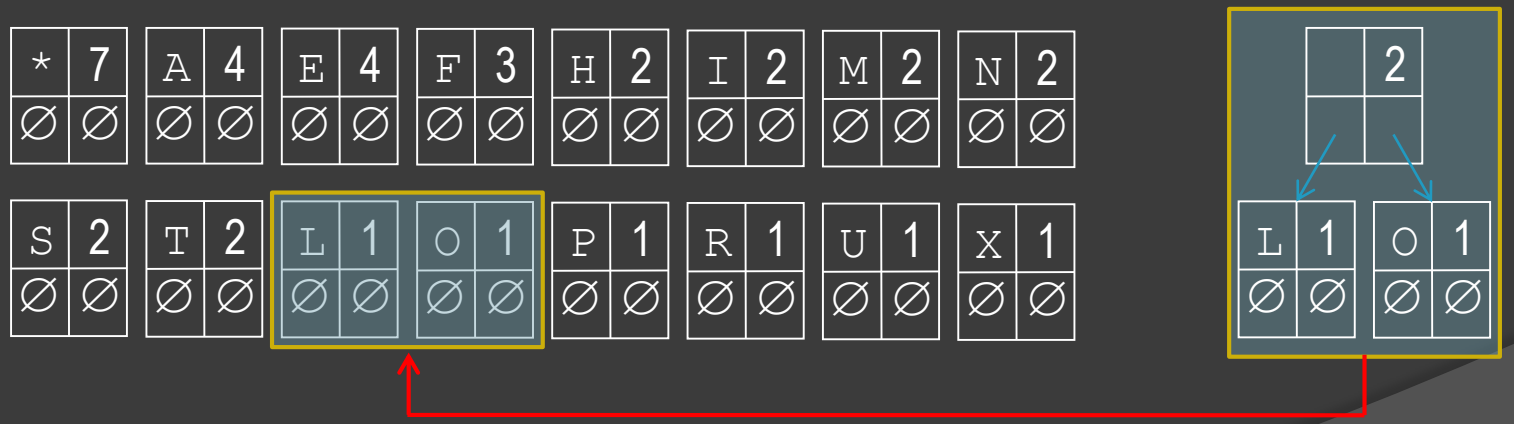
Start Building Up The Tree (1)

- Take the 2 items with the *lowest* weight and make them children of a *new* internal node. If there are more than 2 items with this lowest weight, pick <u>any</u> 2
- The sum of their weights becomes the weight of the internal node (which has no character)



Start Building Up The Tree (1)

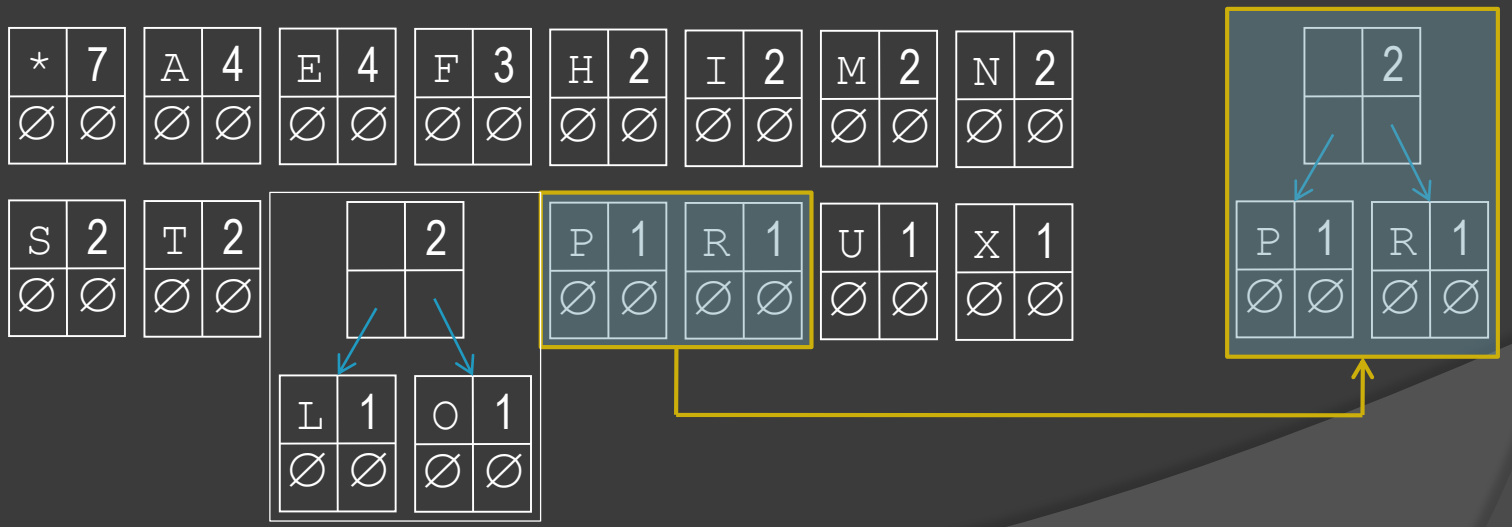
- Replace the two leaves (in our list) with the subtree
- We started with 16 items (16 characters)
- Now we have 15 items: 14 leaf nodes, and one tree containing 2 characters



- Repeat!

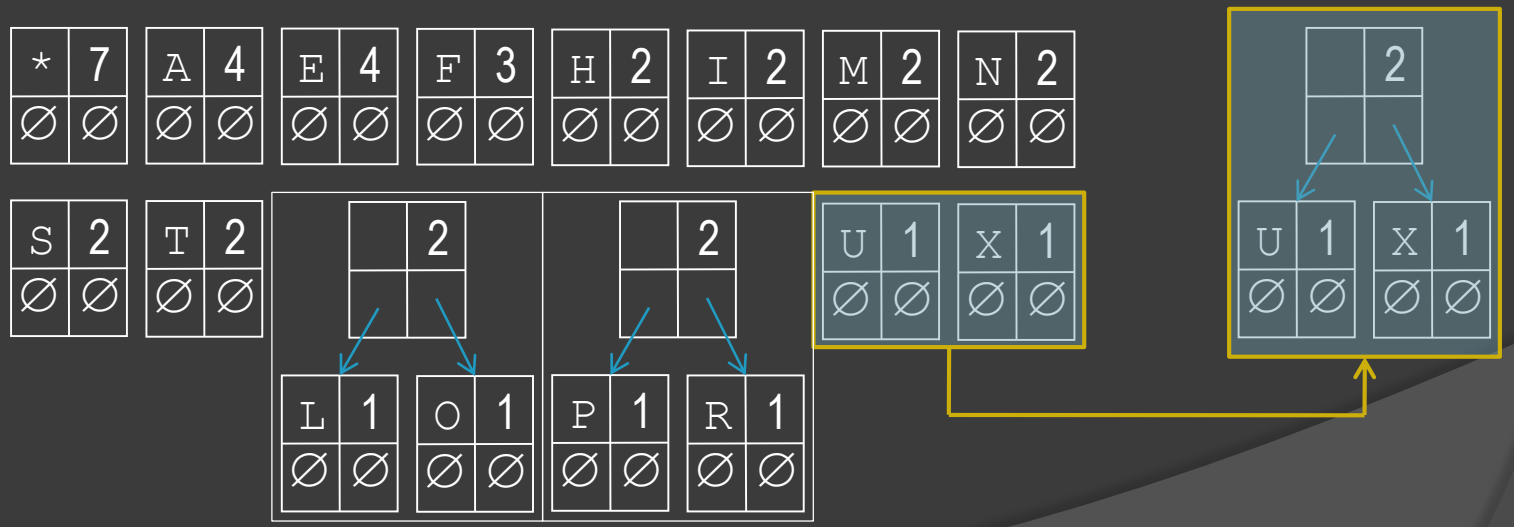
Start Building Up The Tree (2)

- Take the 2 items with the lowest weight, and make them children of a new internal node. If there are more than 2 items with this lowest weight, pick any 2
- The sum of their weights becomes the weight of the internal node (which has no character)



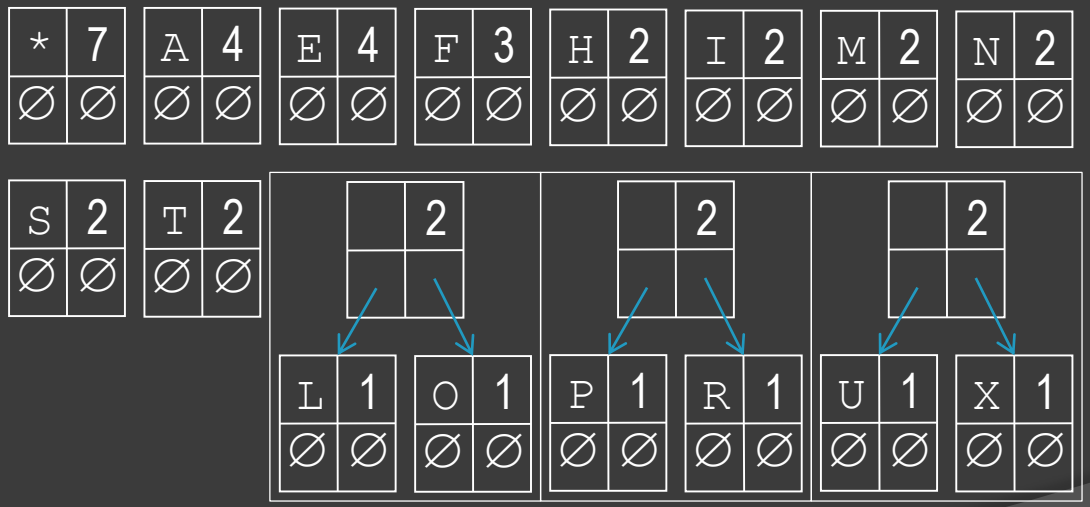
Start Building Up The Tree (3)

- Take the 2 items with the lowest weight, and make them children of a new internal node. If there are more than 2 items with this lowest weight, pick <u>any</u> 2
- The sum of their weights becomes the weight of the internal node (which has no character)



Start Building Up The Tree (4)

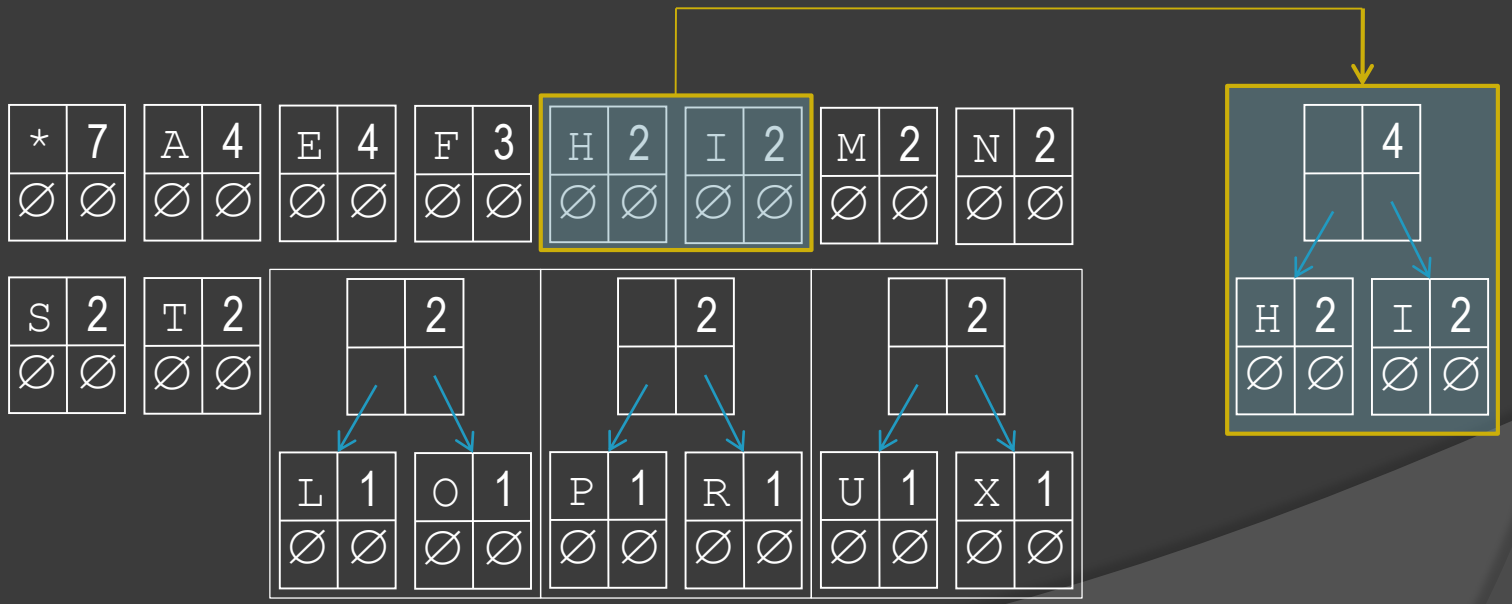
- After three applications of “group the two lowest-weight items into a tree whose parent has their combined weight”, we have what is shown below
  - When we group two items together by making them children of a new node, the new node becomes an item that replaces the two single ones – we never consider those leaves again on their own



- Repeat!

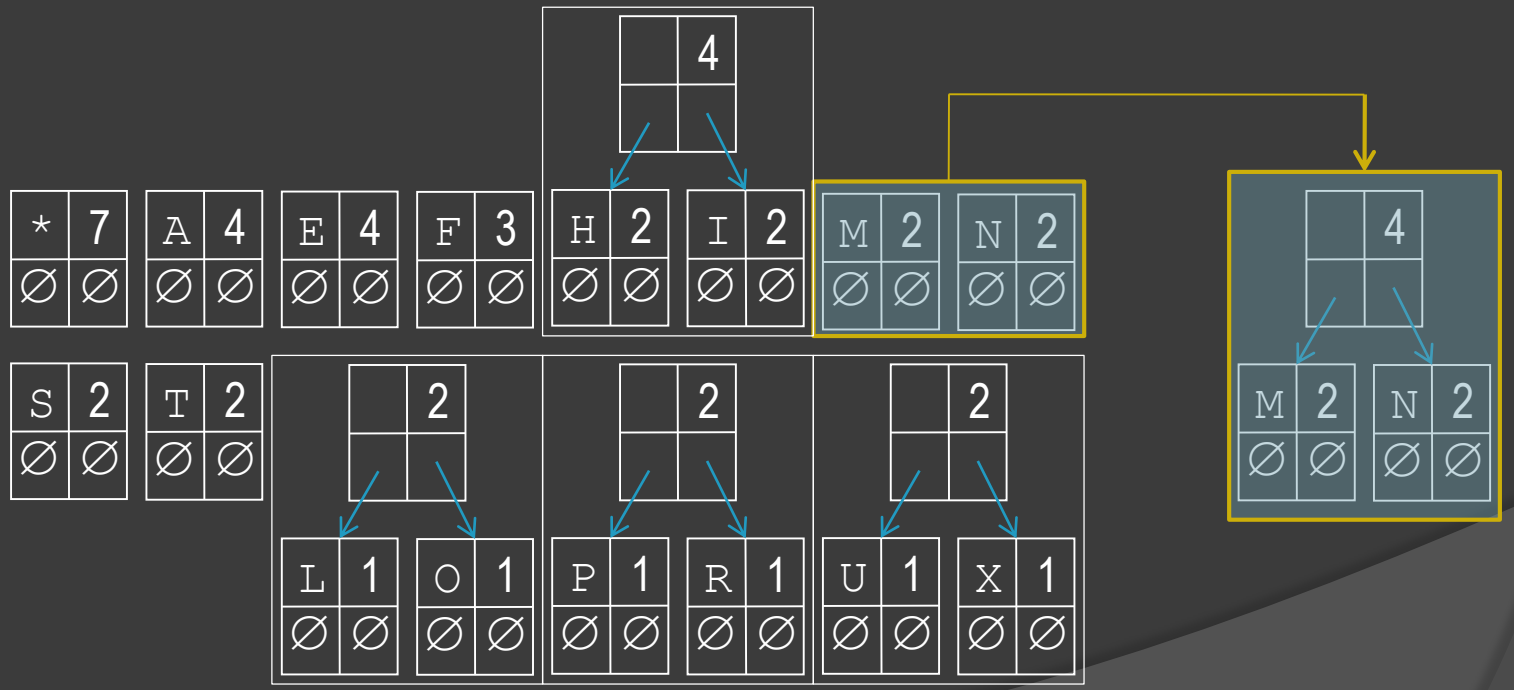
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



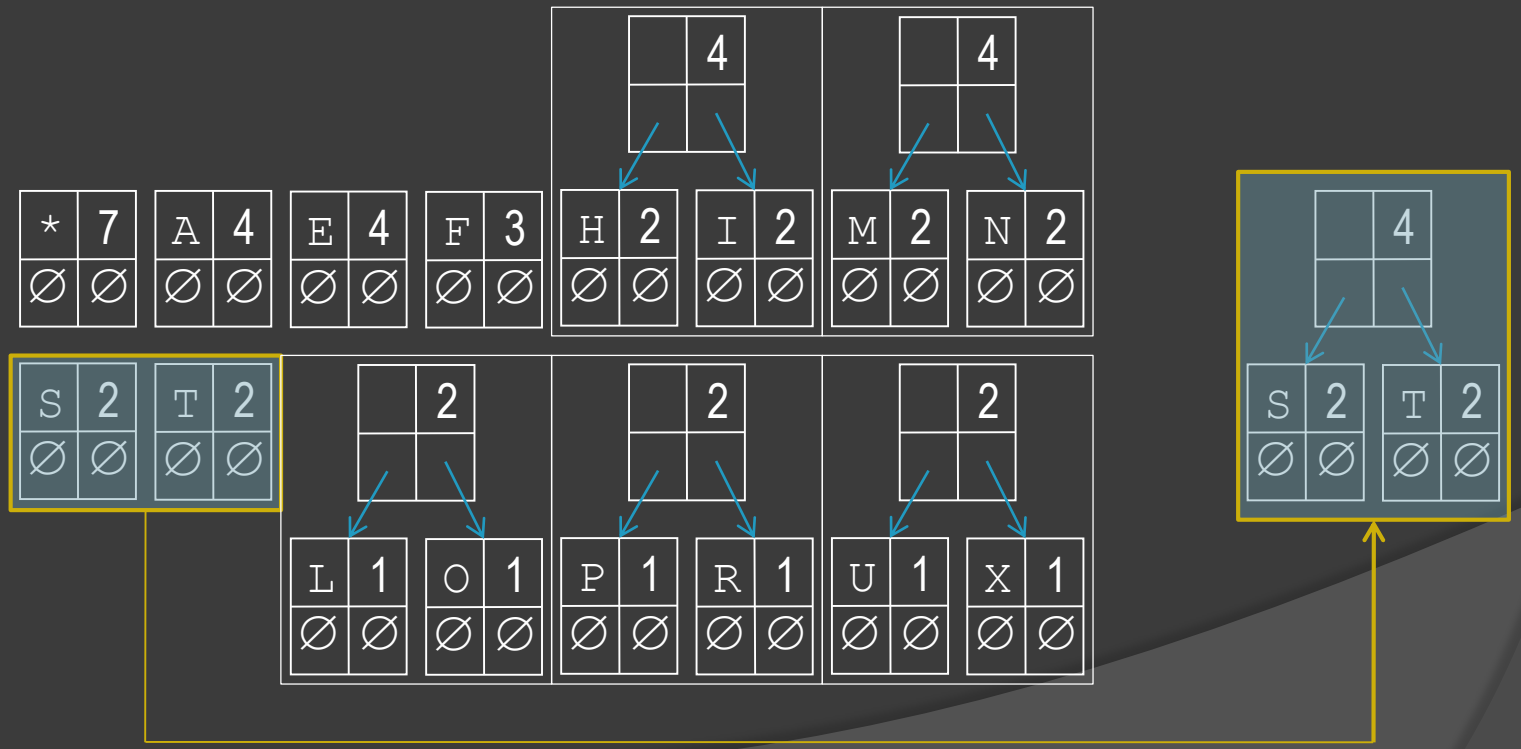
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



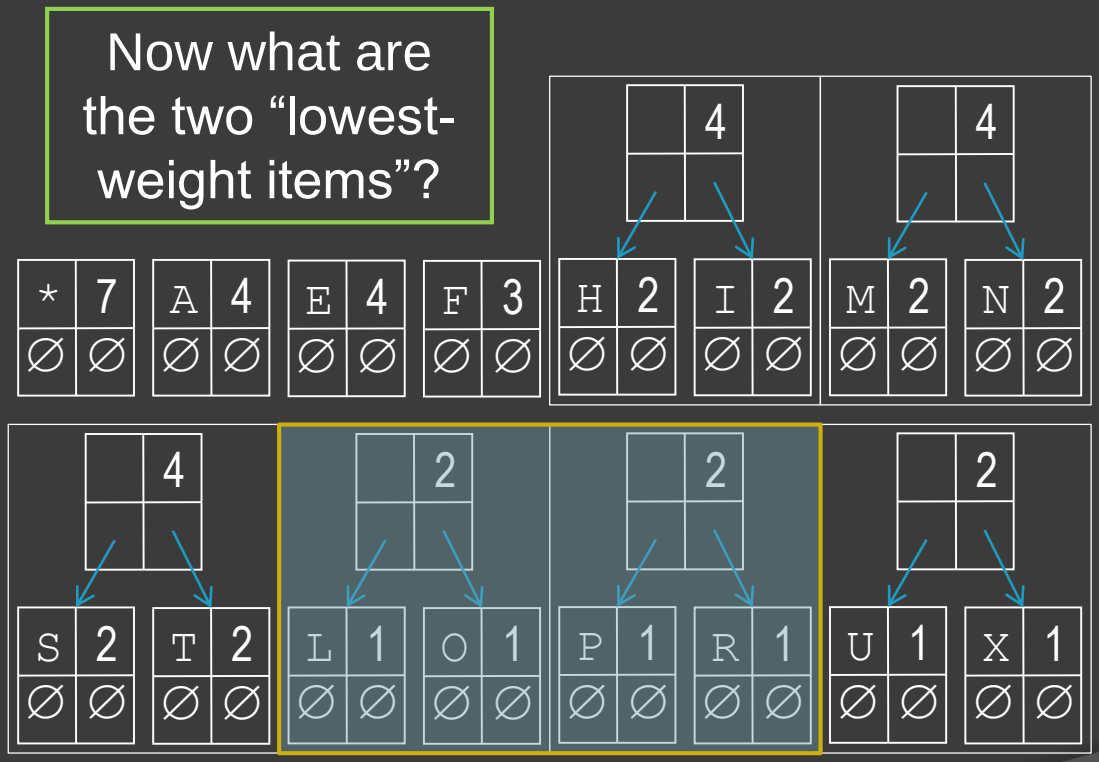
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



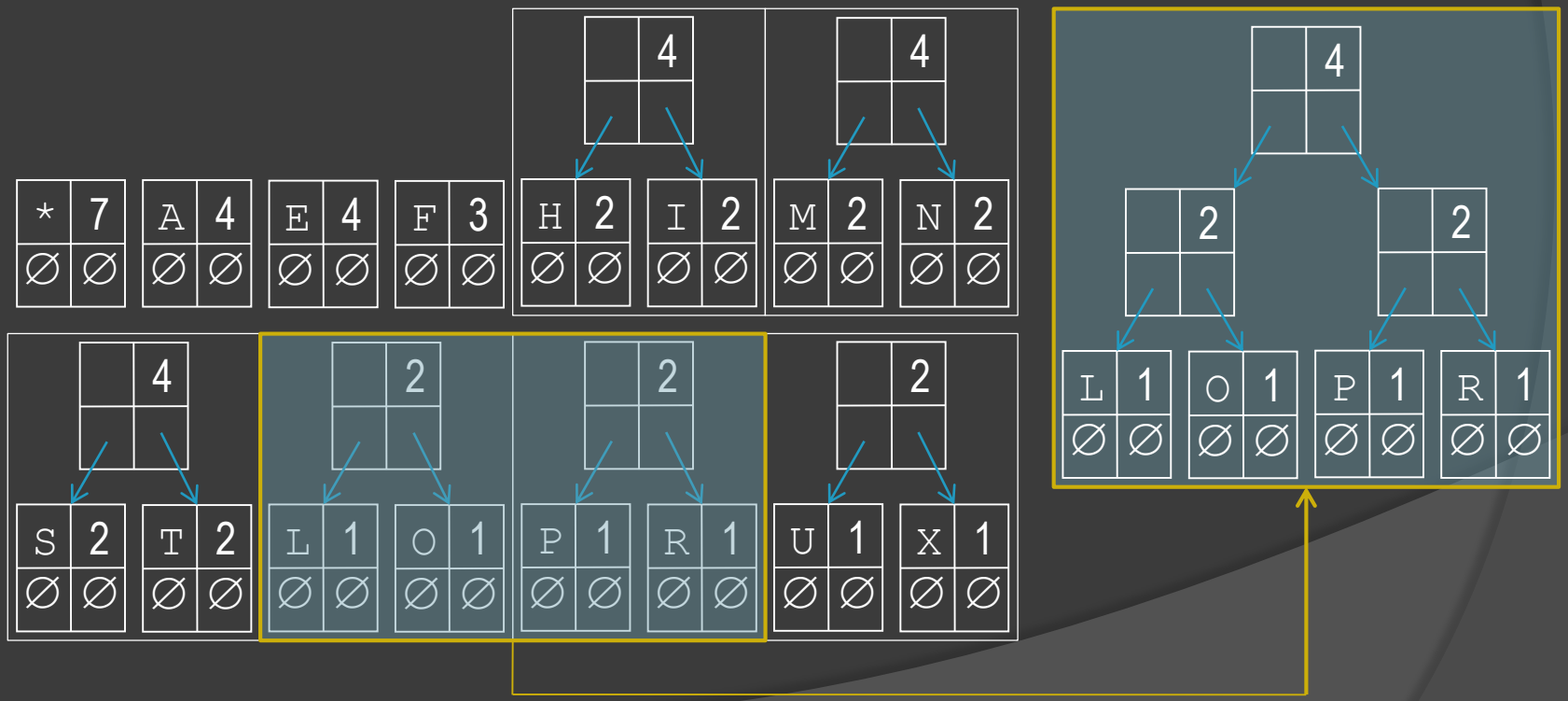
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



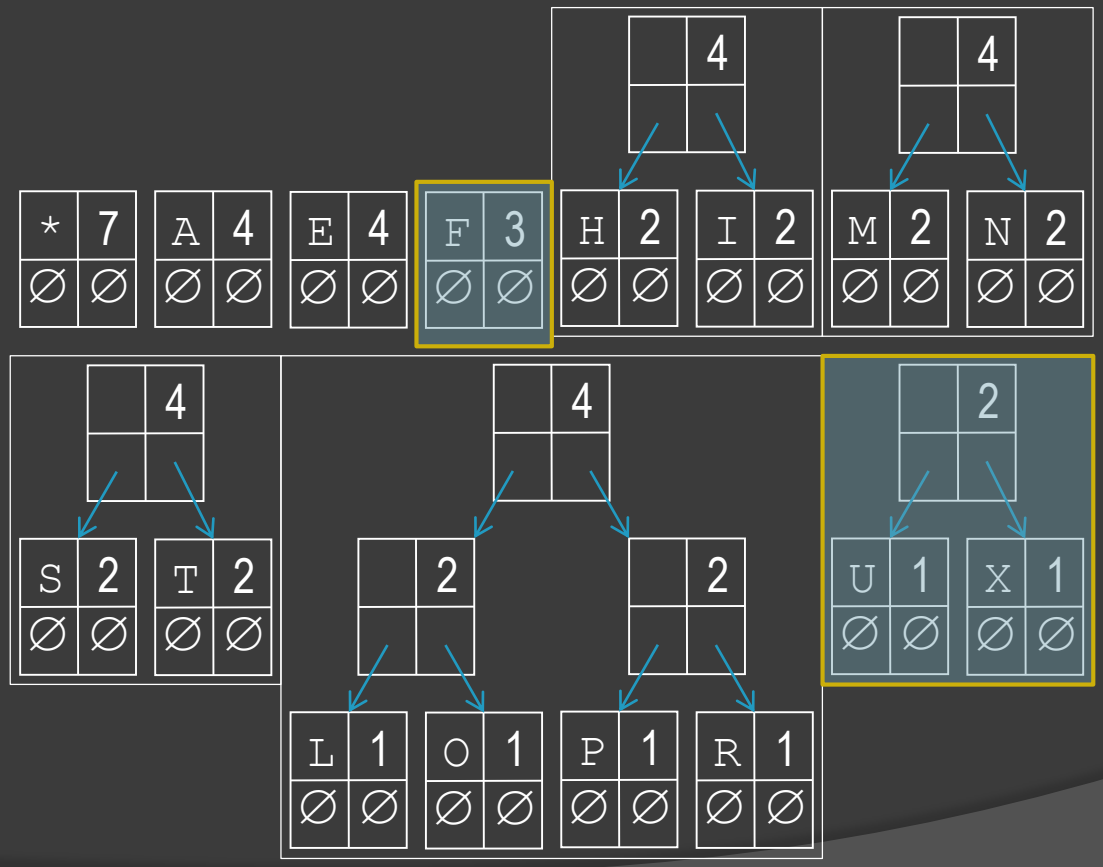
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



Continue Building The Tree

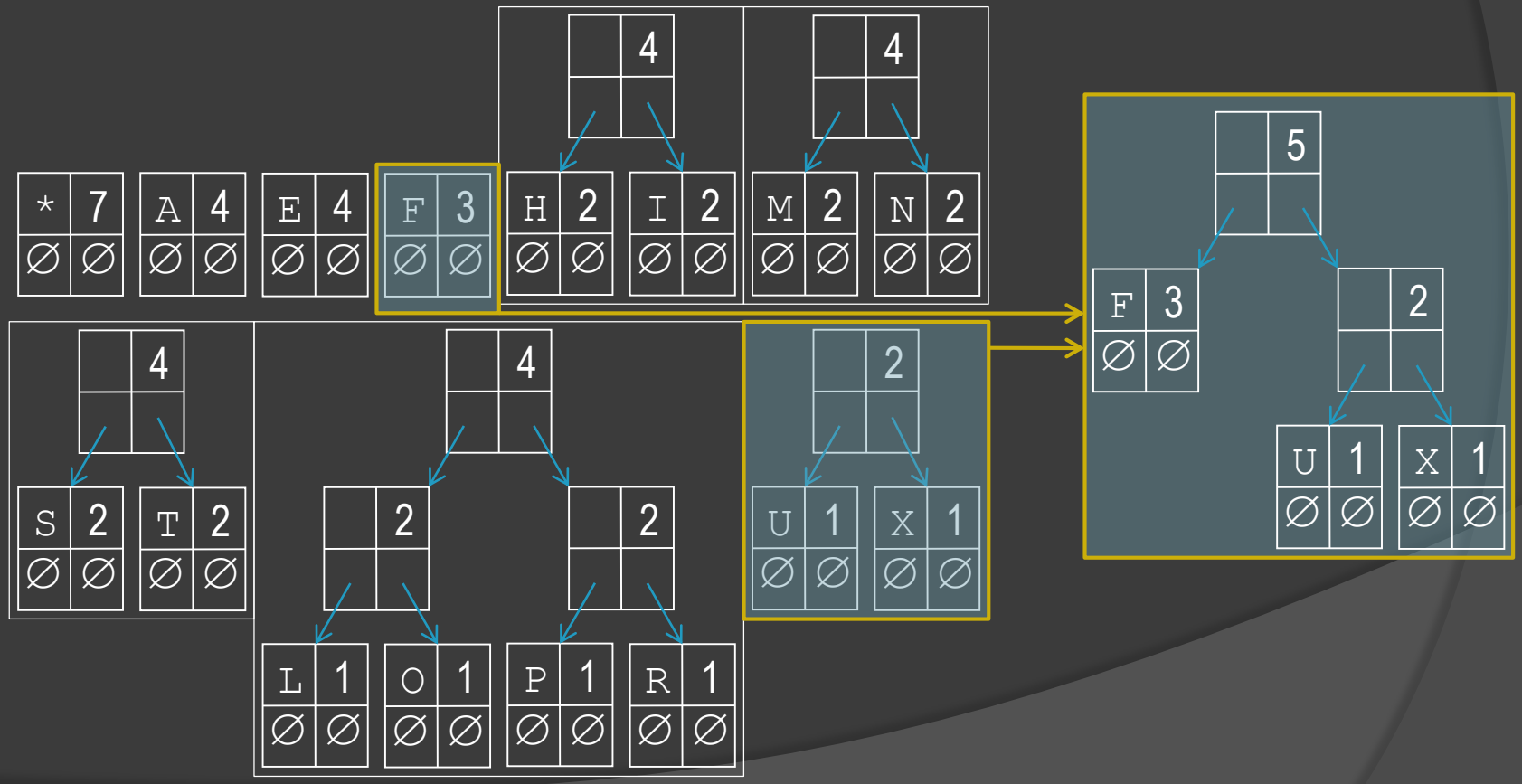
- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



Now what are the two “lowest-weight items”?

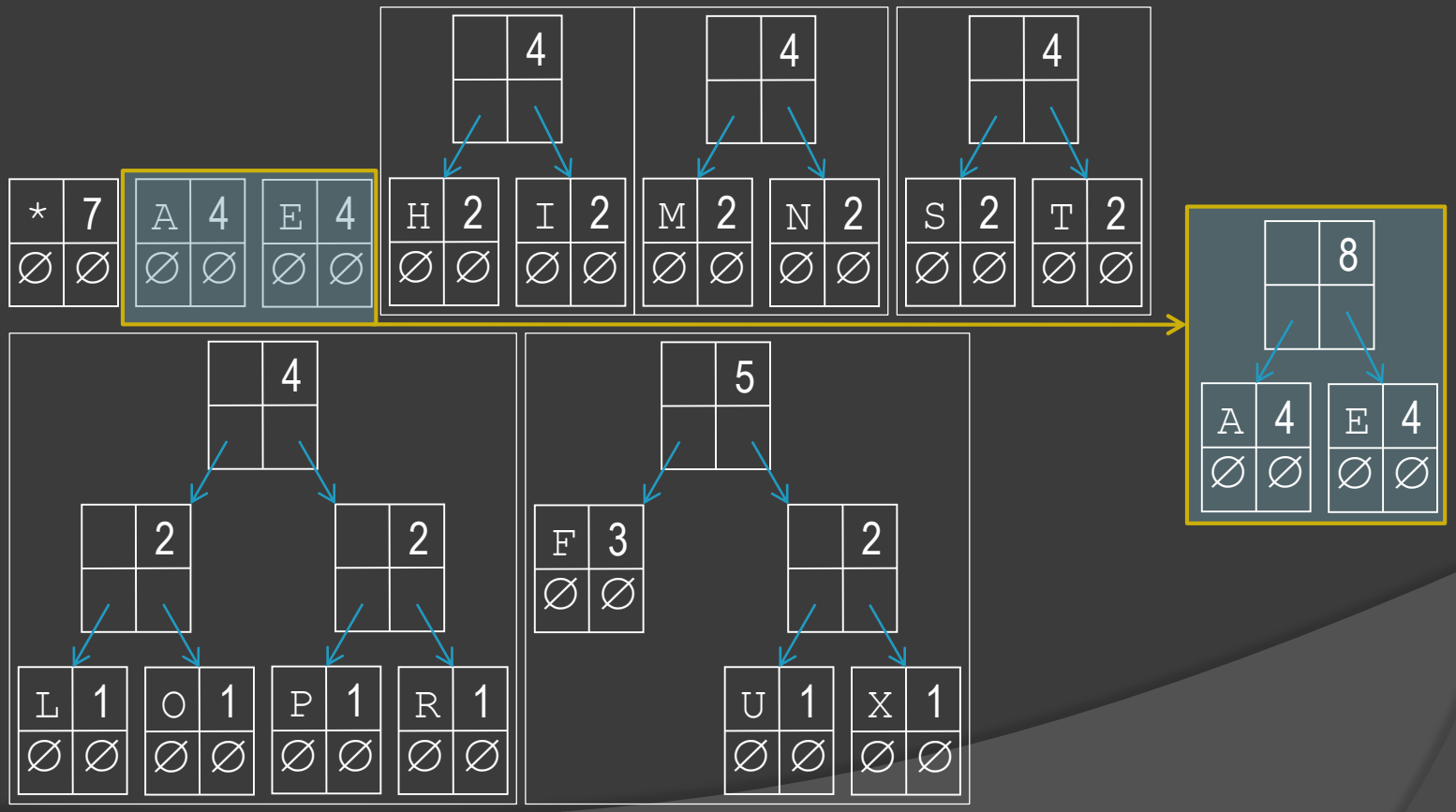
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



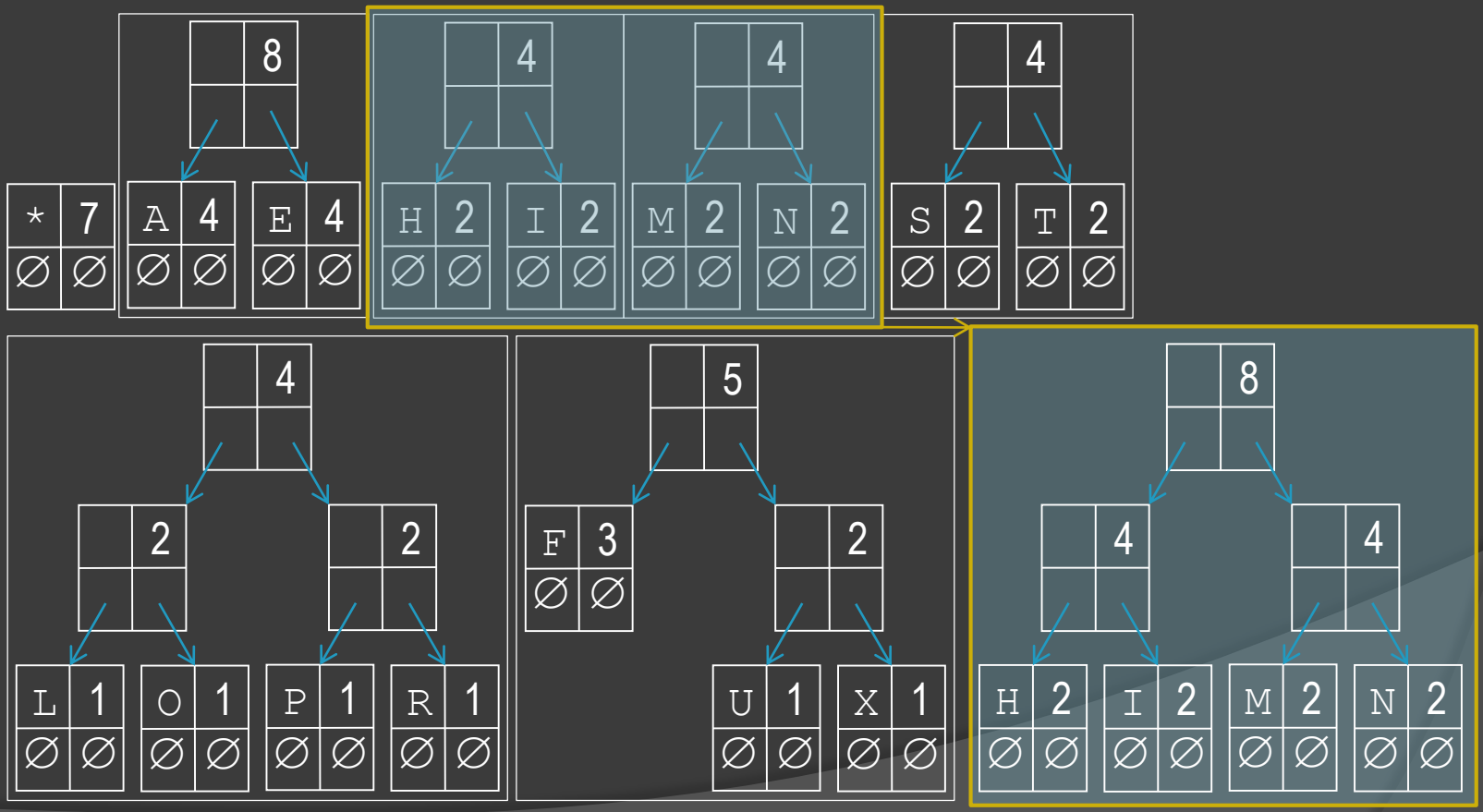
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



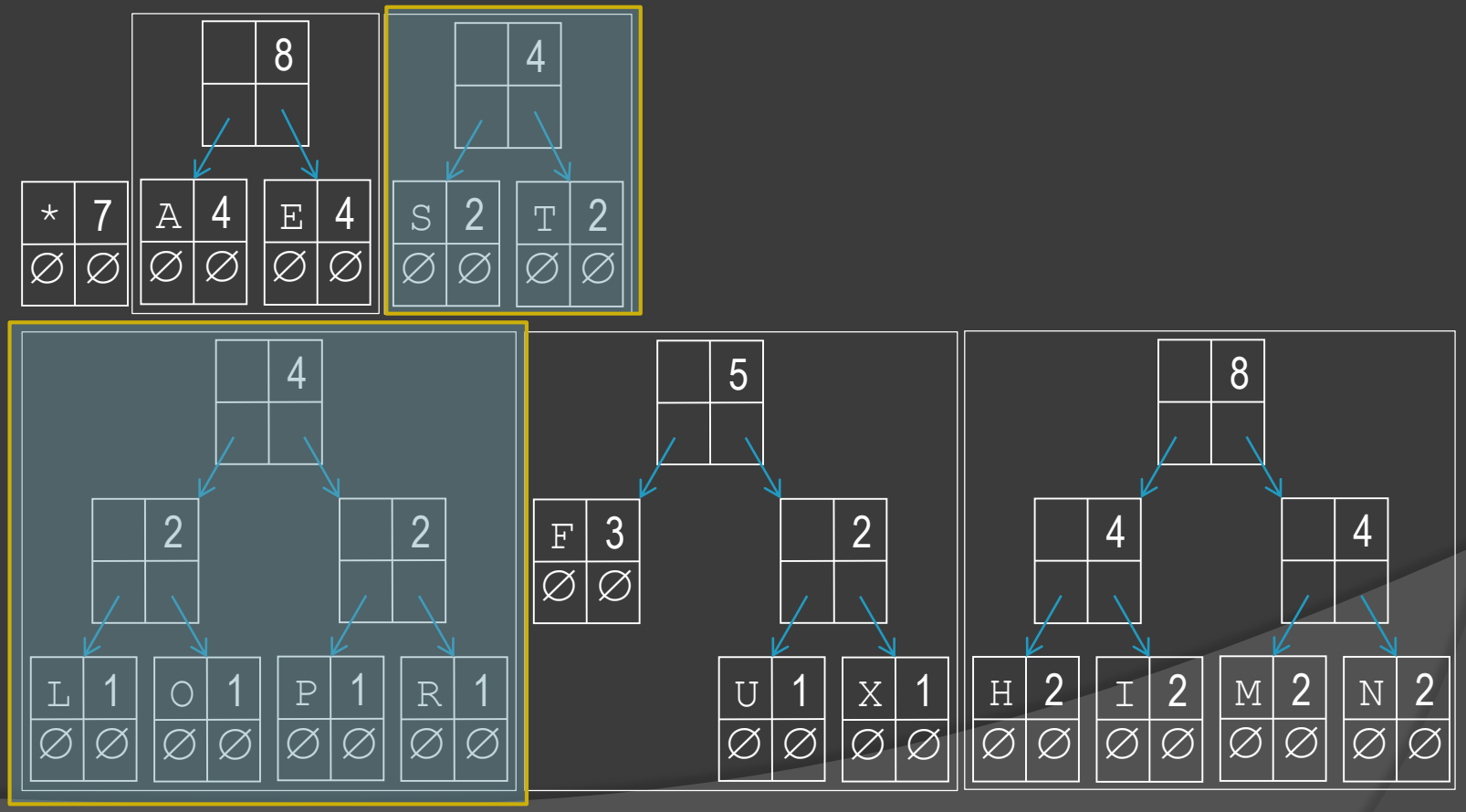
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



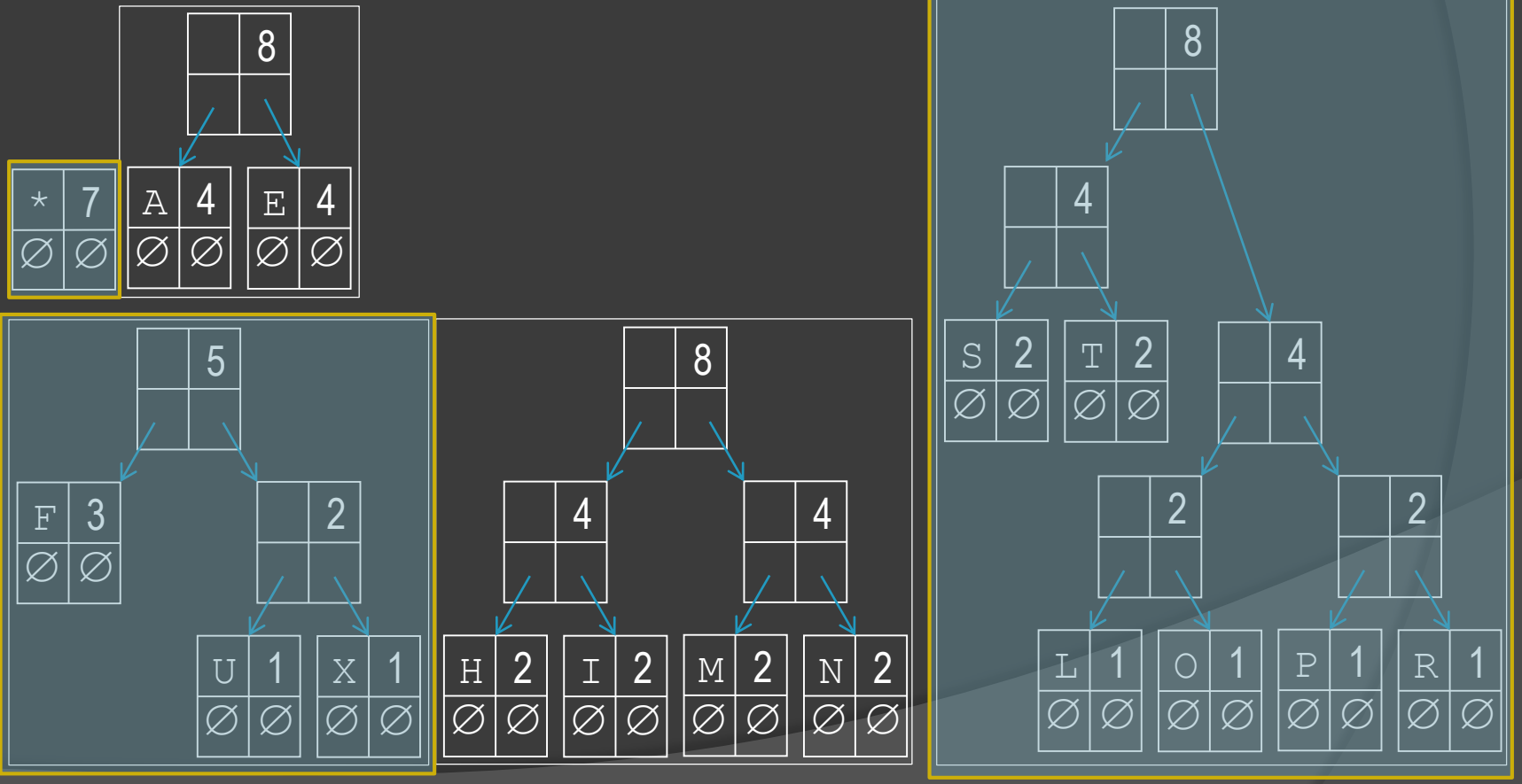
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



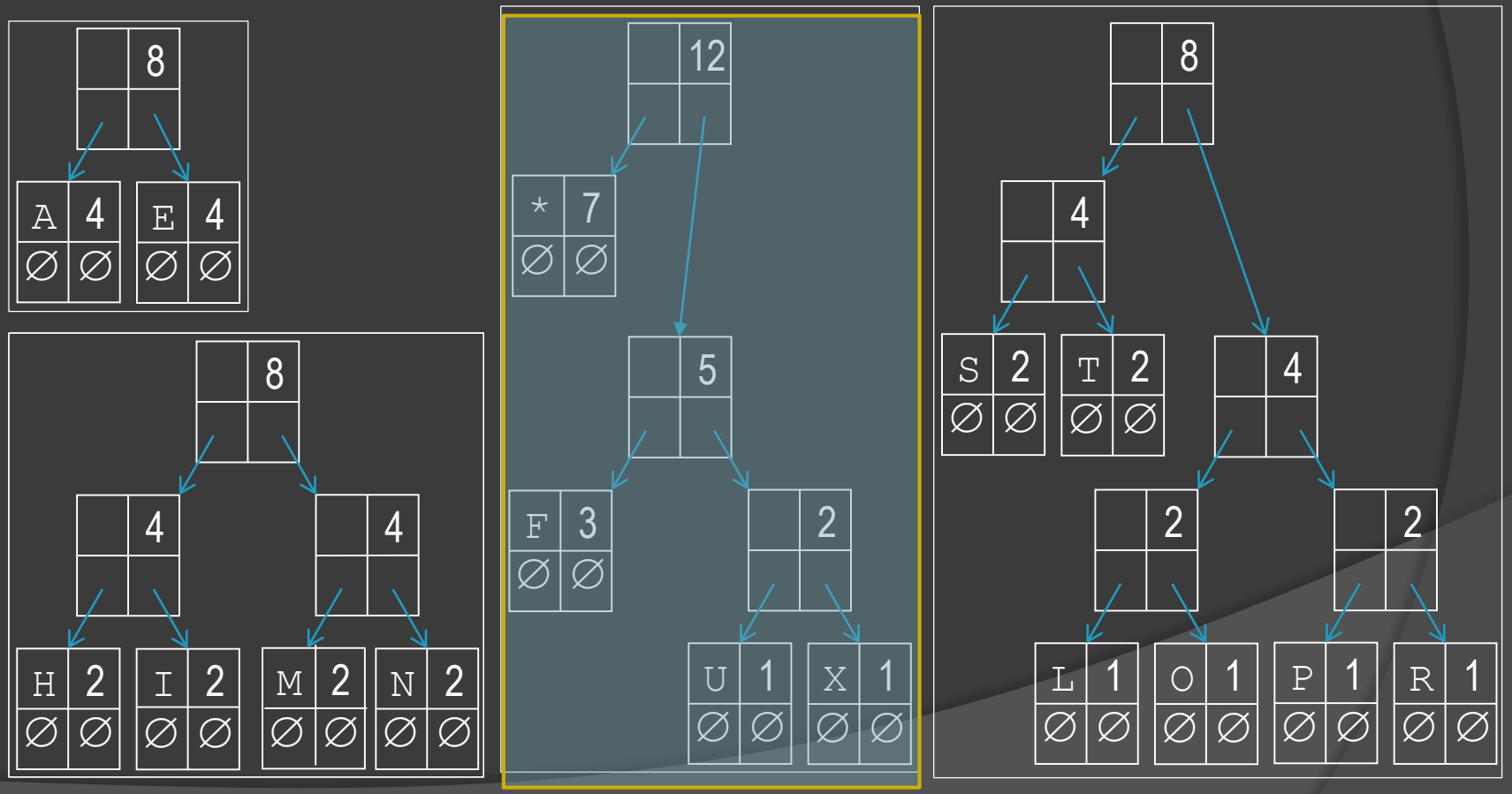
Continue Building The Tree

- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:

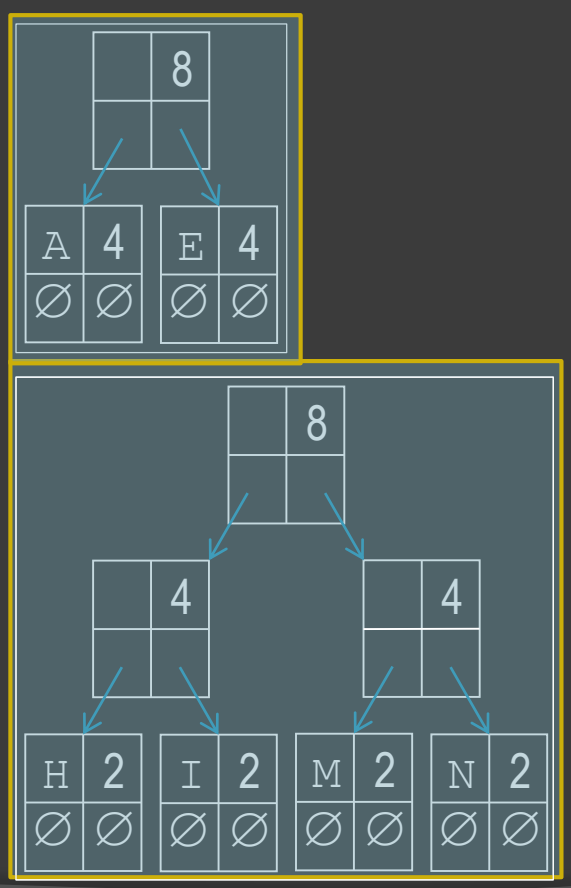


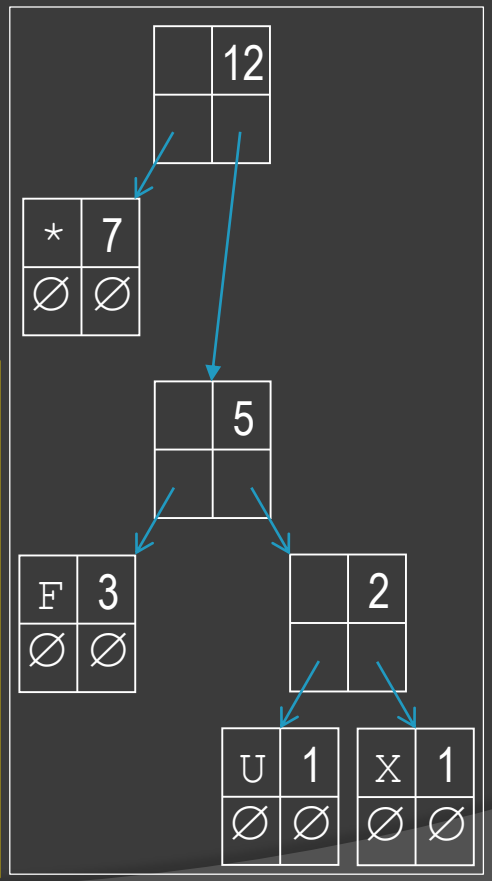
Continue Building The Tree

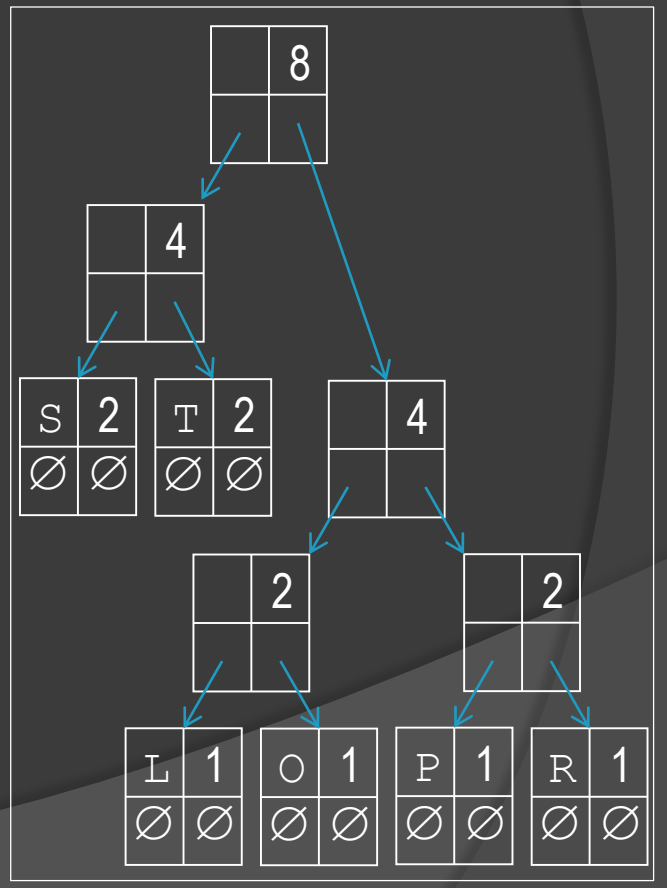
- Keep going: Group the two lowest-weight items into a tree whose parent node has their combined weight:



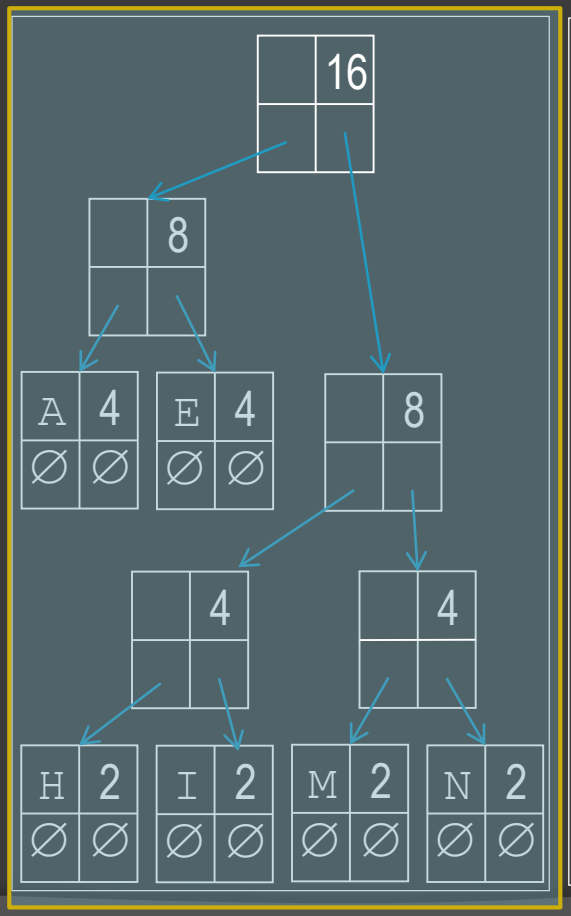
Continue Building The Tree

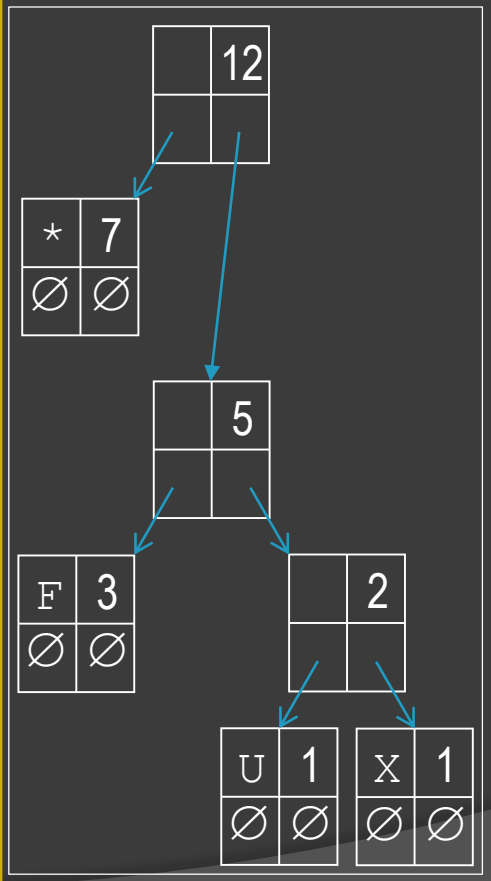


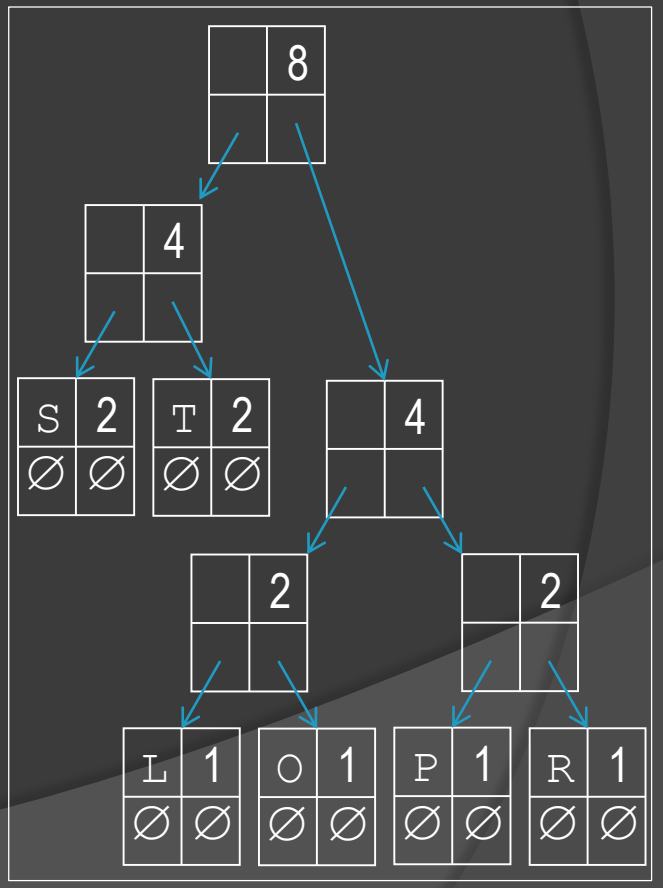




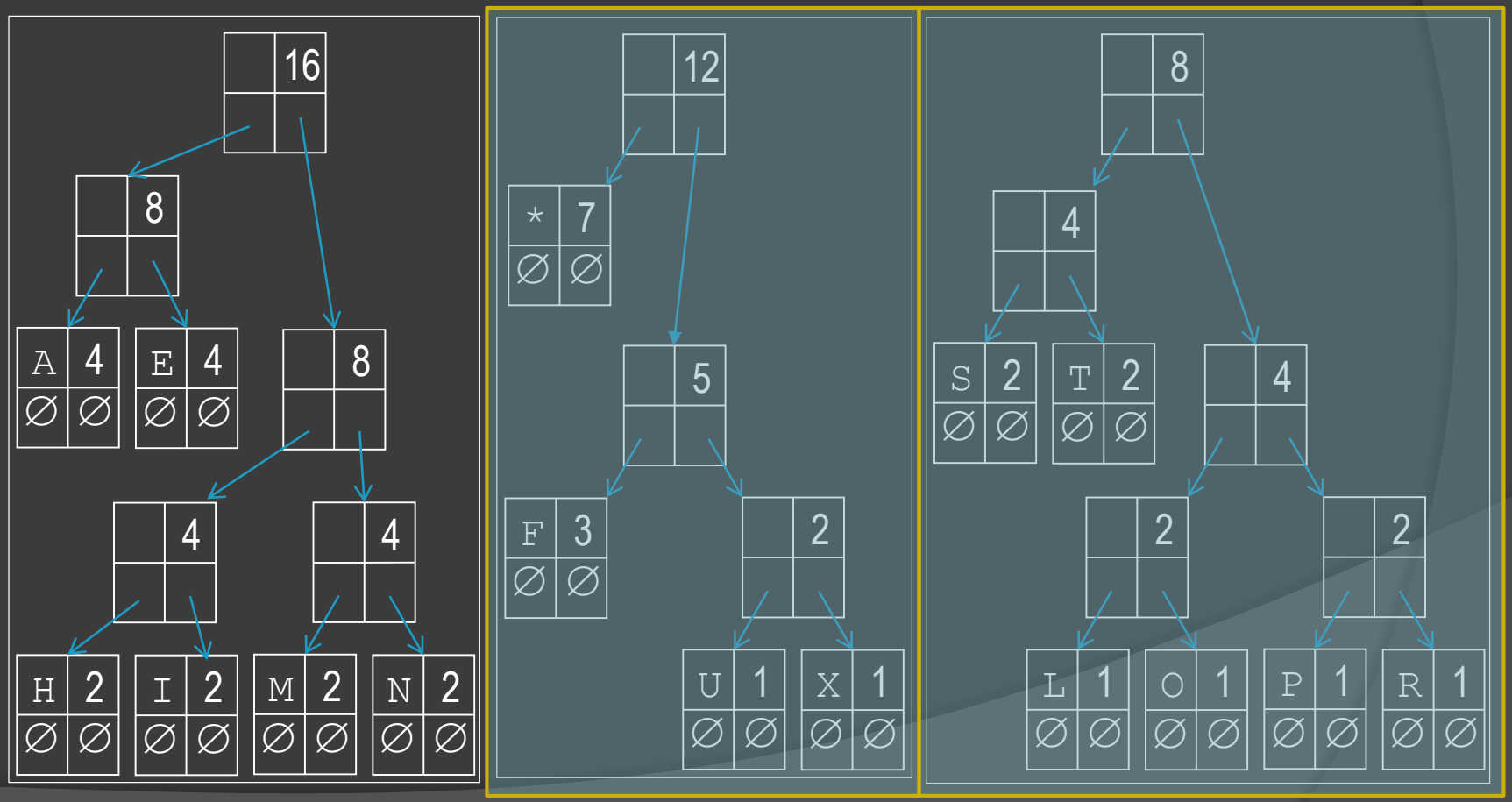
Continue Building The Tree



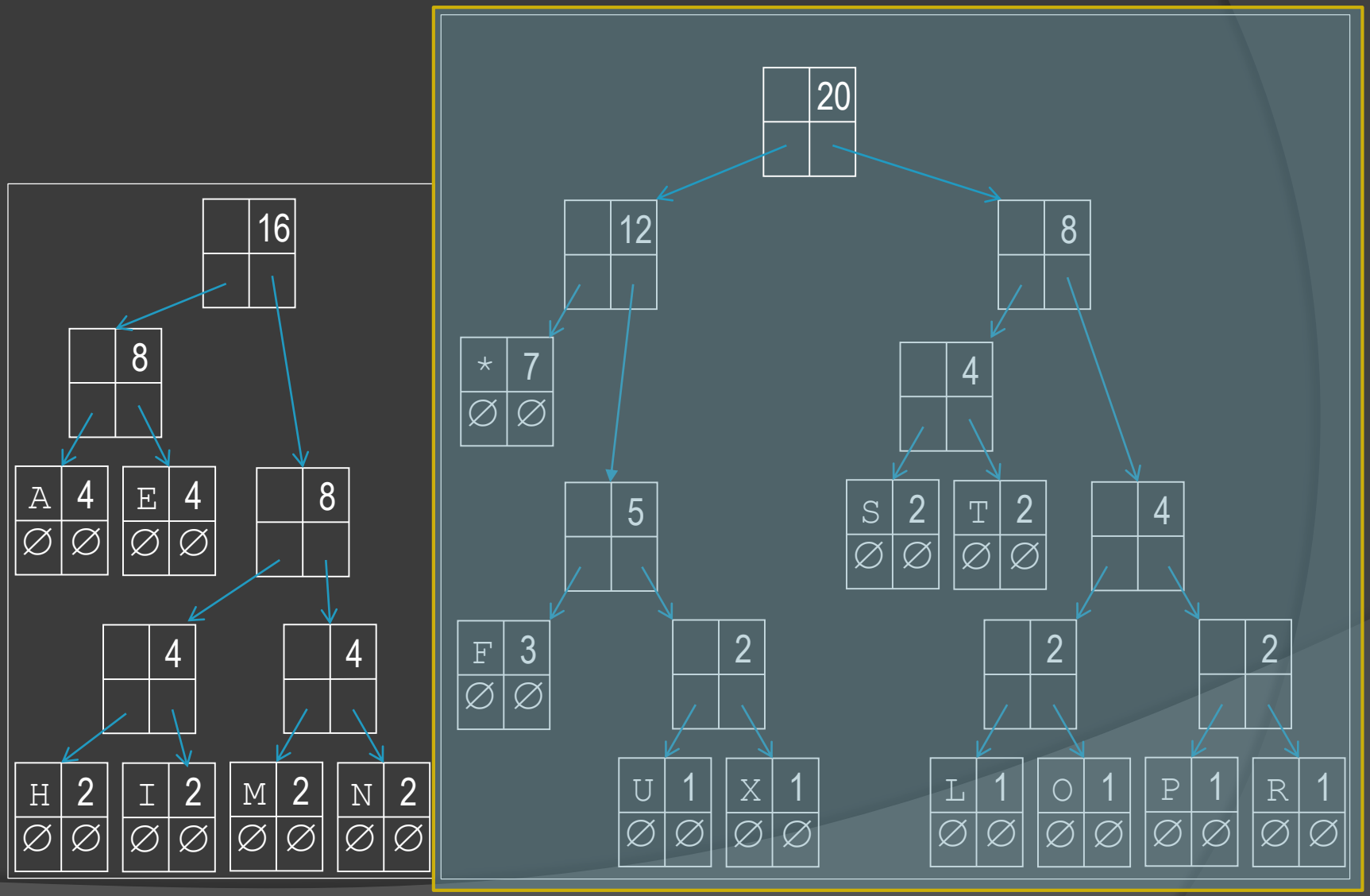




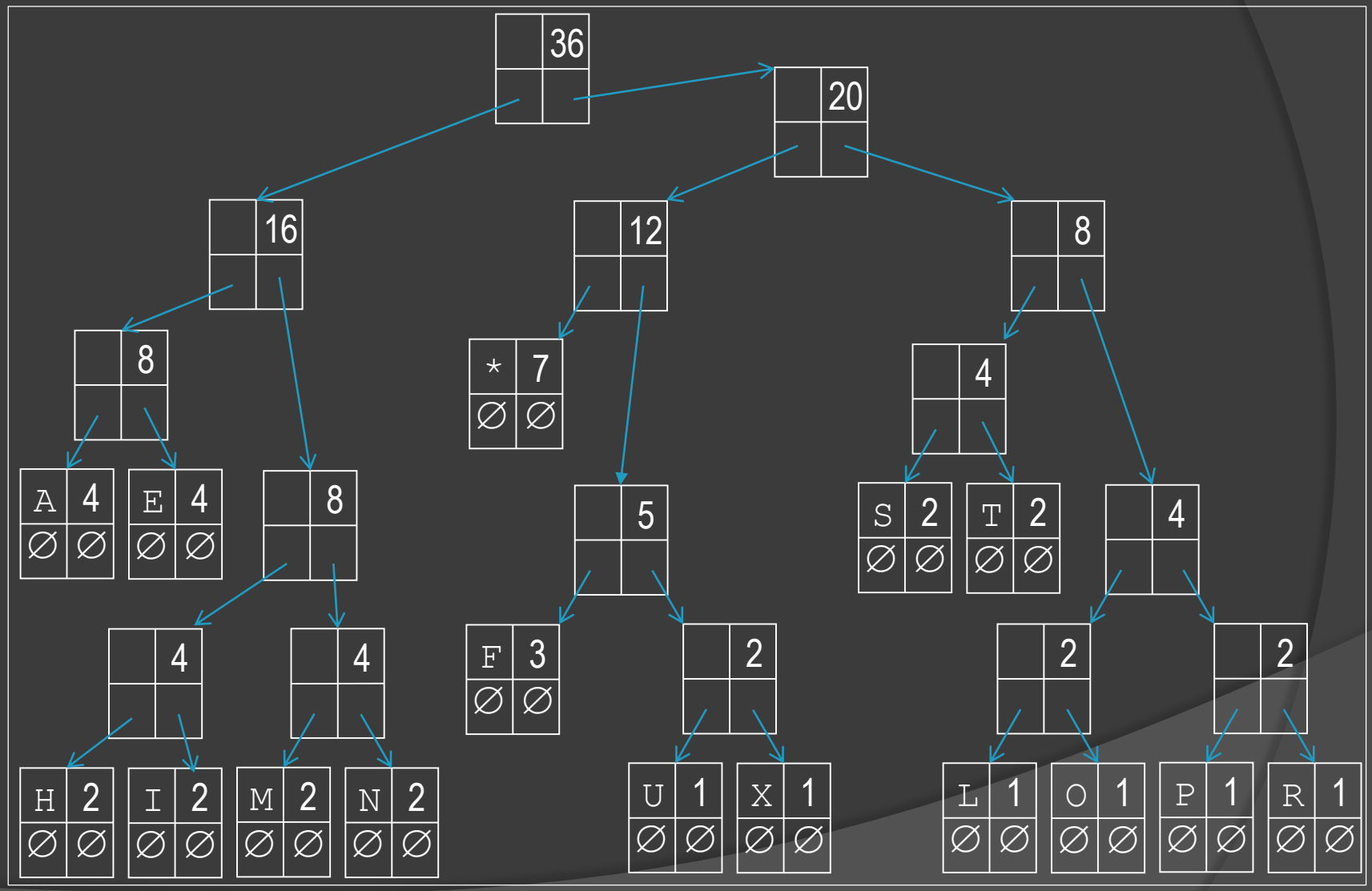
Continue Building The Tree



Continue Building The Tree



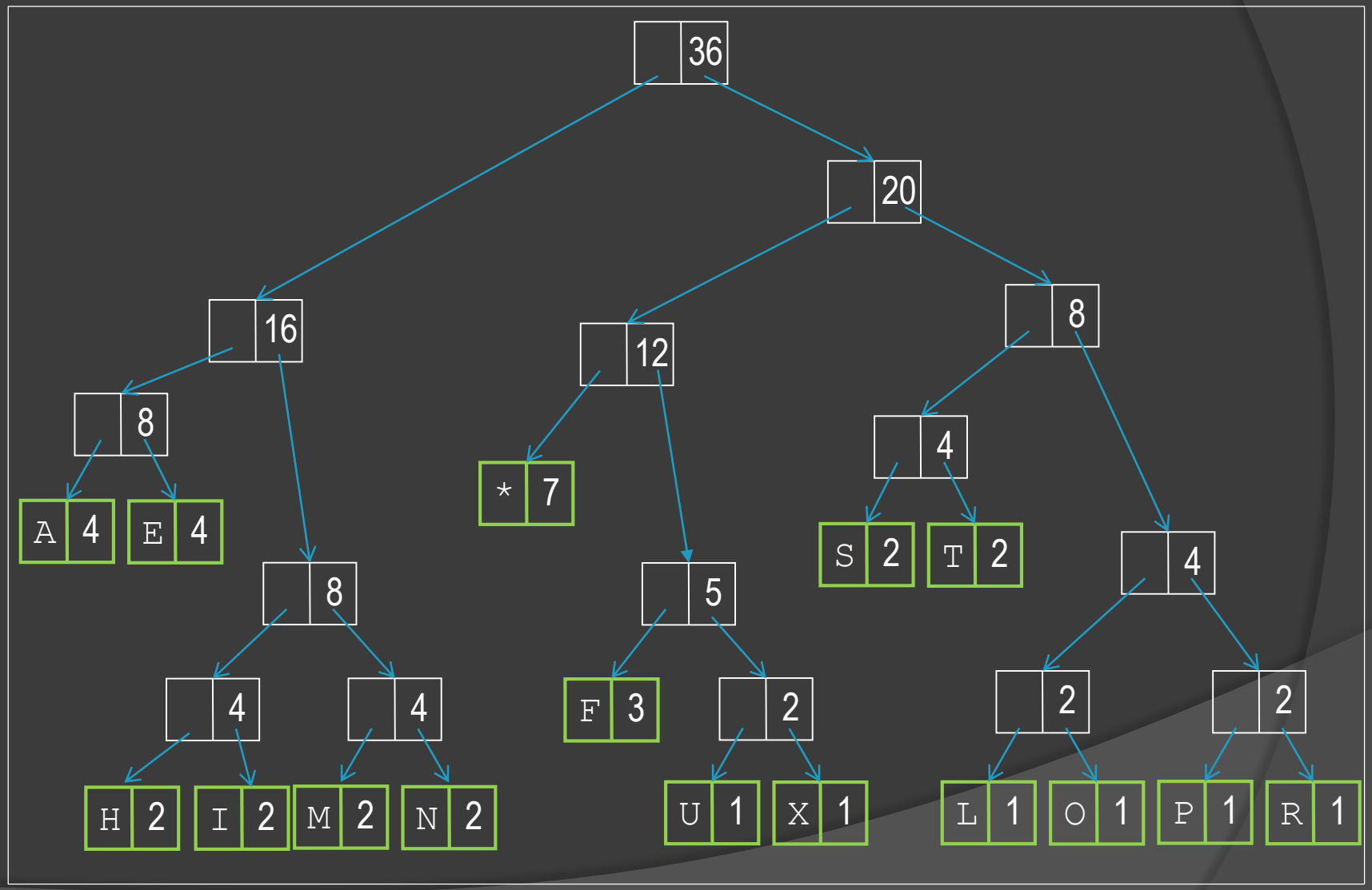
Continue Building The Tree



Let’s Refine The Drawing A Bit

- We know that only leaves have characters (letters), so we’ll drop the “∅” indicators
- Similarly, we can tell where the pointers go, so we’ll drop the LChild and RChild boxes
- Outline the leaves (nodes containing characters) in green
- We're not making any content changes to the tree; just in how we draw it

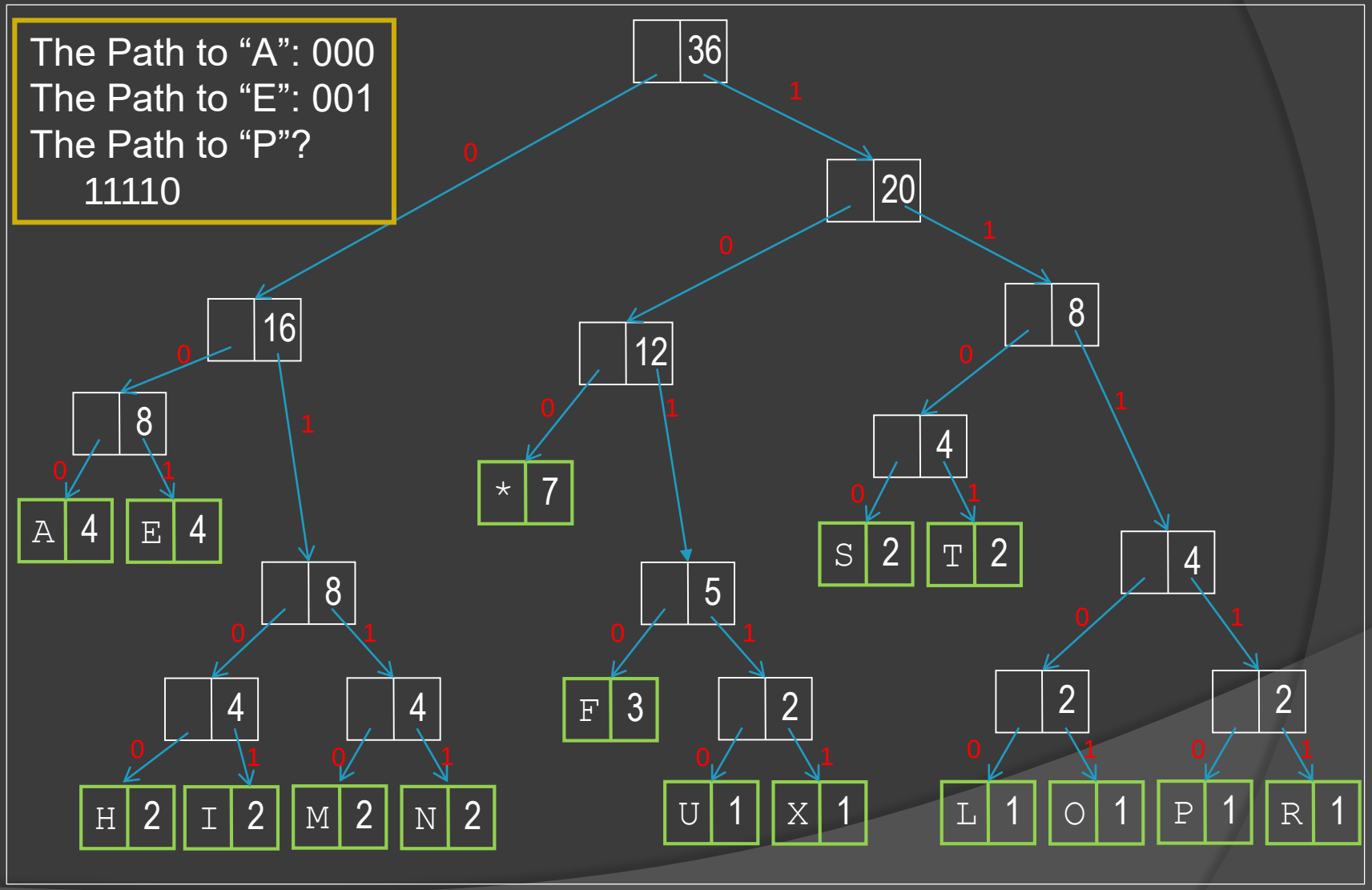
The Finished Tree



What Next?

- At this point, we don’t care about weights, but we will continue to show them
- The weights are only needed to build the tree; from this point on, they're useless, but since they're already in the nodes, we'll just let them sit there.
- Note that the weight at the root equals the number of characters we started with (it should – this is a convenient way to double-check that the tree was not built incorrectly)

Finished Tree – Numbered Branches



The Original Data, Revisited

Character	Occurs	Path		Character	Occurs	Path
space	7	100		S	2	1100
A	4	000		T	2	1101
E	4	001		L	1	11100
F	3	1010		O	1	11101
H	2	0100		P	1	11110
I	2	0101		R	1	11111
M	2	0110		U	1	10110
N	2	0111		X	1	10111

- What do we notice about the paths we wound up with?
- The characters that occur more often have shorter paths (and vice versa)

The Original Data, Revisited

Character	Occurs	Path		Character	Occurs	Path
space	7	100		S	2	1100
A	4	000		T	2	1101
E	4	001		L	1	11100
F	3	1010		O	1	11101
H	2	0100		P	1	11110
I	2	0101		R	1	11111
M	2	0110		U	1	10110
N	2	0111		X	1	10111

- In Huffman Encoding, the PATHs to the leaf nodes become the bit strings we use to represent the characters stored in the leaf nodes
  - Hold onto this idea for a minute

The Original Data, Revisited

Character	Occurs		Character	Occurs
space	7		S	2
A	4		T	2
E	4		L	1
F	3		O	1
H	2		P	1
I	2		R	1
M	2		U	1
N	2		X	1

- The original “alphabet” for this message consists of 16 characters

The Original Data, Revisited

Character	Encoded		Character	Encoded
space	0000		S	1000
A	0001		T	1001
E	0010		L	1010
F	0011		O	1011
H	0100		P	1110
I	0101		R	1101
M	0110		U	1110
N	0111		X	1111

- We *could* have used a four-bit scheme to encode these sixteen characters
- “THIS IS AN EXAMPLE OF A HUFFMAN TREE” (36 characters) takes 36*4 = 144 bits

The Original Data, Revisited

Character	Path		Character	Path
space	100		S	1100
A	000		T	1101
E	001		L	11100
F	1010		O	11101
H	0100		P	11110
I	0101		R	11111
M	0110		U	10110
N	0111		X	10111

- How do we encode "THIS IS AN EXAMPLE OF A HUFFMAN TREE" using our tree paths?
- Simple substitution:

1101	0100	0101	1100	100	0101	1100	100	000	0111	100	001	10111	000	1001
T	H	I	S	*	I	S	*	A	N	*	E	X	A	M…

Total space required using fixed 4-bit encoding: 36 * 4 = 144 bits
Total space required using variable-bit-length encoding: 135 bits
135/144 = 0.9375 → we got 6.25% compression

About Compression Levels

- We got only ~6% compression
- Hardly seems worth the effort
- However: We had a small alphabet, and the character frequency counts did not have widely different values
- Our Huffman tree was "somewhat" imbalanced
- The more imbalanced a Huffman tree is, the higher the level of compression
  - If "space" and "A" had occurred 100 times each, they would have had two-bit patterns → more compression

The Original Data, Revisited

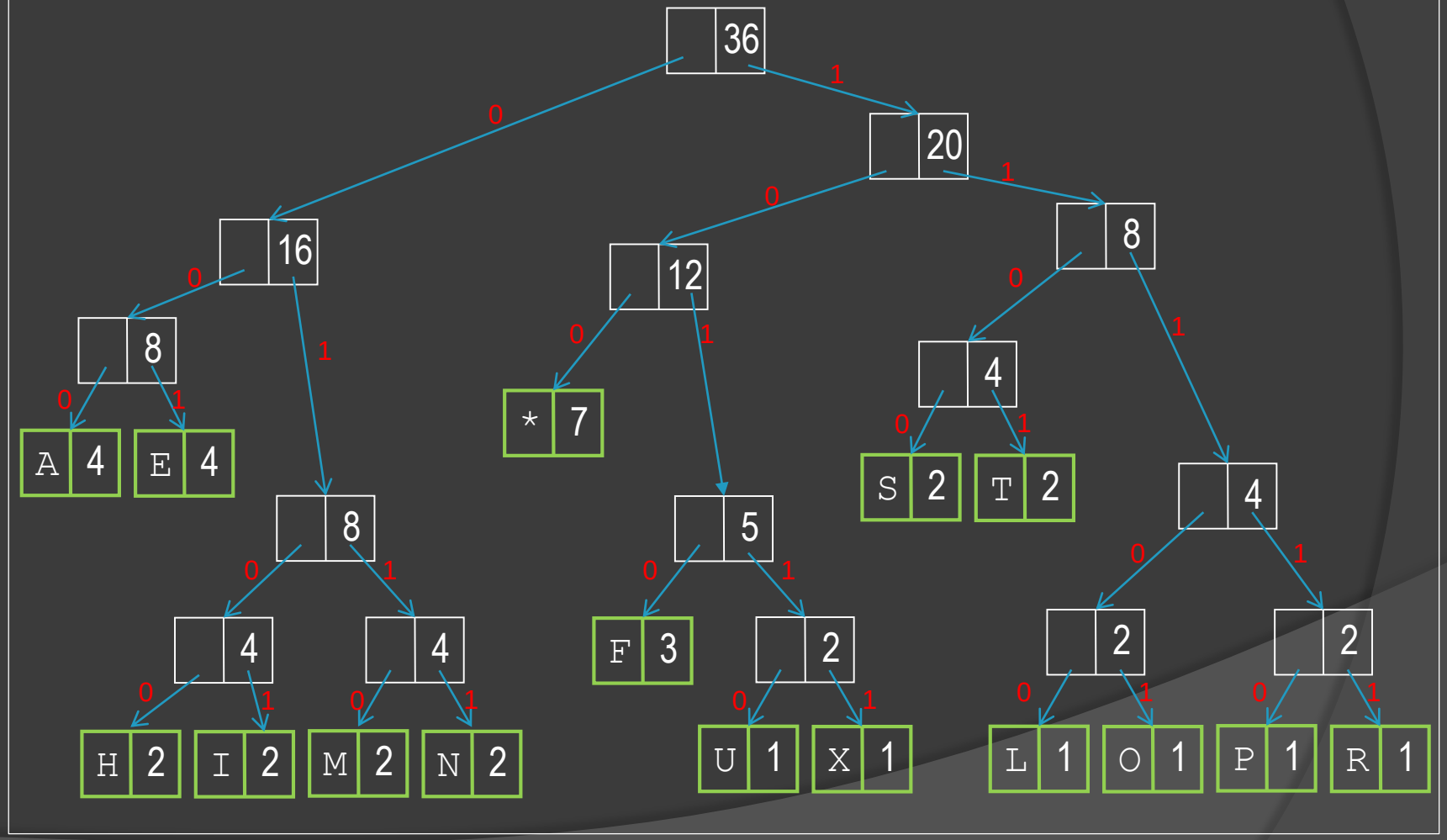
Character	Path		Character	Path
space	100		S	1100
A	000		T	1101
E	001		L	11100
F	1010		O	11101
H	0100		P	11110
I	0101		R	11111
M	0110		U	10110
N	0111		X	10111

- Encoding is only half of the process – we have to be able to *DECODE* as well

- What does this say (i.e., how do we decode)?
- 1111011111010101111101001111111000101110010011101011 1
- We use the tree we built!

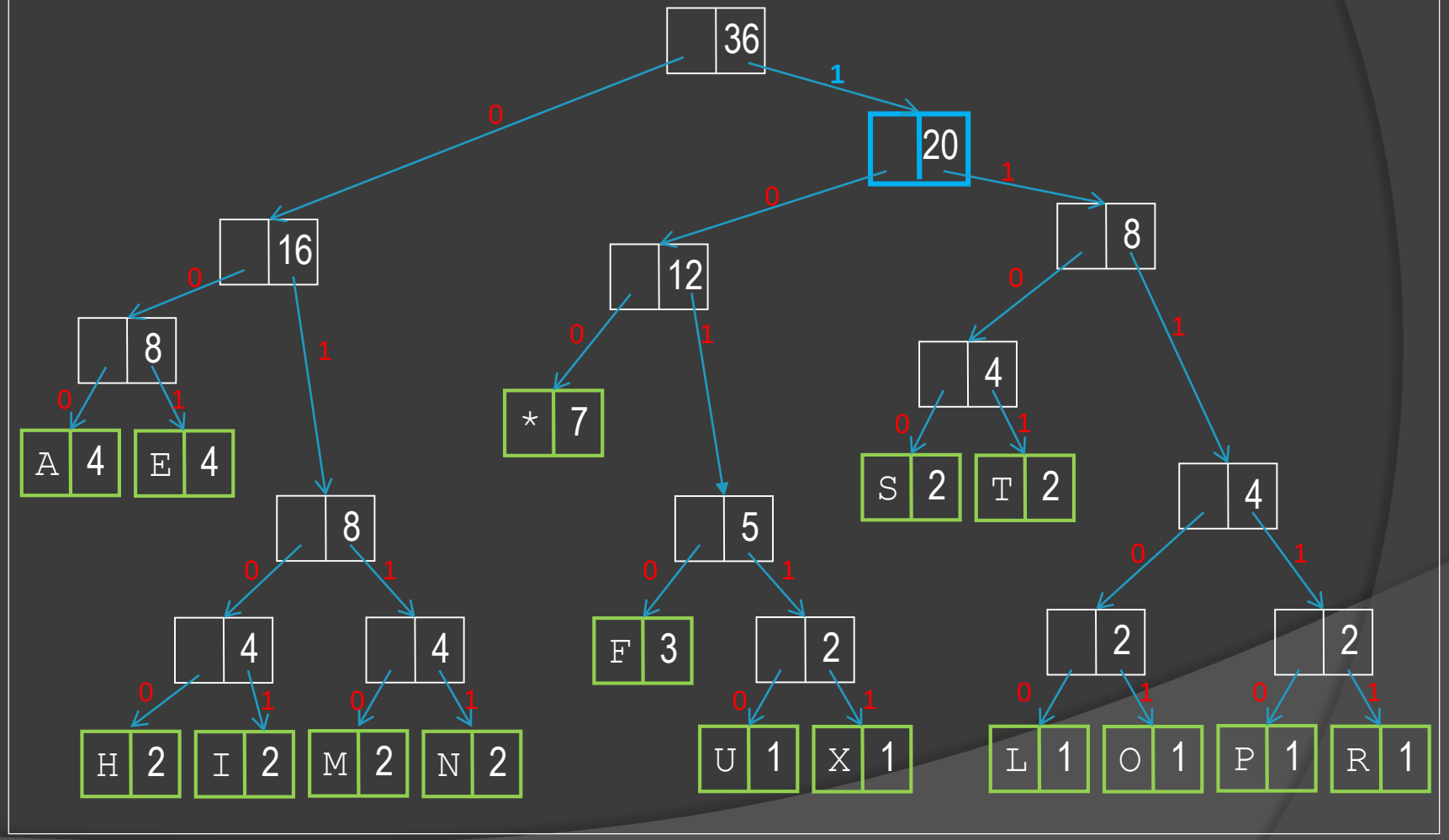
Using The Tree To Decode

Decode: 111101111101010111110100111111100010111001001110101111



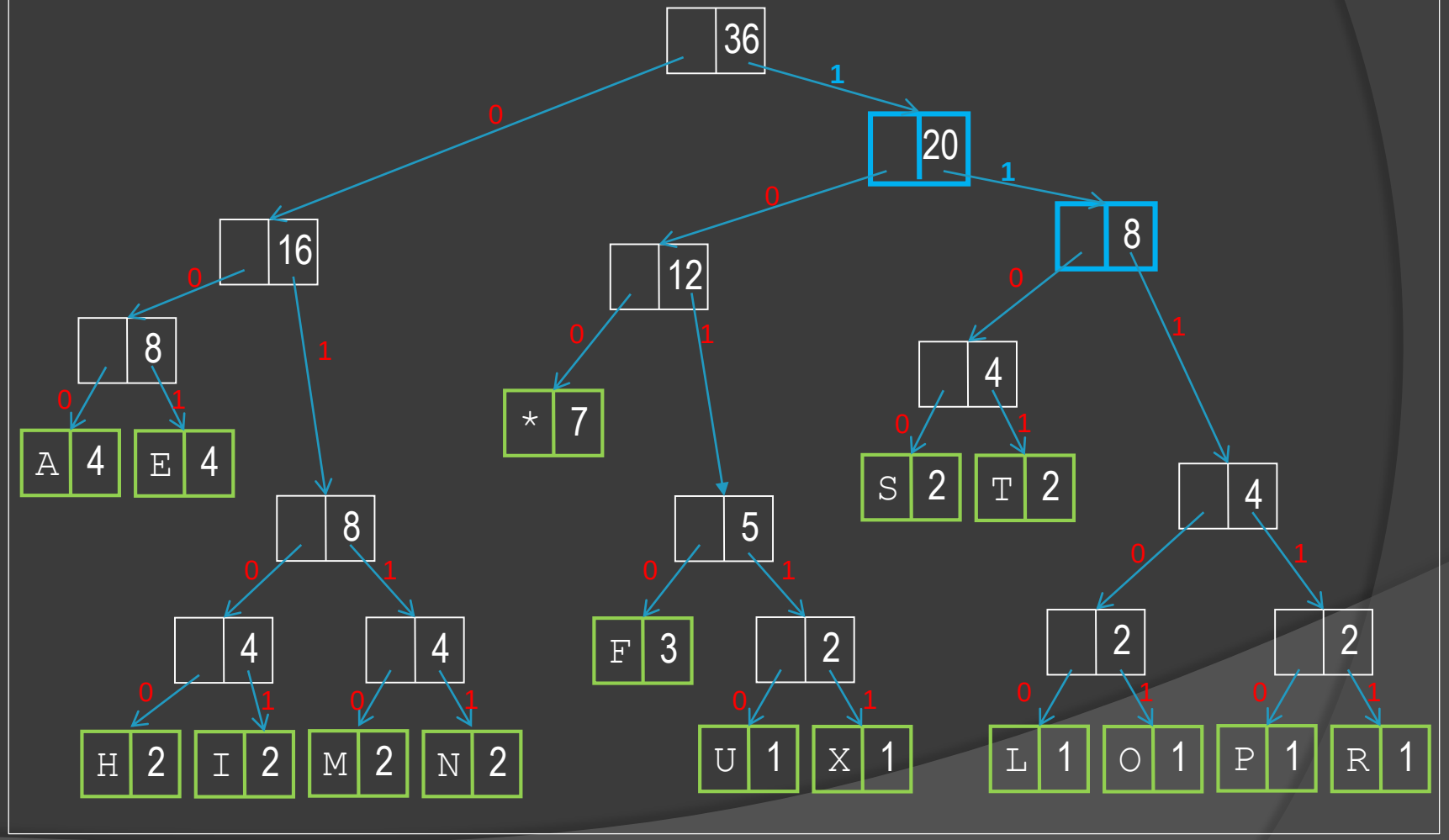
Using The Tree To Decode

Decode: 1111011111010101111101001111111000101110010011101011 1



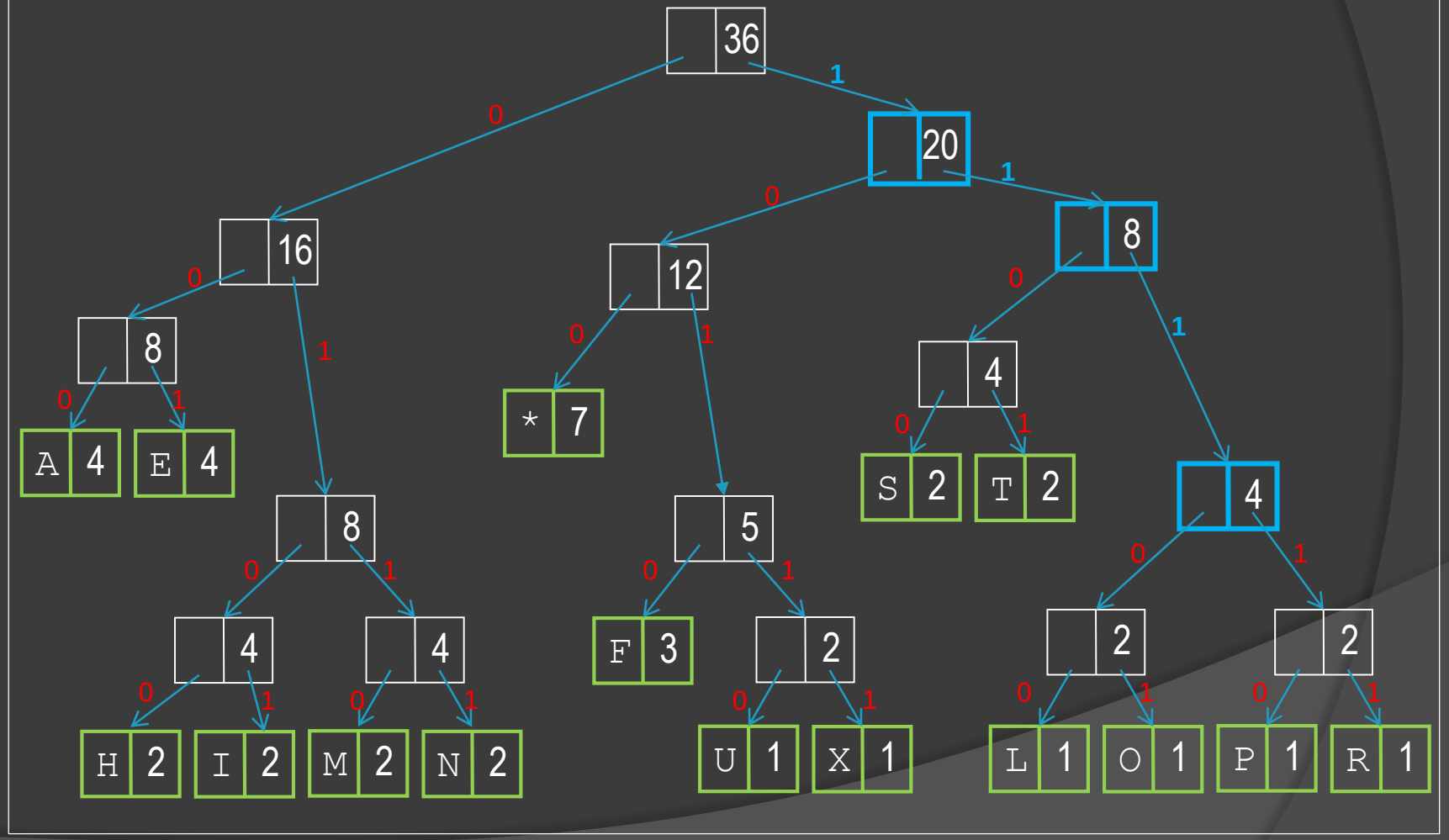
Using The Tree To Decode

Decode: 11110111110101011111010011111110001011100100111010111



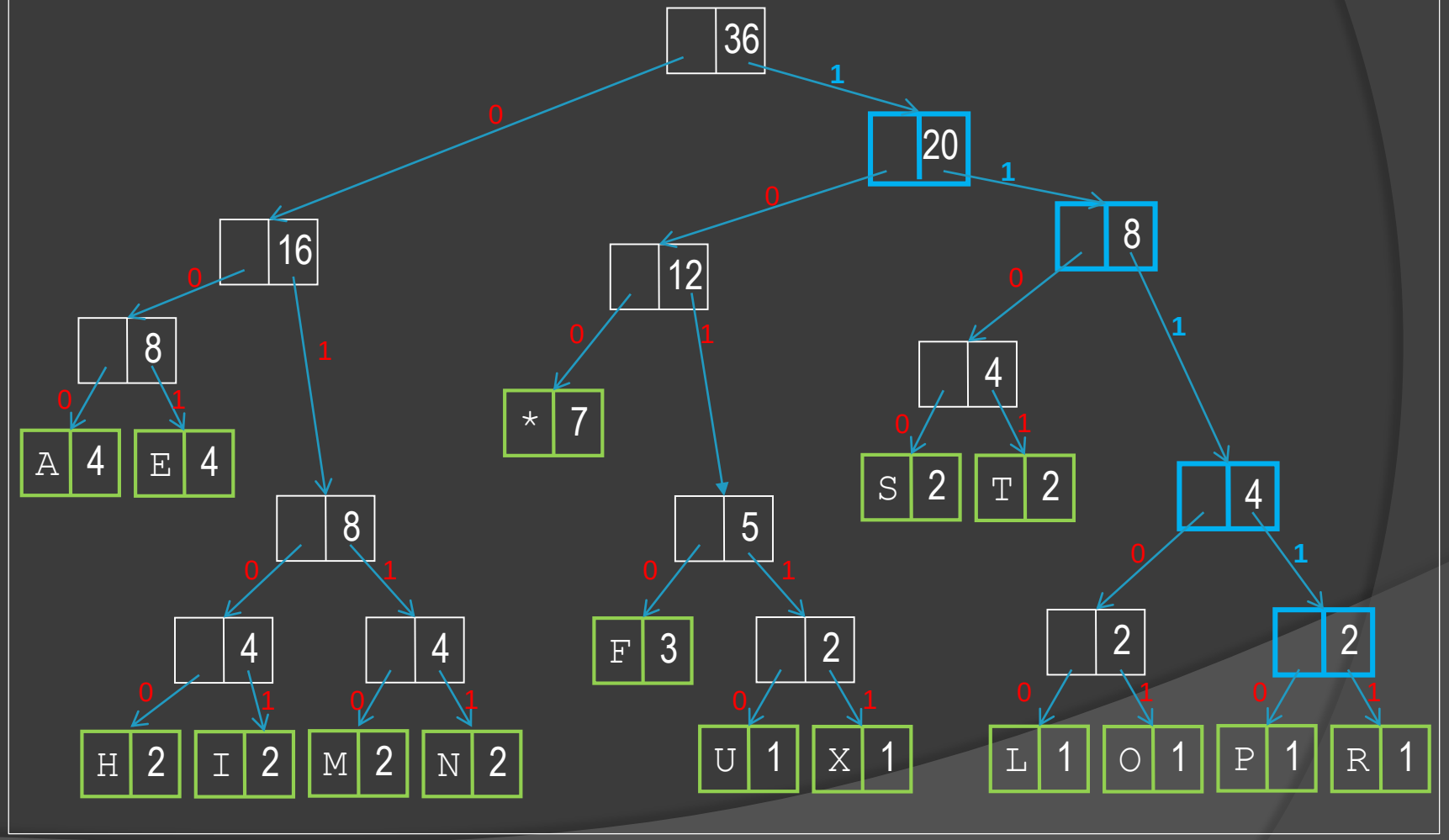
Using The Tree To Decode

Decode: 111101111101010111110100111111100010111001001110101111



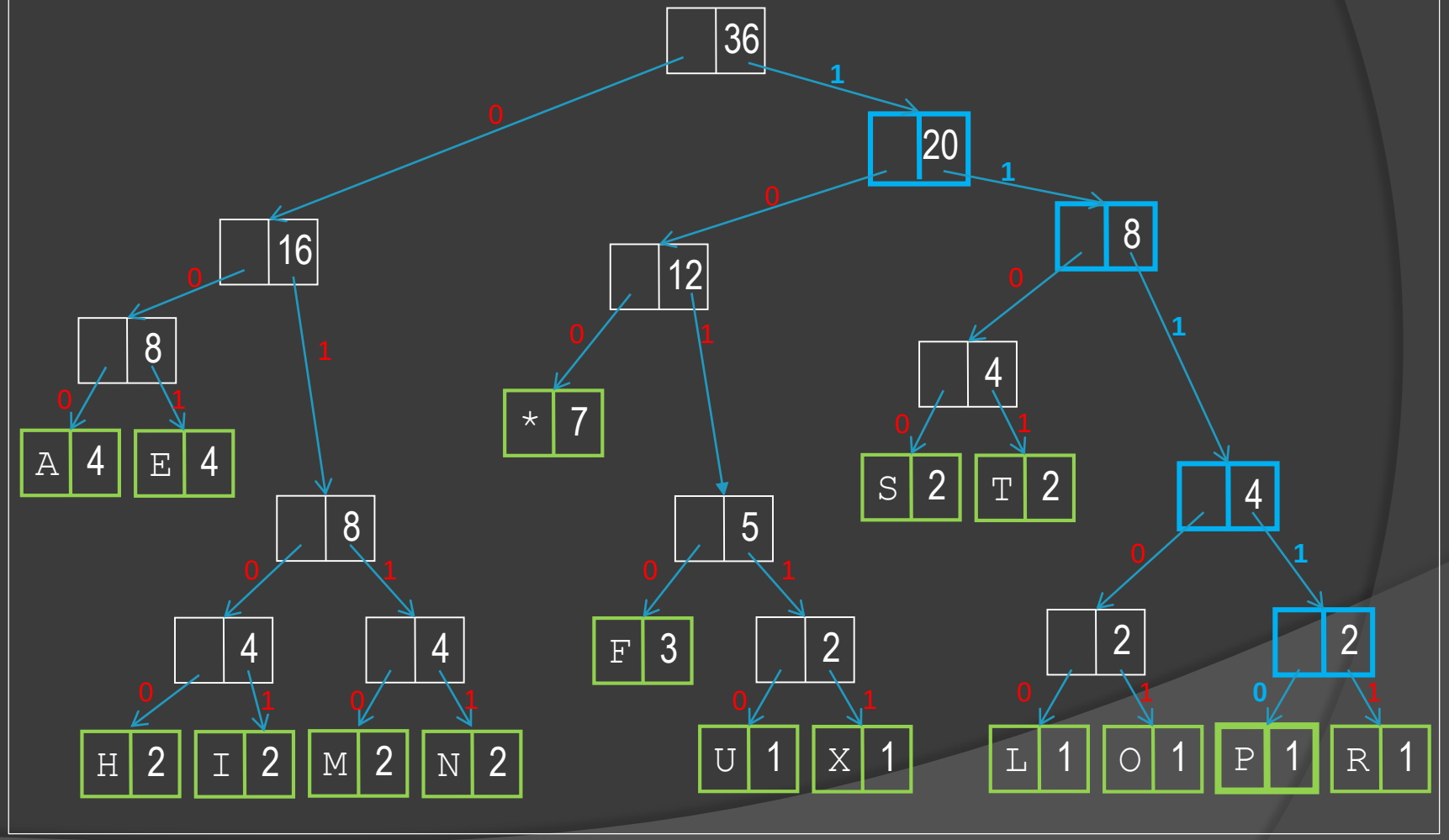
Using The Tree To Decode

Decode: 1111011111101010111110100111111100010111001001110101111

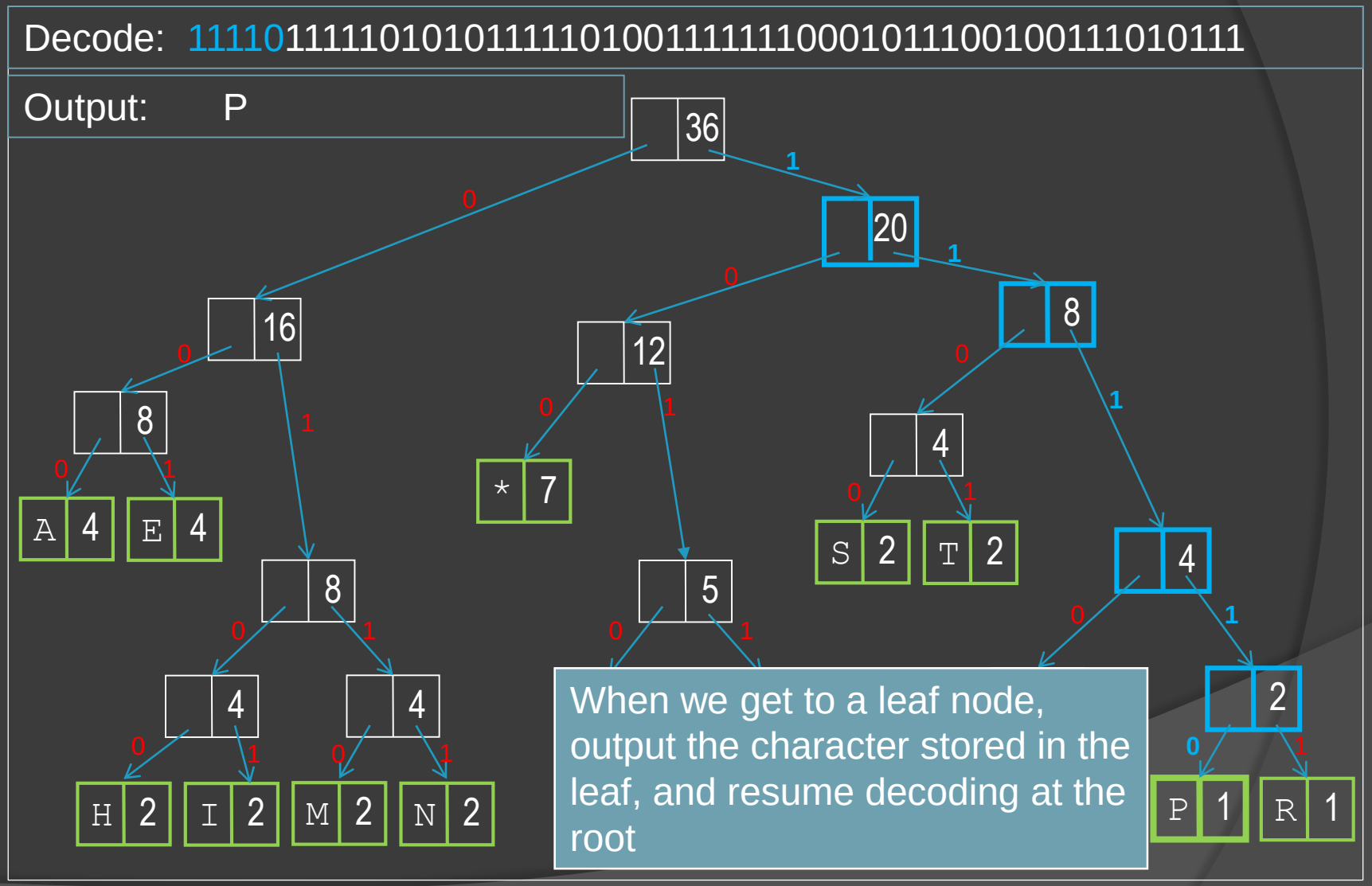


Using The Tree To Decode

Decode: 11110111111010101111101001111111000101110010011101011 1



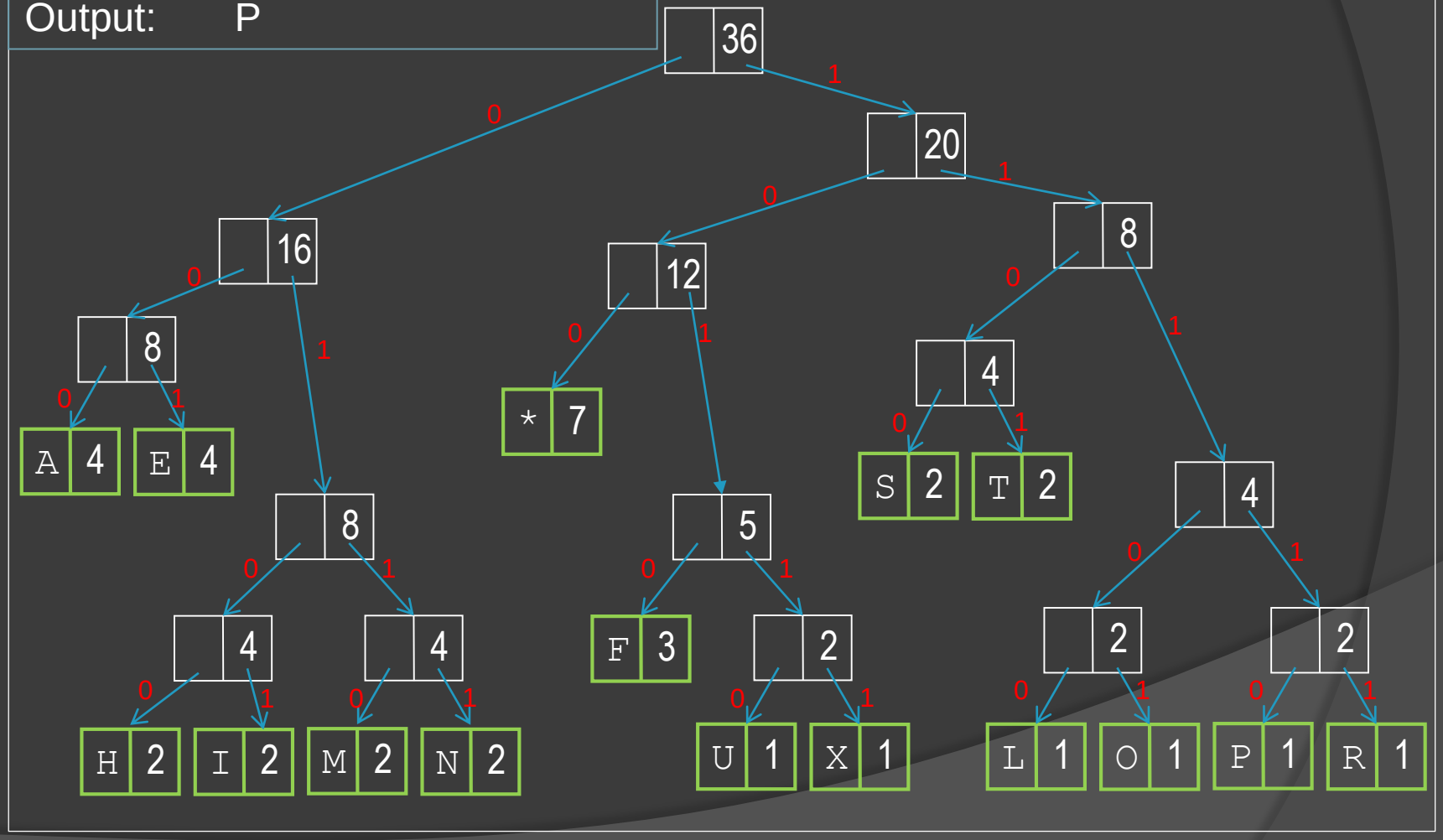
Using The Tree To Decode



Using The Tree To Decode

Decode: 11110111111010101111101001111111000101110010011101011 1

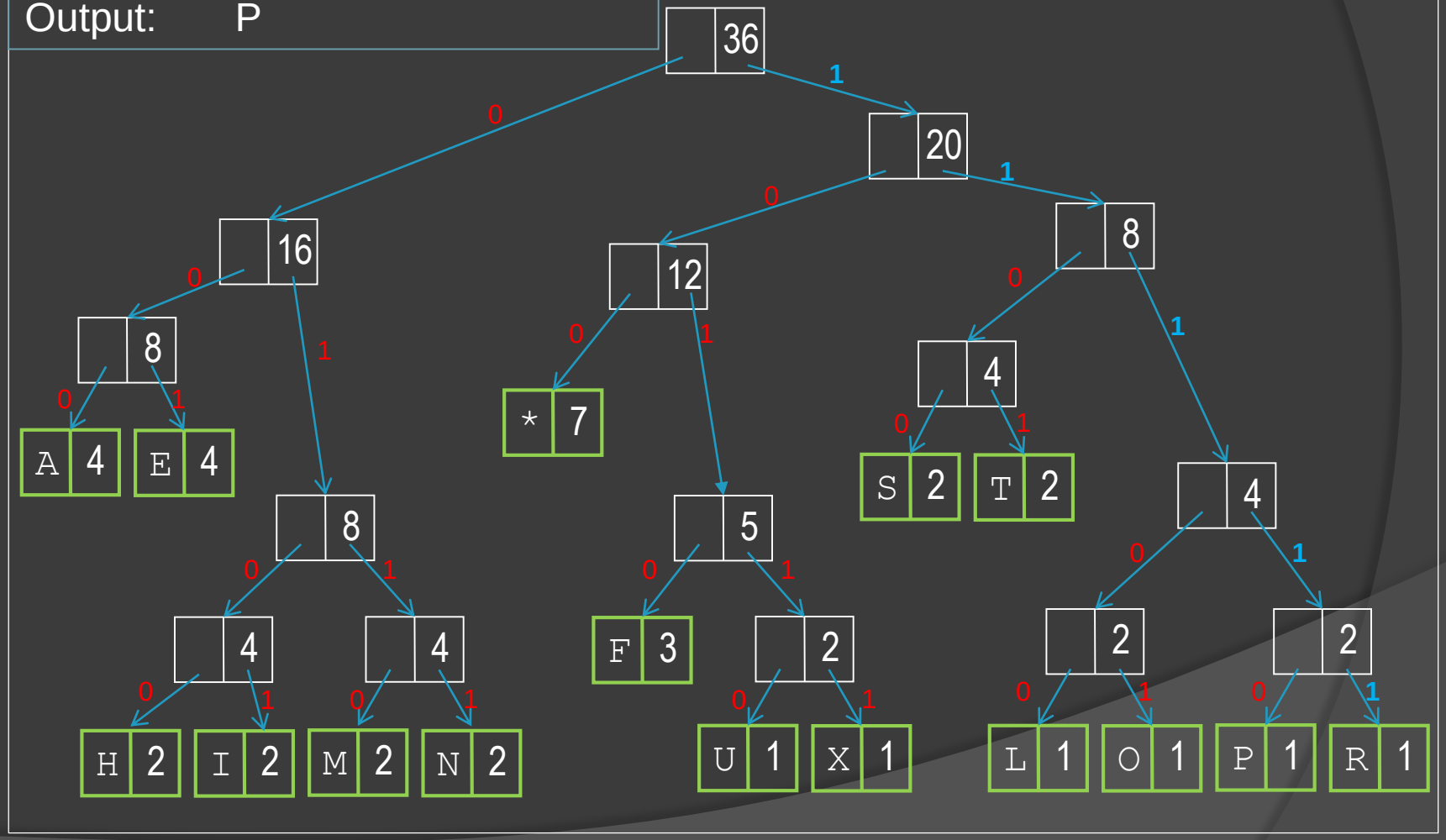
Output: P



Using The Tree To Decode

Decode: 1111011111010101111101001111111000101110010011101011 1

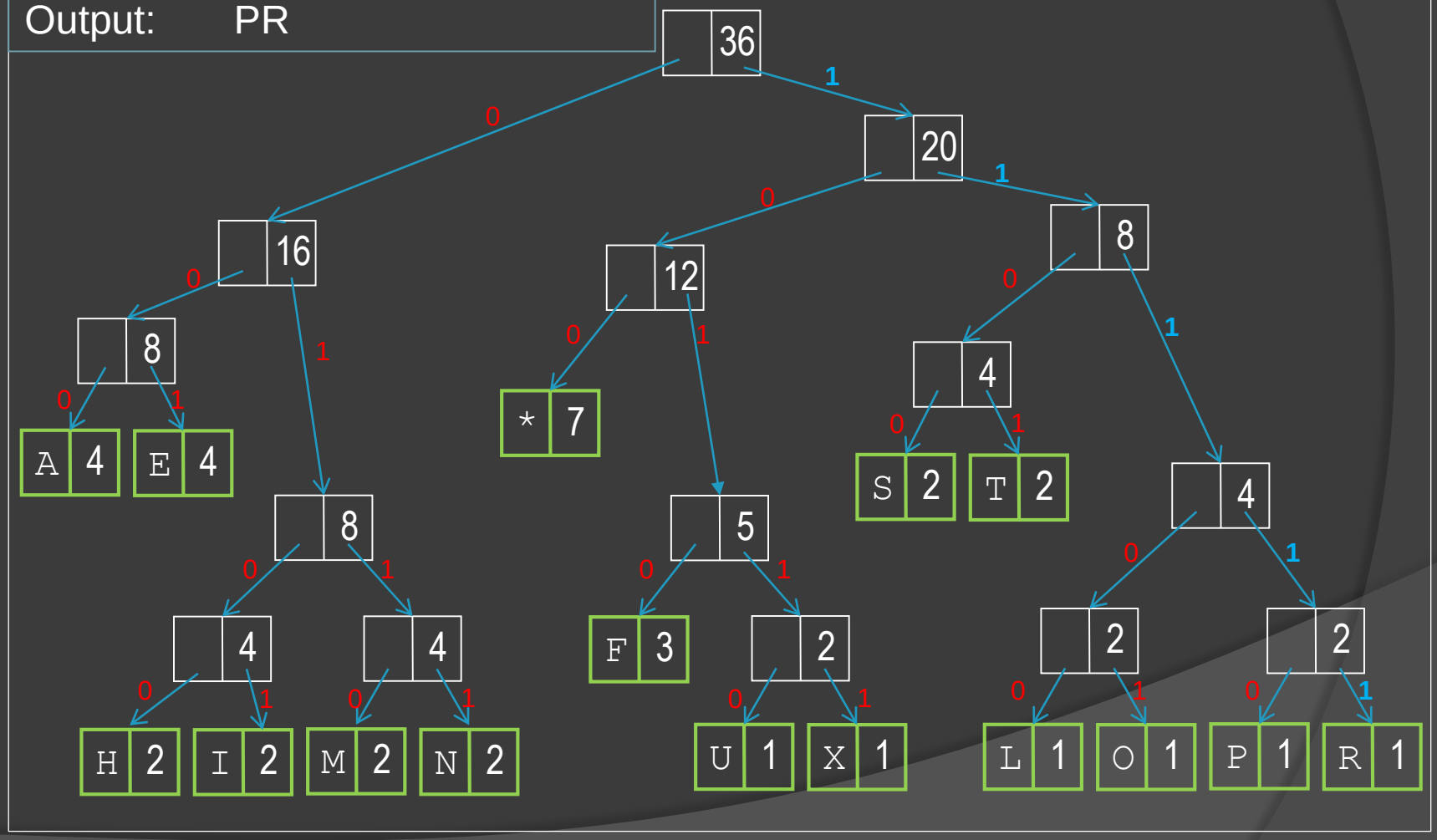
Output: P



Using The Tree To Decode

Decode: 1111011111010101111101001111111000101110010011101011 1

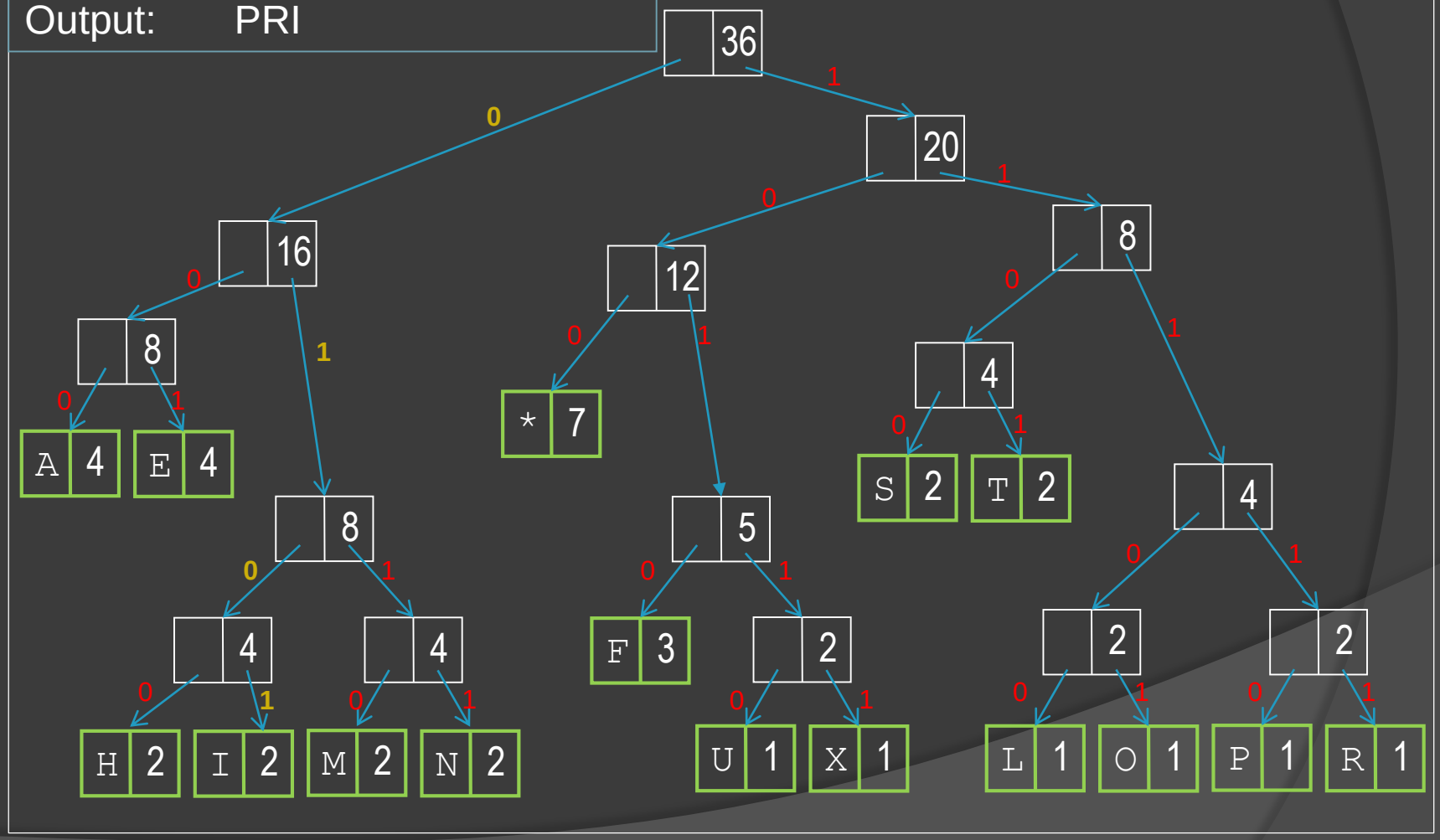
Output: PR



Using The Tree To Decode

Decode: 1111011111010101111110100111111100010111001001110101111

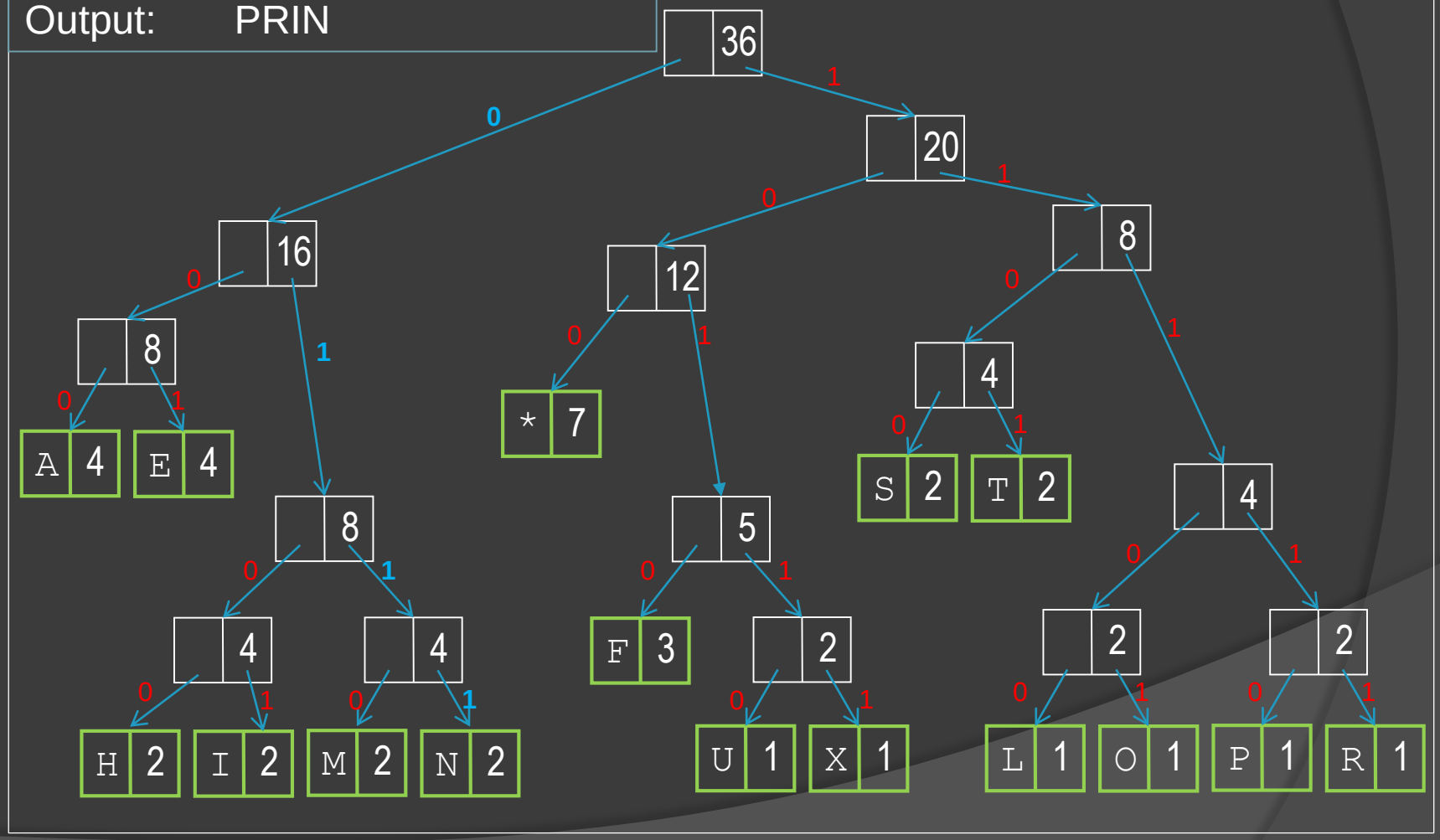
Output: PRI



Using The Tree To Decode

Decode: 11110111110101011110100111111100010111001001110101111

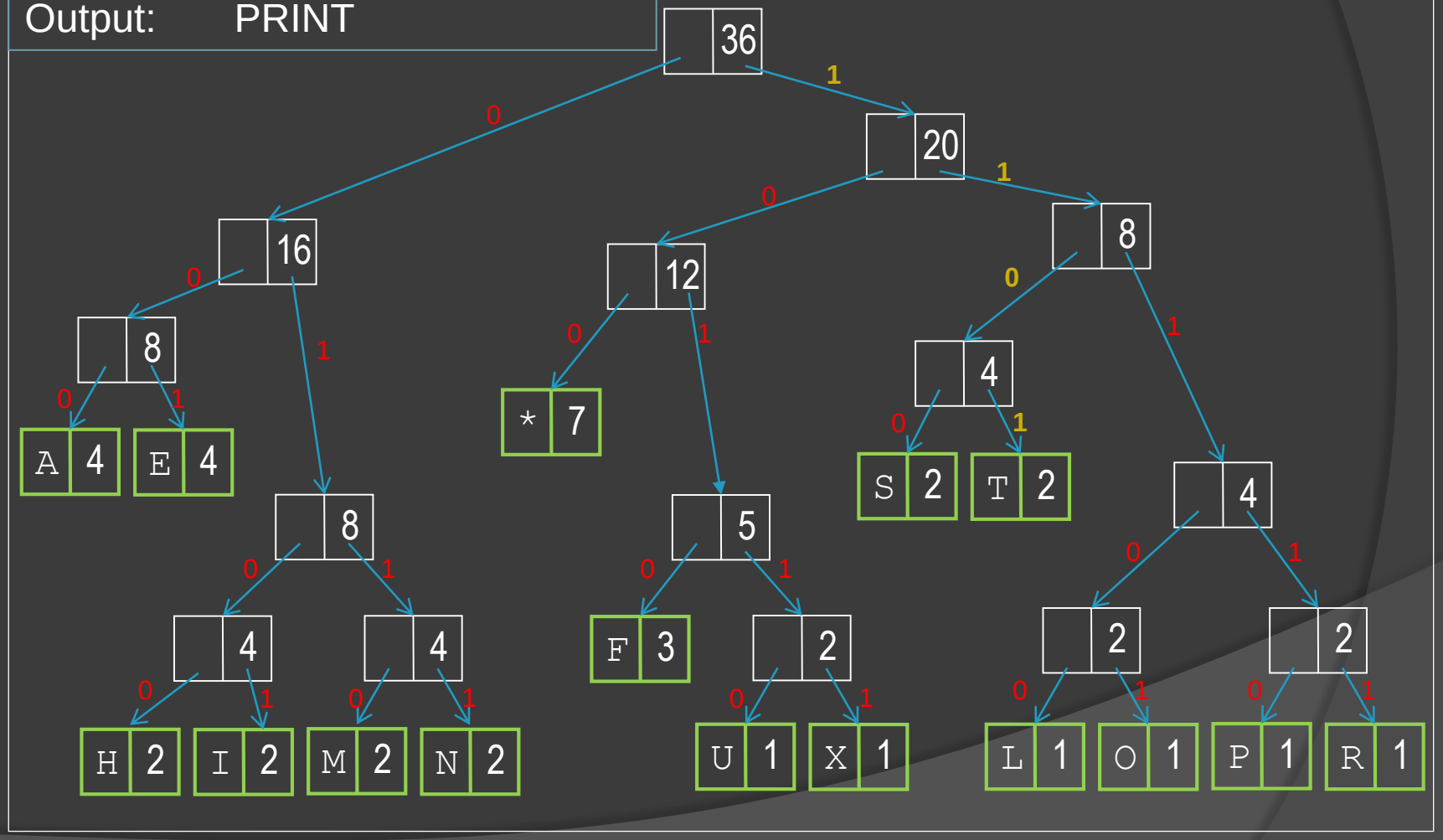
Output: PRIN



Using The Tree To Decode

Decode: 11110111110101011111010011111111000101110010011101011 1

Output: PRINT

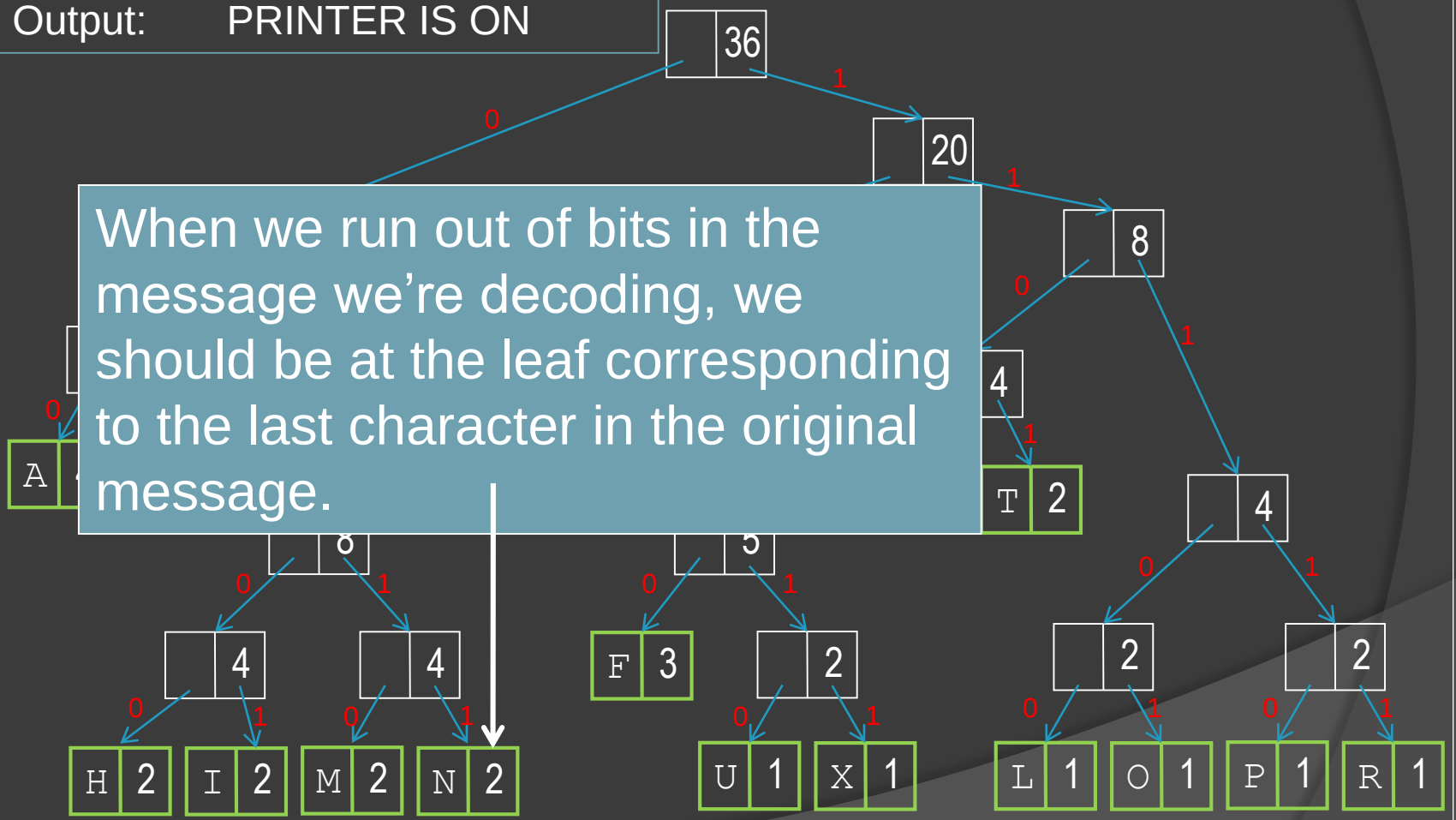


Decode The Rest Of This String

Decode: 1111011111010101111101001111111000101110010011101011 1

Output: PRINTER IS ON

When we run out of bits in the message we're decoding, we should be at the leaf corresponding to the last character in the original message.



Storing Variable-Length Bit Strings

- (Most) computers work in 8-bit bytes
- How can we store these variable-length strings in fixed-length bytes?

```
11110111  11010101  11110100  11111110  00101110  01001110  10111XXX
```

- We add *padding bits* to fill the void at the end
- We need three bits to fill the last byte
- What bits do we choose?

Storing Variable-Length Bit Strings

- If we're storing this variable-length bit string in 8-bit bytes:

```
11110111 11010101 11110100 11111110 00101110 01001110 10111XXX
```

- What 3 padding bits to use?
  - 000?
  - 001?
  - 010?
  - 011?
  - 100?
  - 101?
  - 110?
  - 111?

Character	Path		Character	Path
space	100		S	1100
A	000		T	1101
E	001		L	11100
F	1010		O	11101
H	0100		P	11110
I	0101		R	11111
M	0110		U	10110
N	0111		X	10111

Storing Variable-Length Bit Strings

- A couple of observations:
  - Whatever we choose *must not* change the meaning of the message as-transmitted
  - If we need padding bits at all, then we need somewhere between 1 and 7 of them, if we're using 8-bit bytes
  - We may not need *any* padding bits at all

Character	Path		Character	Path
space	100		S	1100
A	000		T	1101
E	001		L	11100
F	1010		O	11101
H	0100		P	11110
I	0101		R	11111
M	0110		U	10110
N	0111		X	10111

Choosing Padding Bits

- We have a few options:
  1. Use (up to) 3 spaces (and trim when received)
  2. Add some special end-of-transmission character to our alphabet (may not be possible/practical)
  3. If the longest bit string is longer than the maximum number of padding bits we might need, then we could use that bit string, knowing it would never generate a complete trip to a leaf:
     - No unwanted character!

Character	Path		Character	Path
space	100		S	1100
A	000		T	1101
E	001		L	11100
F	1010		O	11101
H	0100		P	11110
I	0101		R	11111
M	0110		U	10110
N	0111		X	10111

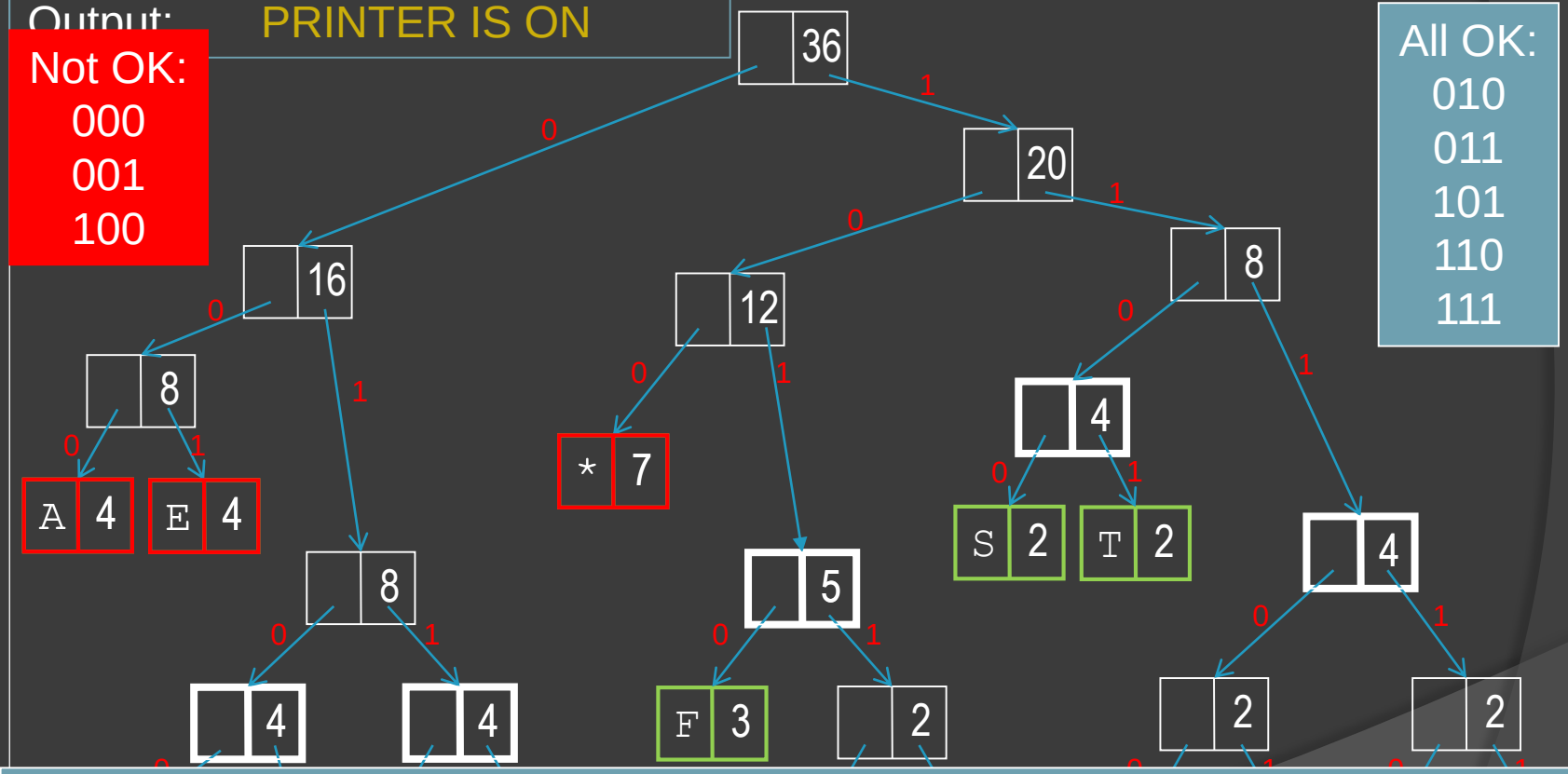
Choosing Padding Bits

Decode: 1111011111010101111101001111111000101110010011101011 1xxx

Output: PRINTER IS ON

Not OK:
000
001
100

All OK:
010
011
101
110
111



If we've chosen our padding bits wisely, we will be somewhere in the middle of the tree (at some internal node) when we run out of bits, so we can just stop without generating some unwanted character

Huffman Coding - Summary

1. If not already known, establish frequency of occurrence counts / ratios
2. Build the tree iteratively with the “aggregate the two smallest-weight items” algorithm
3. Traverse the tree to identify the encoding strings, and store them in a table
4. Encode via table look-up, padding if needed
5. Decode via tree navigation, outputting characters when leaves reached, continuing at root. Stop when we run out of bits (even if not at a leaf node)

Readings

Read chapter 16.3