Algorithms and Data Structures for Data Science
lab_huffman

CS 277
Brad Solomon

March 31, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

Review fundamentals of binary trees

Experience using data structures for data compression

Practice more open-ended coding problems

Optimal Storage Costs

Achieving an optimal storage cost for a dataset is often important

Let's use strings as an accessible example!

What is the minimum bits needed to encode the message:

Char	Binary
f	000
e	001
d	010
m	100
r	011
o	101
‘ ‘	110

`‘feed me more food’`

Optimal Storage Costs

Using three bits per character, we have 51 bits total. But can we do better?

`‘feed me more food’`

If we think about our input as a sorted list of frequencies, yes!

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Using binary trees for string encoding

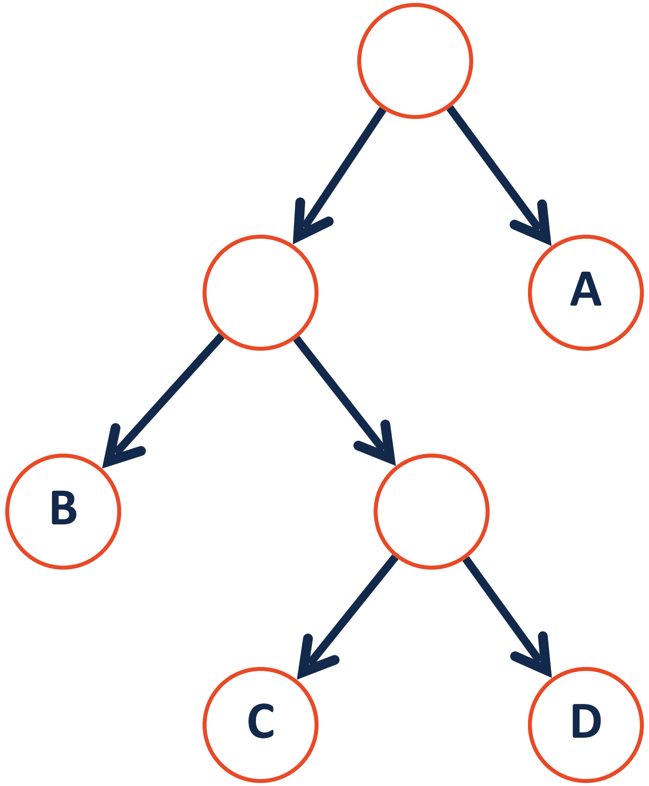
Lets define a tree with the following:

The keys are individual characters

The values are the frequencies of those characters

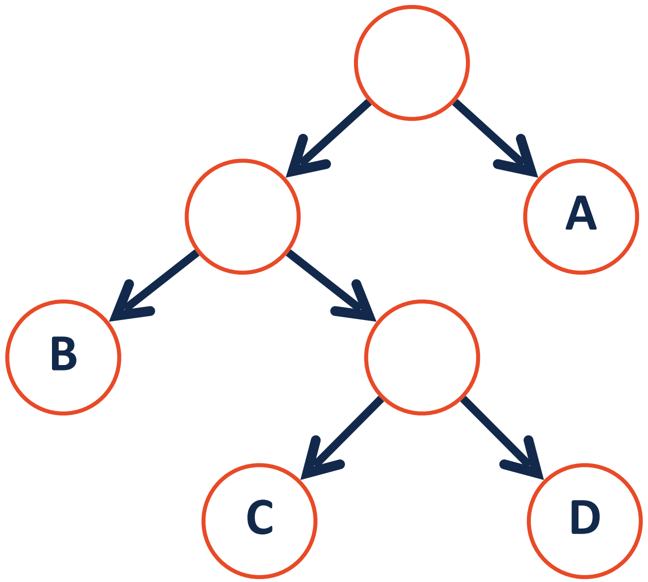
```
class bstNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```

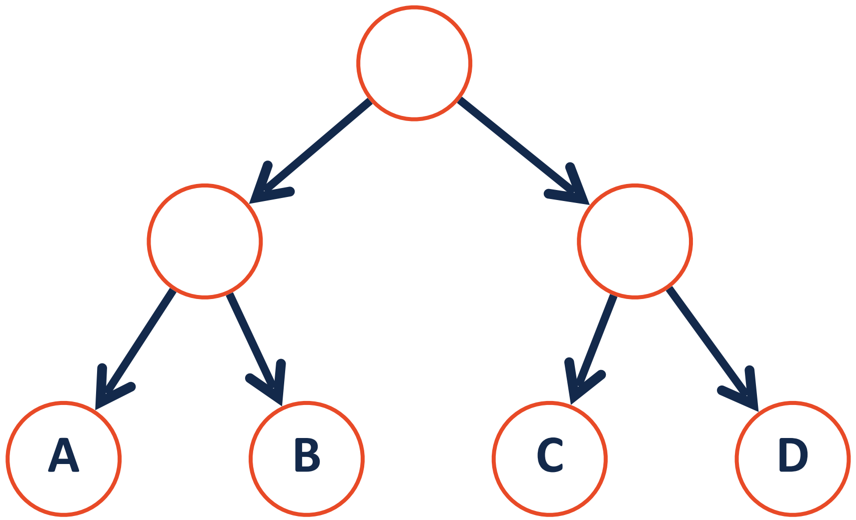
Key	A	B	C	D
Value	7	5	2	4



Binary Tree encoding

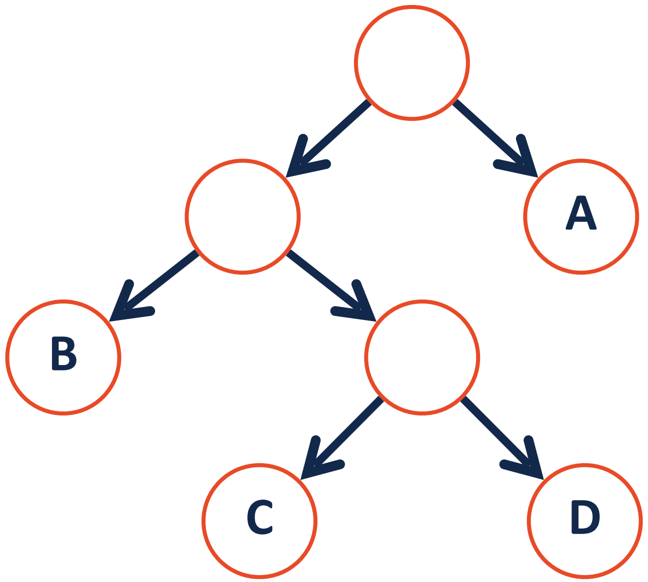
Given the following two trees, how might we define an encoding?



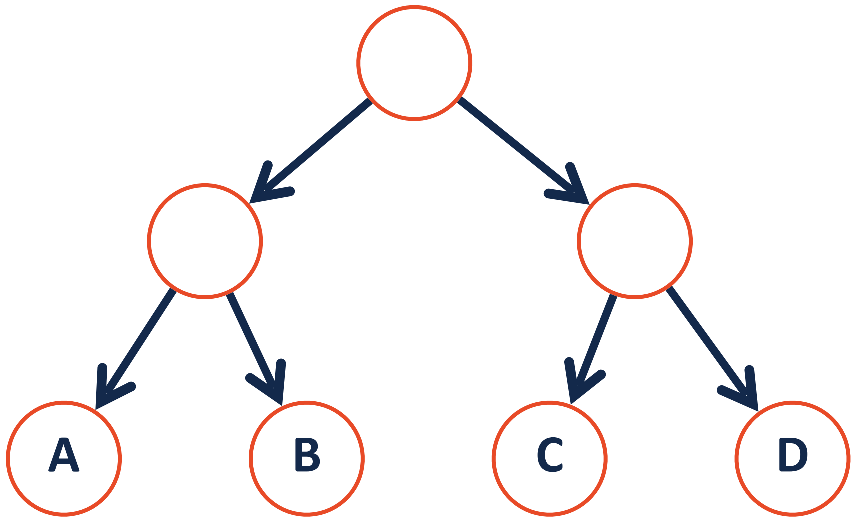


Binary Tree encoding

How did we produce this encoding?



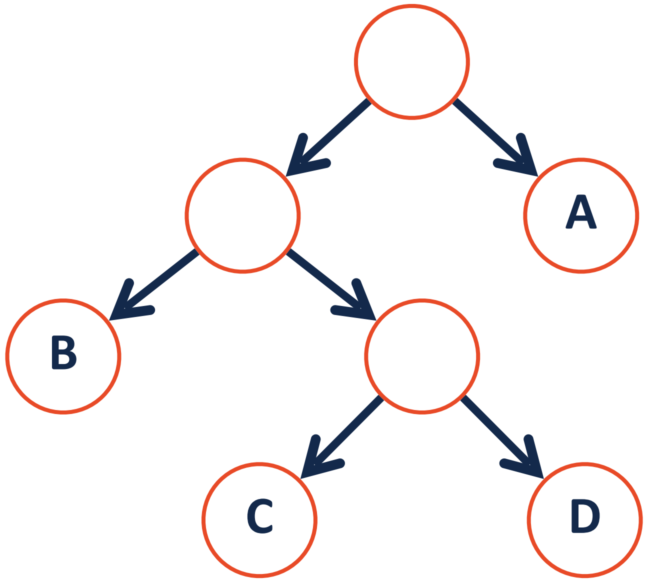
Char	Binary
A	1
B	00
C	010
D	011



Char	Binary
A	00
B	01
C	10
D	11

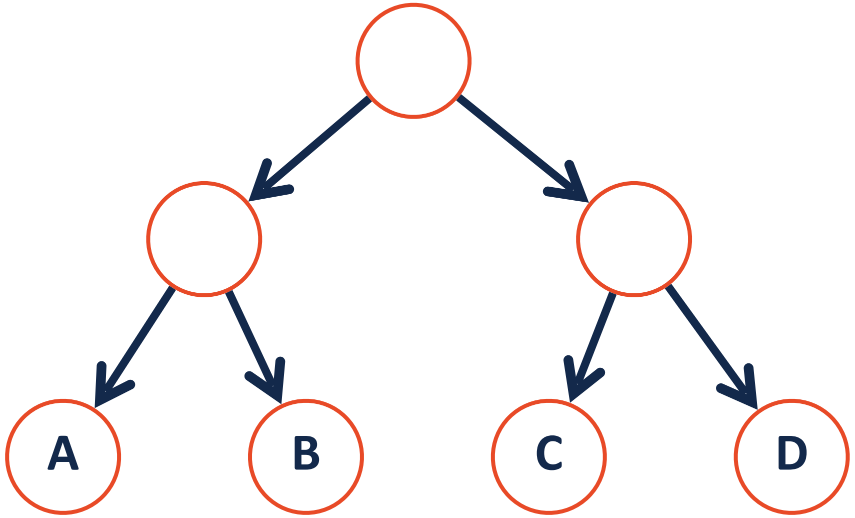
Binary Tree encoding

The **path** from root to leaf defines our encoding, but which tree is best?



Going left = 0

Going right = 1

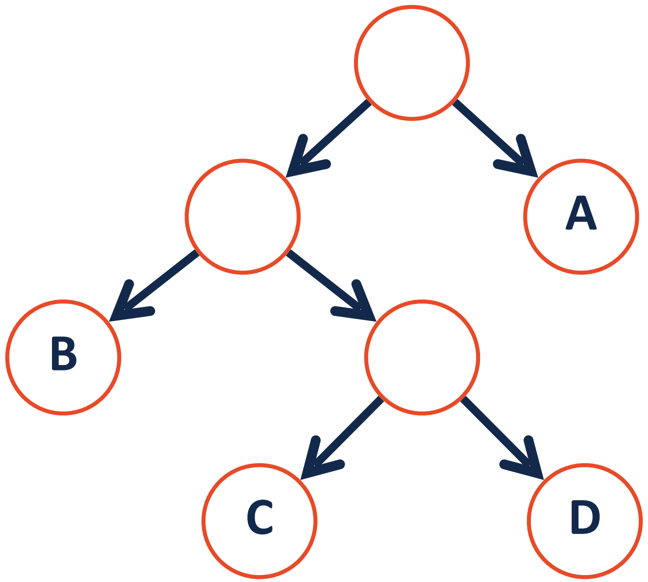


Char	Binary
A	1
B	00
C	010
D	011

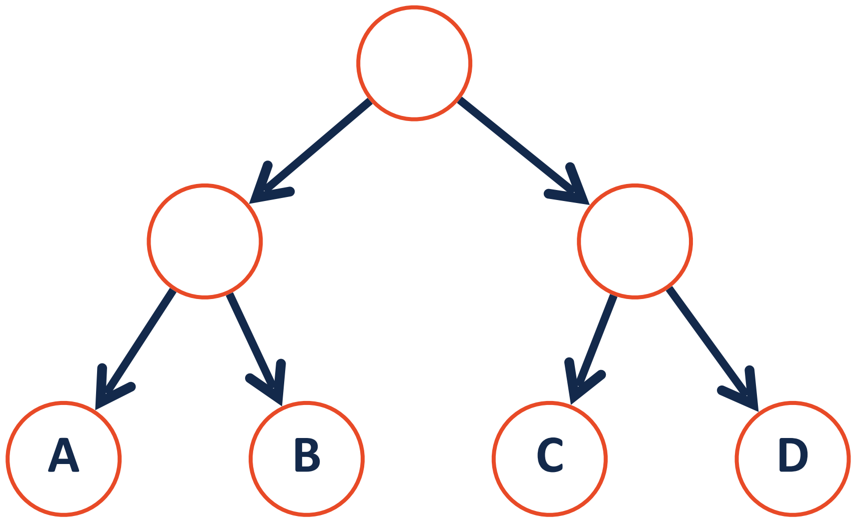
Char	Binary
A	00
B	01
C	10
D	11

Binary Tree encoding

If my frequencies are {A : 7 | B : 5 | C : 2 | D : 4 }, which tree was better?



Char	Binary
A	1
B	00
C	010
D	011



Char	Binary
A	00
B	01
C	10
D	11

Building the Huffman Tree

The **Huffman Tree** is the tree with the optimal total path length for a given set of characters and their frequencies.

Step 1: Calculate the frequency of every character in text and order by increasing frequency. Store in a queue (a sorted list).

`Input:` `’feed me more food’`

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Building the Huffman Tree

Step 2: Build a tree from the bottom up. Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

Input:

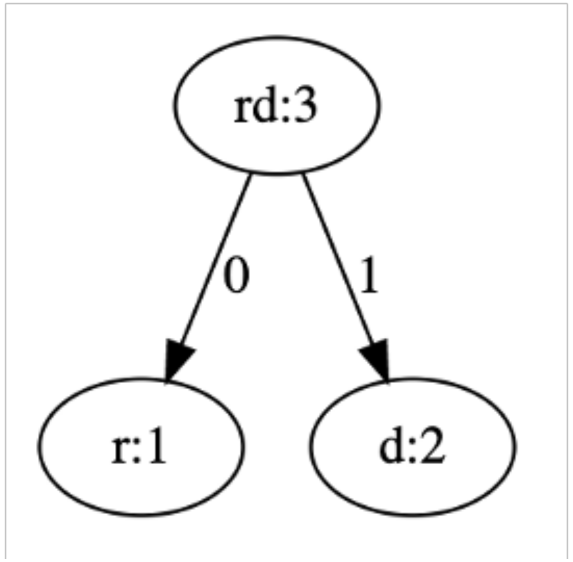
r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Building the Huffman Tree

Step 2: Build a tree from the bottom up. Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

Input:

r : 1 | d : 2 | f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4



Output:

Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

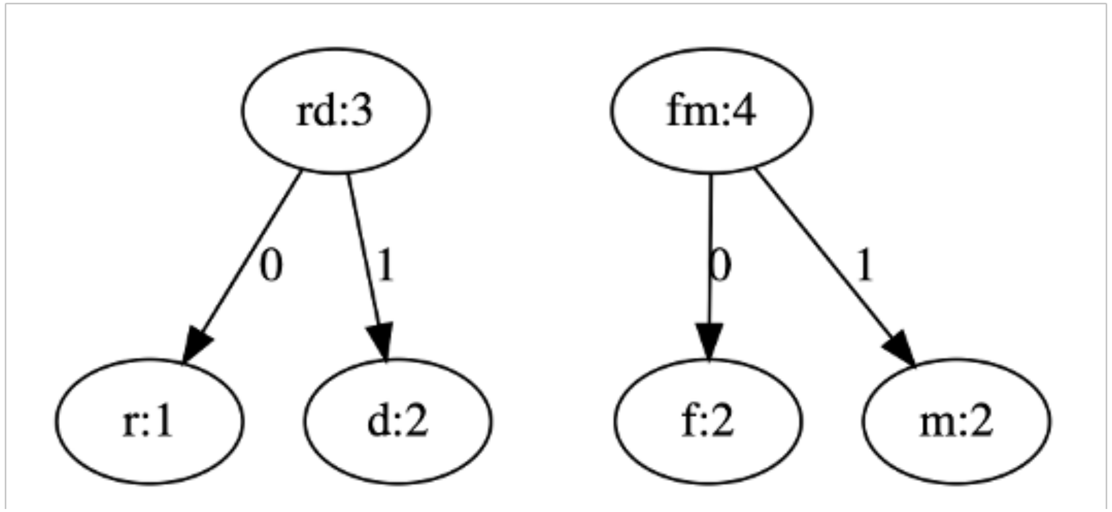
Single: f : 2 | m : 2 | o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3

Output:

Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4



Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

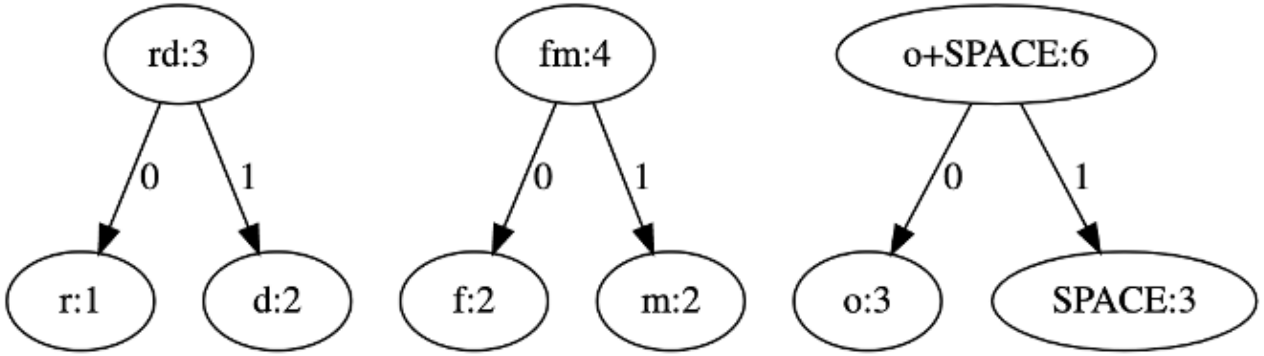
Single: o : 3 | 'SPACE' : 3 | e : 4

Merged: rd : 3 | fm : 4

Output:

Single: e : 4

Merged: rd : 3 | fm : 4 | o'SPACE' : 6



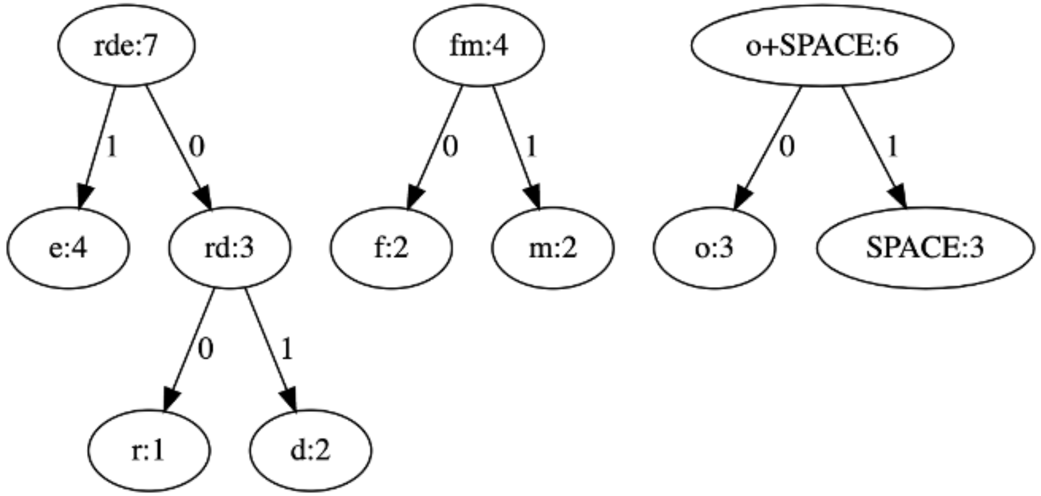
Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

Single: e : 4

Merged: rd : 3 | fm : 4 | o'SPACE' : 6



Output:

Single:

Merged: fm : 4 | o'SPACE' : 6 | rde : 7

Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items (while considering both single characters and merged characters).

Input:

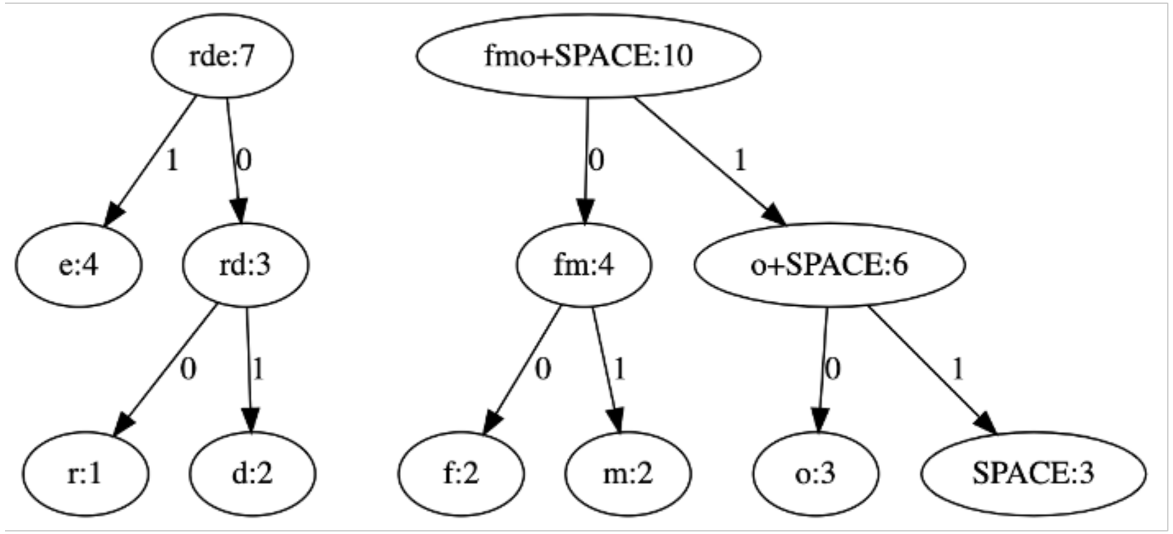
Single:

Merged: fm : 4 | o'SPACE' : 6 | rde : 7

Output:

Single:

Merged: rde : 7 | fmo'SPACE' : 10



Building the Huffman Tree

Step 4: Stop when there is only a single item in either queue.

Input:

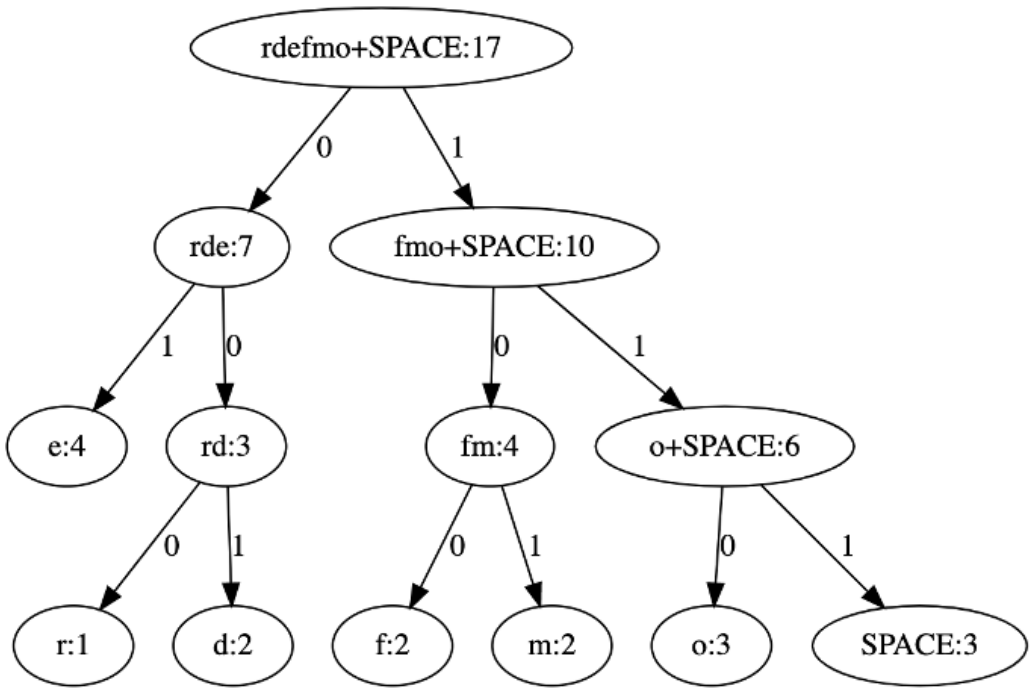
Single:

Merged: rde : 7 | fmo'SPACE' : 10

Output:

Single:

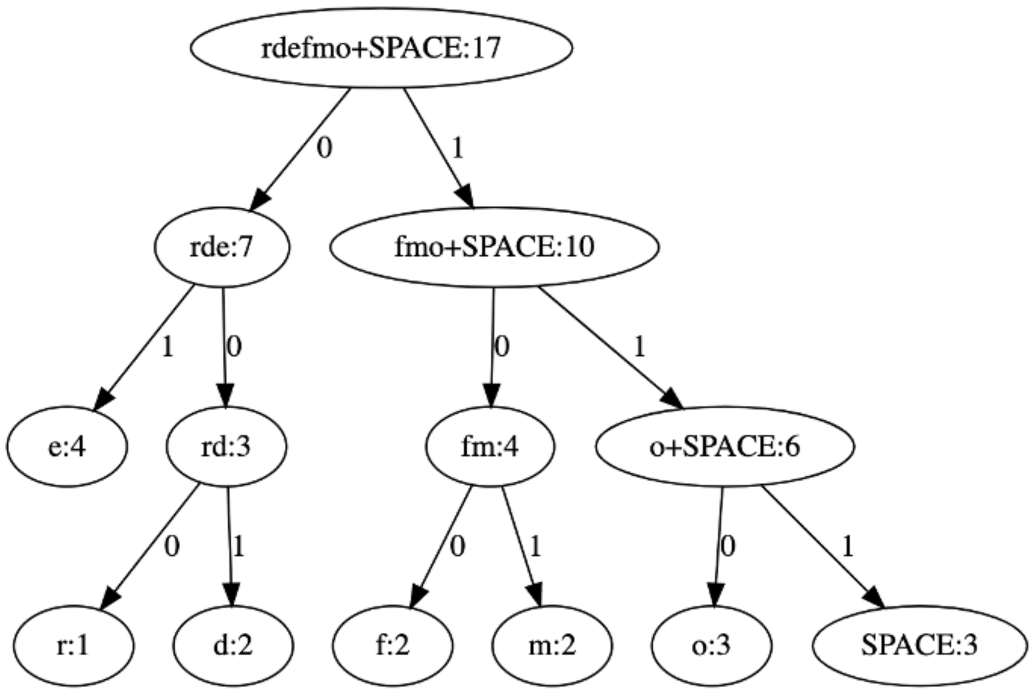
Merged: rdefmo'SPACE' : 17



Encoding using the Huffman Tree

The path through the tree defines each individual character's encoding!

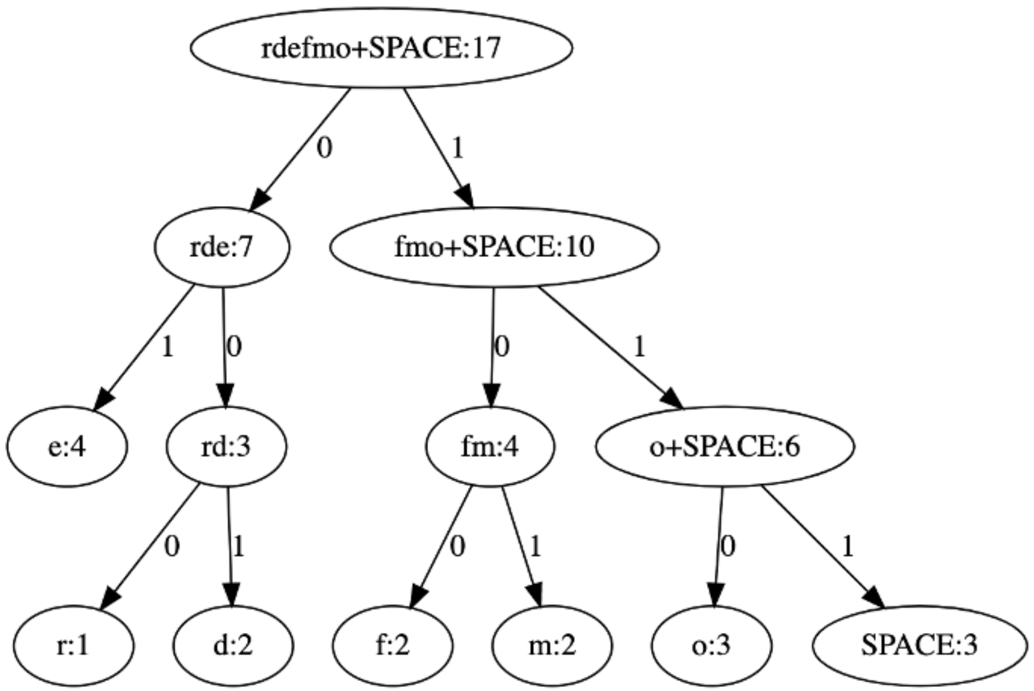
Char	Binary
f	
e	
d	
m	
r	
o	
' '	



Encoding using the Huffman Tree

The path through the tree defines each individual character's encoding!

Char	Binary
f	100
e	01
d	001
m	101
r	000
o	110
' '	111

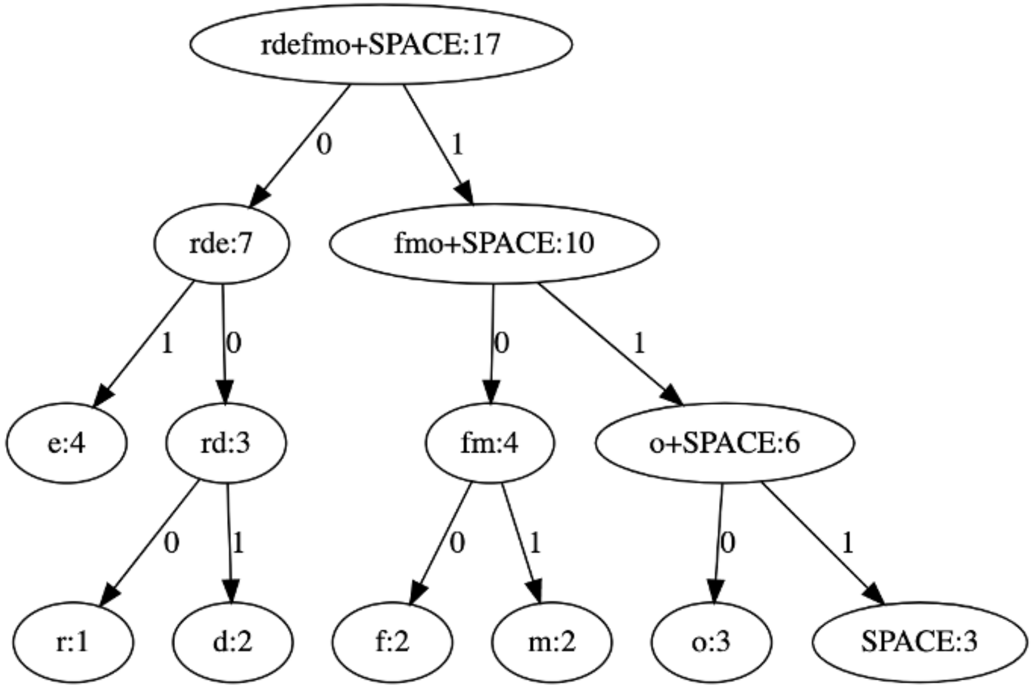


Decoding using the Huffman Tree

We can decode by walking through the tree using 0s and 1s as instructions!

Input: `10001010011111 0101`

Output:



Assignment Tips

Your assignment is to implement *just* encoding. Decoding is for fun.

1. Create a method to find the smallest bstNode (by frequency)

```
getSmallest(single, merged)
```

2. Build a Huffman Tree based on an input string

```
buildHuffman(instring)
```

3. Given a Huffman Tree, build a dictionary of all the characters encodings

```
buildEncoder(node, code, outDict)
```