Huffman Encoding

Priority Queue

- Like a queue, but items removed in sorted order rather than in the order added
  - “sorted” according to comparable interface

Method	Behavior
PriorityQueue<E>()	Constructs a priority queue containing objects of type E (must implement Comparable interface)
add(E value)	Inserts the value into the priority queue
peek()	Returns the “smallest” item in the priority queue
remove()	Returns and removes the “smallest” item from the priority queue
size()	Returns the number of items in the priority queue

Overview: Encoding

- Computers store all values as 1s and 0s
  - 1s and 0s represent numbers
- Characters each have a number used to “encode” it
  - “Default” encoding is ASCII
    - Uses 8 bits to represent each character
    - Number between 0 and 255

Char	Int	Binary
a	97	01100001
b	98	01100010
c	99	01100011
e	101	01100101
g	103	01100111
w	119	01110111
‘ ‘ (space)	32	00100000

Encoding/Decoding with ASCII

- Encoding:
  - Replace each character of text with its binary representation
    - This includes all punctuation, whitespace, etc.
- Decoding:
  - Take the binary encoding and break it up into chunks of 8 bits
  - Use the encoding table to find which letter each chunk represents

Encoding Example

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

- wigg

Char	Int	Binary
a	97	01100001
b	98	01100010
e	101	01100101
g	103	01100111
i	105	01101001
w	119	01110111
‘ ‘ (space)	32	00100000

Decoding Example

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

- 01110111011010010110011101100111

Char	Int	Binary
a	97	01100001
b	98	01100010
e	101	01100101
g	103	01100111
i	105	01101001
w	119	01110111
‘ ‘ (space)	32	00100000

Fixed Width vs Variable Width

- ASCII is an example of *fixed width encoding*
  - Each character's encoding is the same size (8 bits for ASCII)
- Huffman is an example of *variable width encoding*
  - Different characters may have different length encodings
  - Why do this? Compression!
    - Some characters are more common than others, give the more common characters shorter code words (even if rare characters get longer ones)
  - This makes encoding/decoding tricky…

A “Bad” Variable Length Encoding

- 1010
- Goal: pick encodings to make this unambiguous

Character	Encoding
a	01
e	1
t	0
r	010
n	10

Huffman Coding Strategy

- Use variable length codes to take up less space
  - Don’t have codes for unused characters
  - Give frequent characters shorter codes
  - Give infrequent characters longer codes
- Select code words to make decoding unambiguous
  - You can tell when one char ends and the next begins
  - We will use a binary tree!

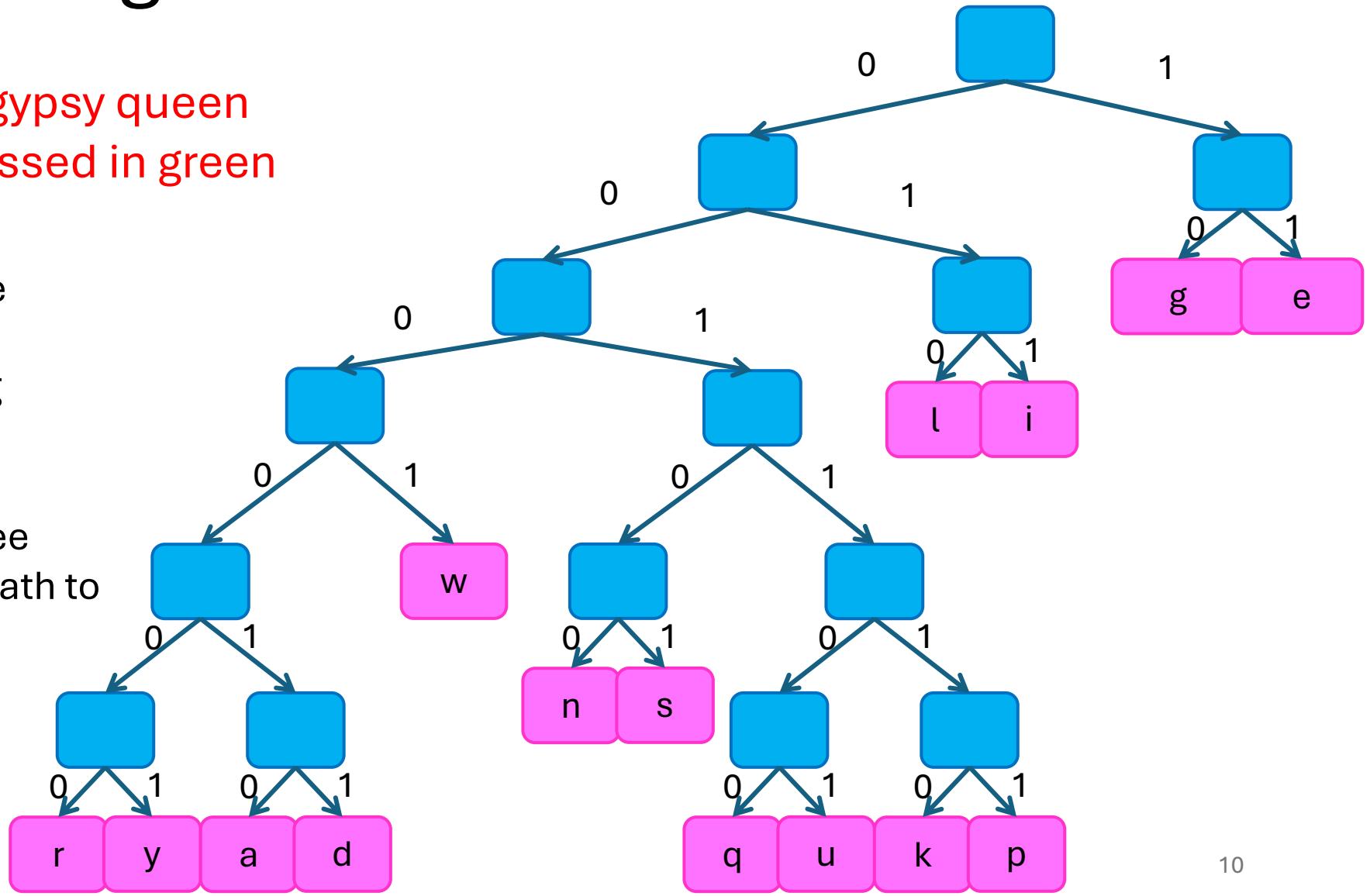
Huffman Coding

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

Characters are leaves in the tree
0 = go left, 1 = go right
Path to character is its encoding

To encode “wigg”:

- Find each character in the tree
- Replace character with the path to that node
- Result: 00010111010



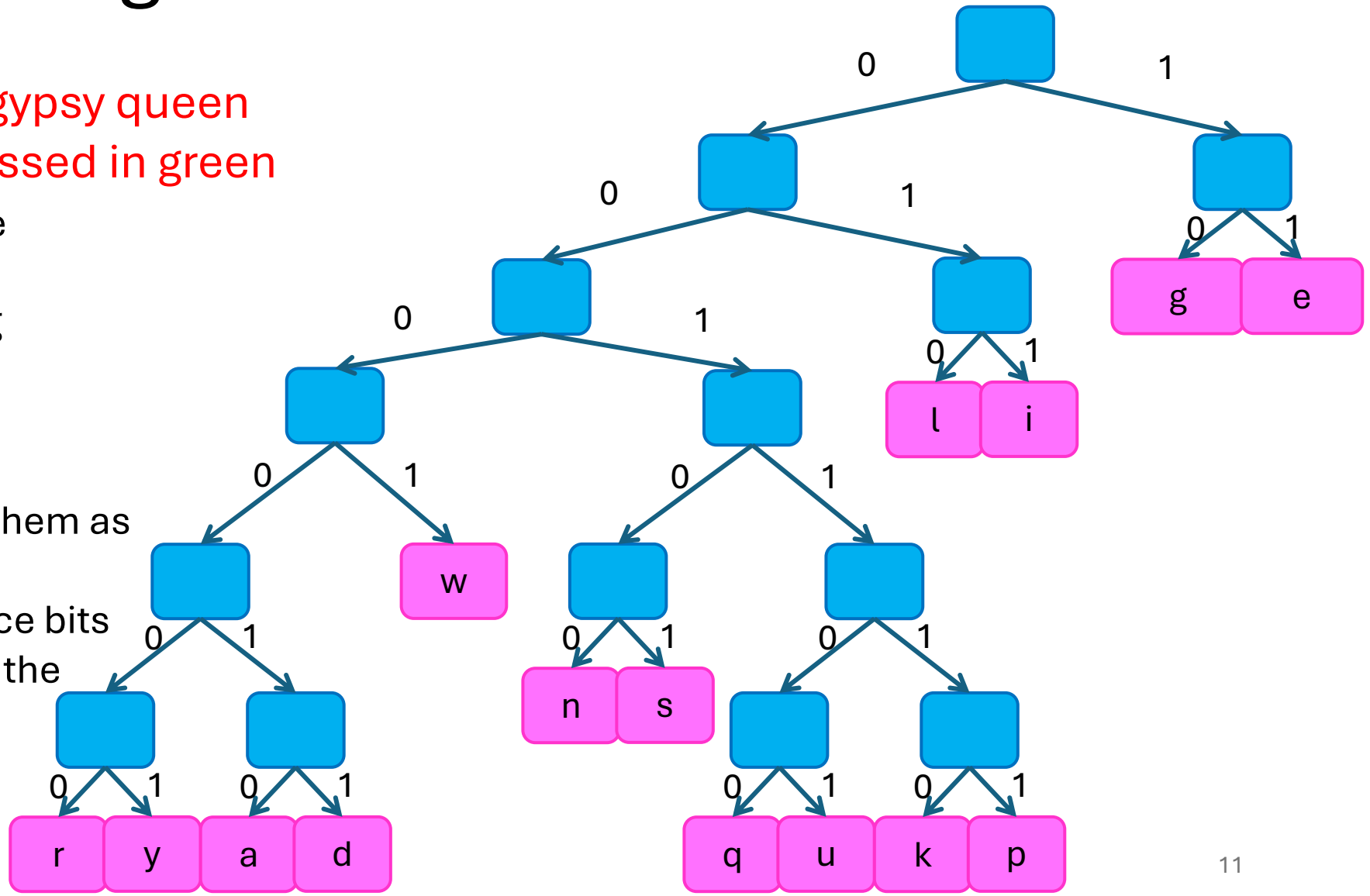
Huffman Coding

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

Characters are leaves in the tree
0 = go left, 1 = go right
Path to character is its encoding

To decode “00010111010”:

- Start from root of the tree
- Read bits one at a time, use them as “directions”
- When you reach a leaf, replace bits with that character, return to the root of the tree
- Result: “wigg”



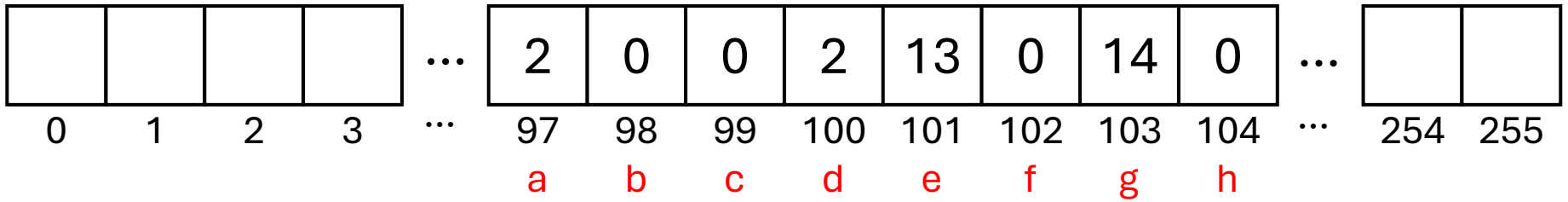
P3 Process

- Encoding:
  - Generate the Huffman tree for the text given (algorithm soon)
  - Store the tree in a .code file
  - Encode the text using that .code file
- Decoding:
  - Rebuild the stored tree (trickiest part of assignment)
  - Read the encoding one character at a time to navigate the tree
  - Print out a character each time you hit a leaf node

Encoding (Generating Huffman Tree)

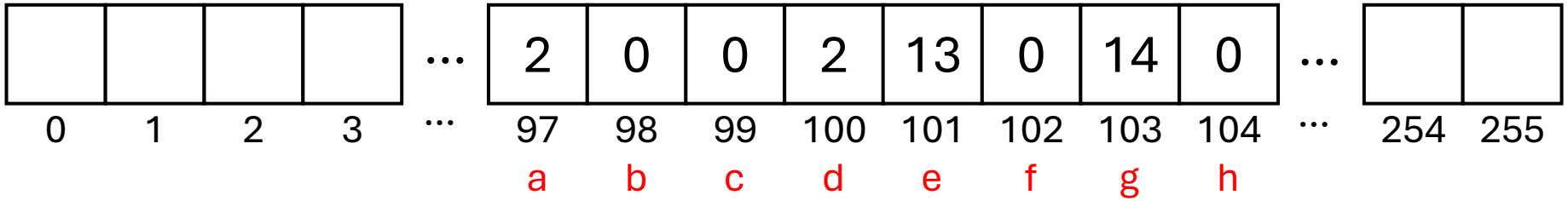
- Step 1: Count occurrences of each character
  - We do this for you!
    - We give you an array where each index is an ascii character, the value is the number of occurrences

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green



Encoding (Generating Huffman Tree)

- Step 1: Count occurrences of each character
- Step 2: Make a HuffmanTreeNode per character
  - You will write this class
  - It must implement the comparable interface
  - Nodes should be compared by the character frequency



Char: ‘a’
Freq: 2

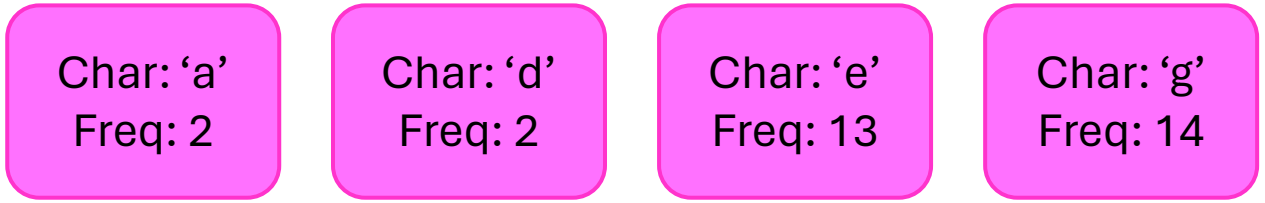
Char: ‘d’
Freq: 2

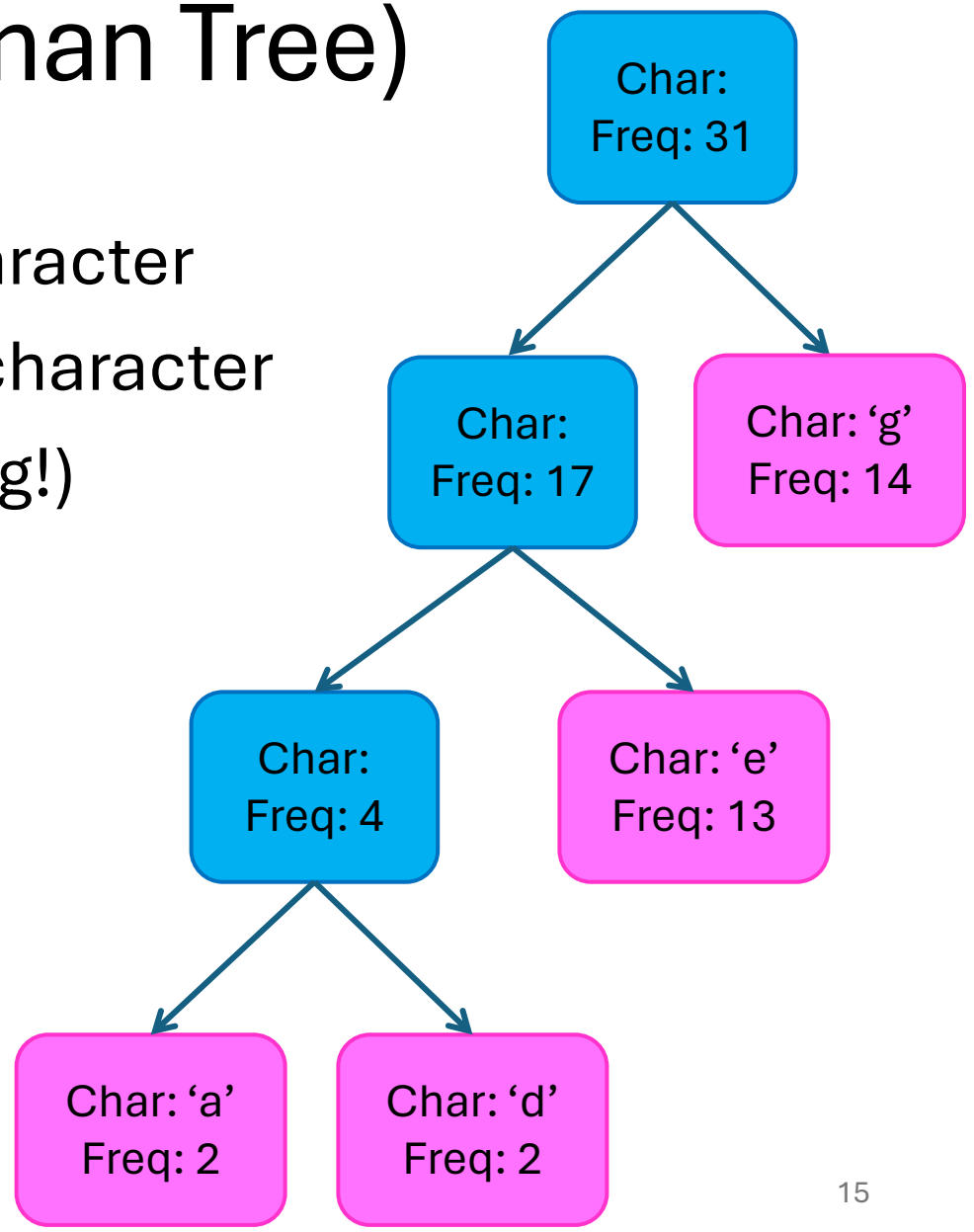
Char: ‘e’
Freq: 13

Char: ‘g’
Freq: 14

Encoding (Generating Huffman Tree)

- Step 1: Count occurrences of each character
- Step 2: Make a HuffmanTreeNode per character
- Step 3: Build the Tree (algorithm coming!)

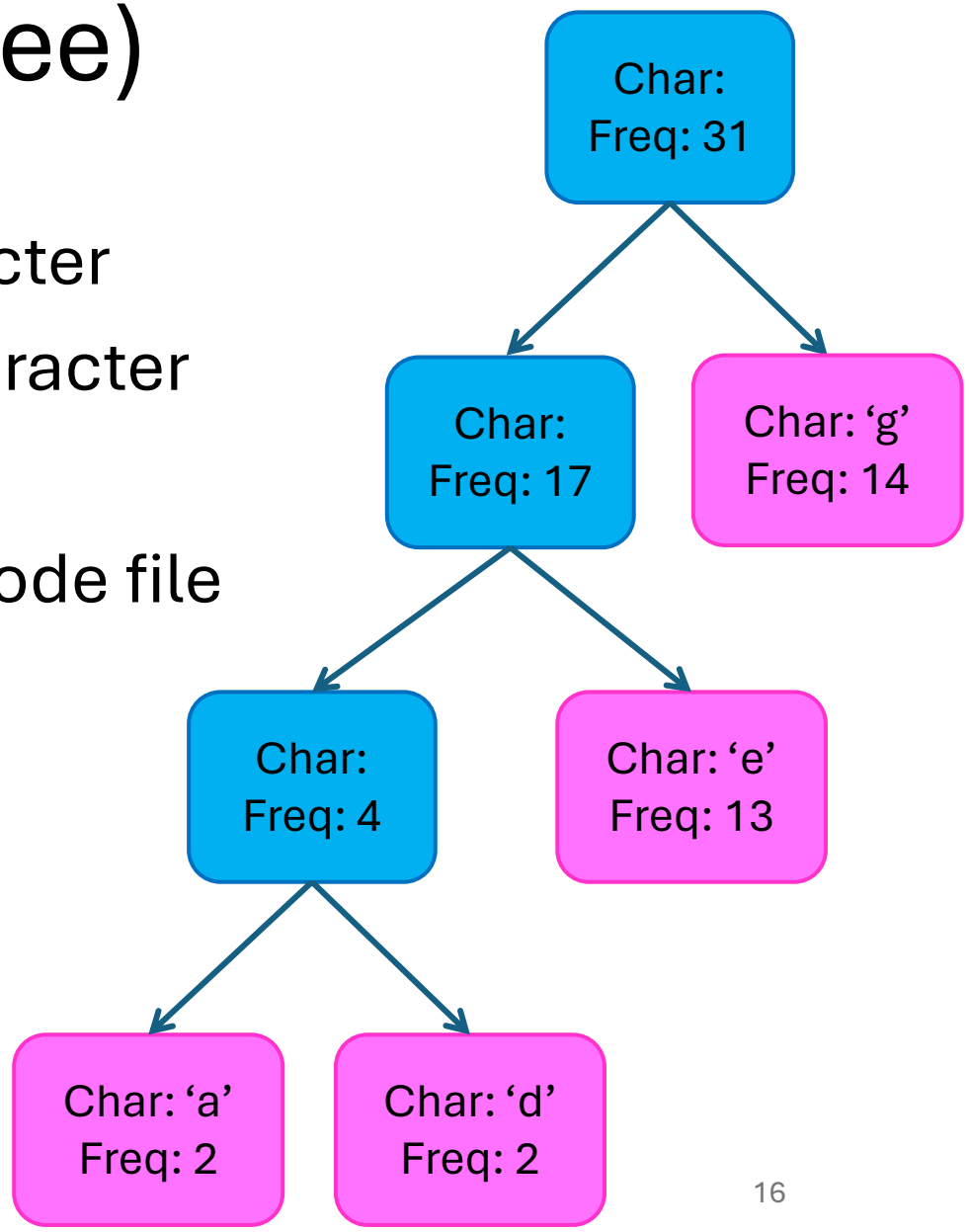




Encoding (Storing Huffman Tree)

- Step 1: Count occurrences of each character
- Step 2: Make a HuffmanTreeNode per character
- Step 3: Build the Tree (algorithm coming!)
- Step 4: Save per-character encoding to .code file

Ascii value	97
Path	000
Ascii value	100
Path	001
Ascii value	101
Path	01
Ascii value	103
Path	1



Encoding (Use the Codes)

- Step 1: Count occurrences of each character
- Step 2: Make a HuffmanTreeNode per character
- Step 3: Build the Tree (algorithm coming!)
- Step 4: Save per-character encoding to .code file
- Step 5: Replace characters with their codes

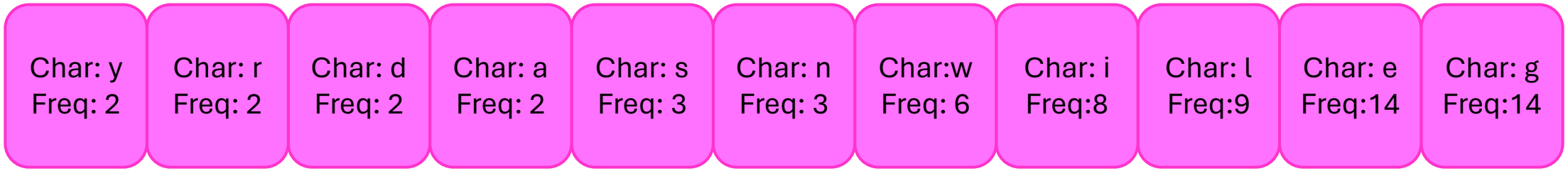
a	97
	000
d	100
	001
e	101
	01
g	103
	1

“addage” becomes
000100100000101

Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

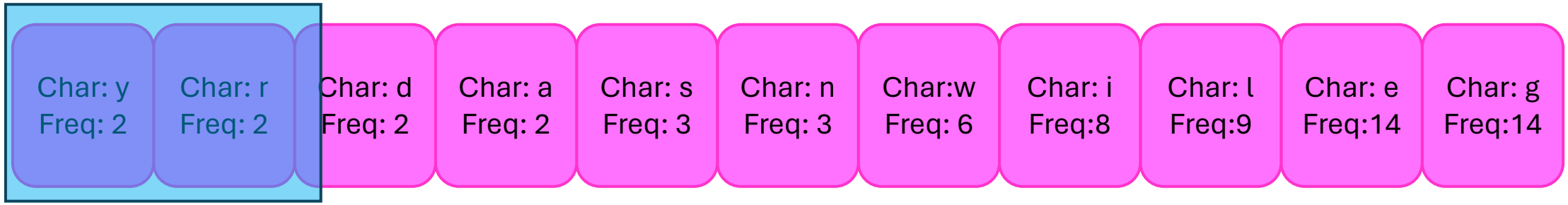
- From step 2 we have a HuffmanTreeNode per character
- Put all nodes into a priority queue, ordered by frequency



Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

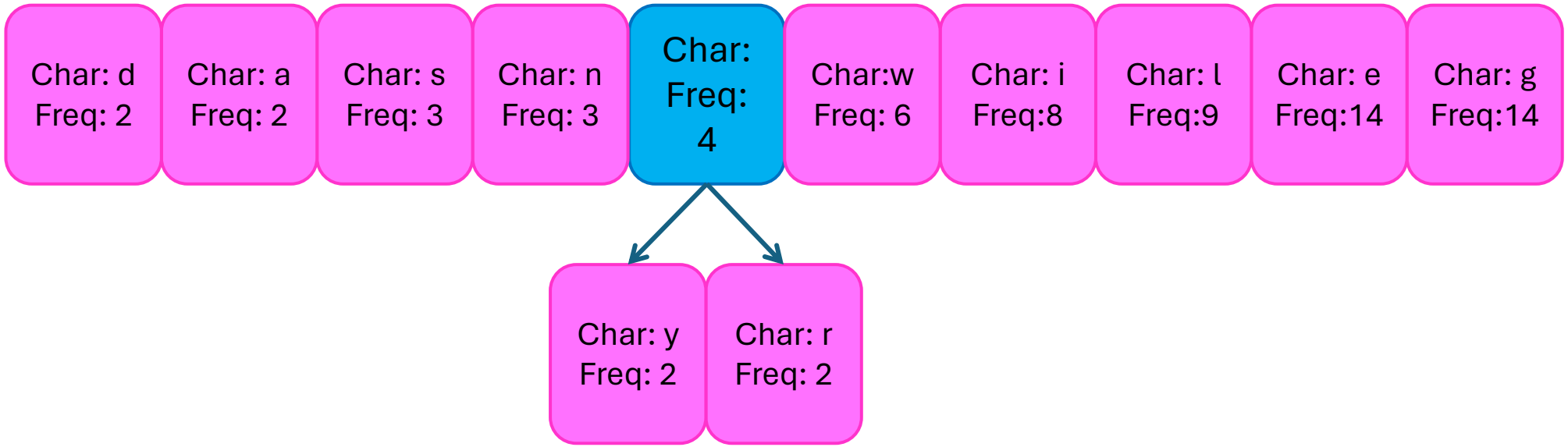
- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

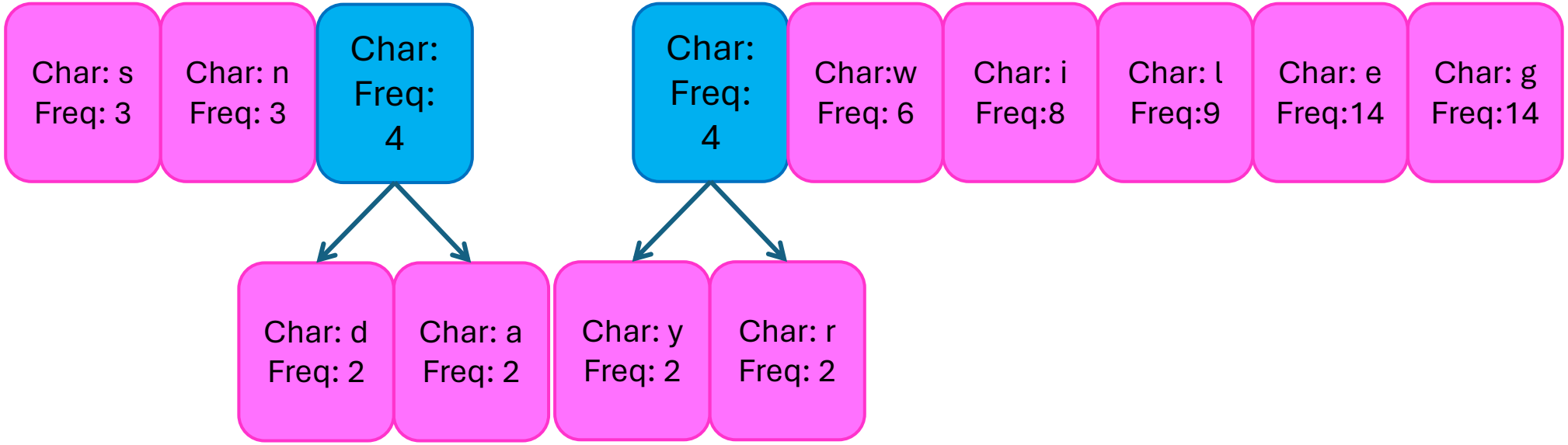
- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

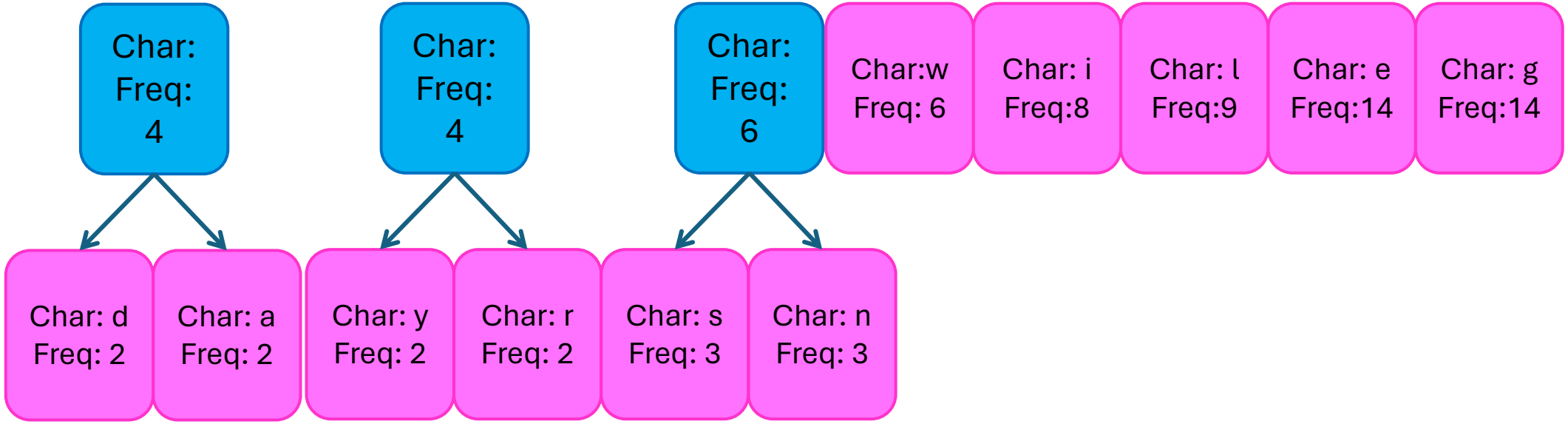
- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

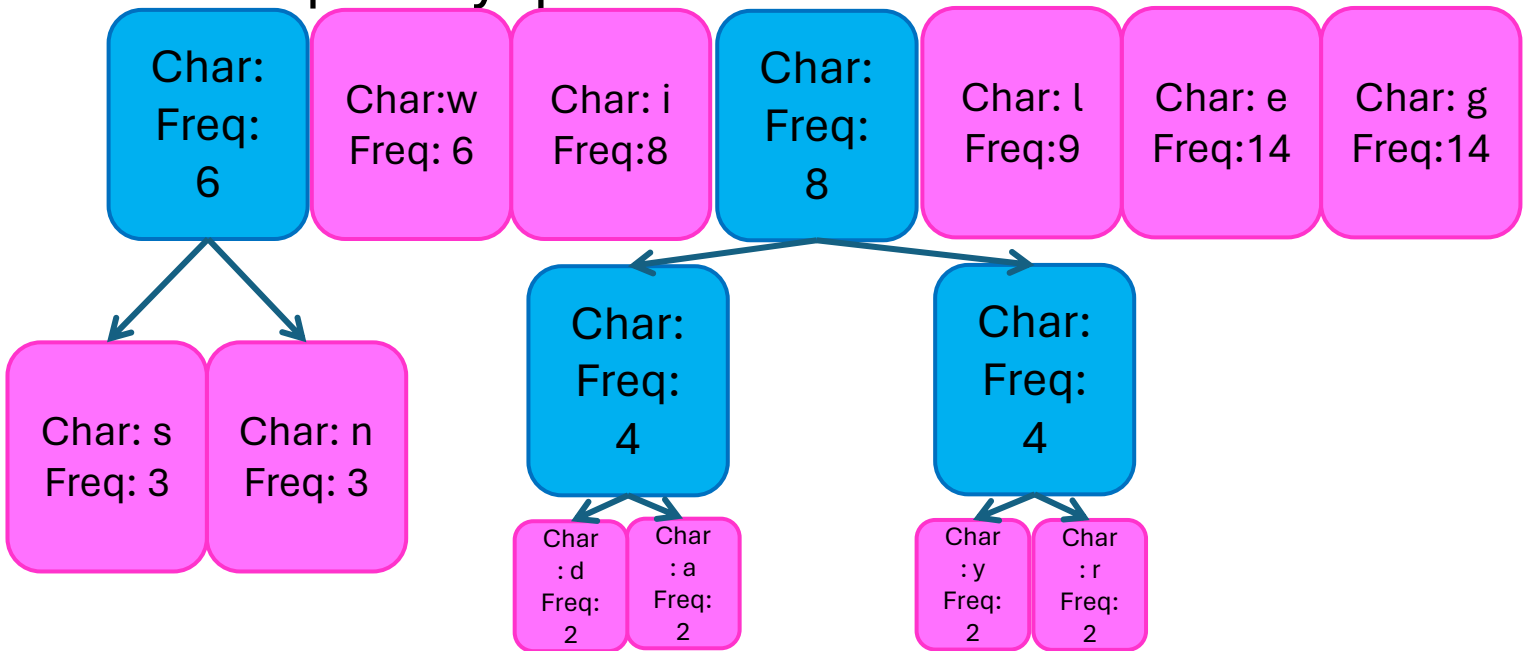
- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

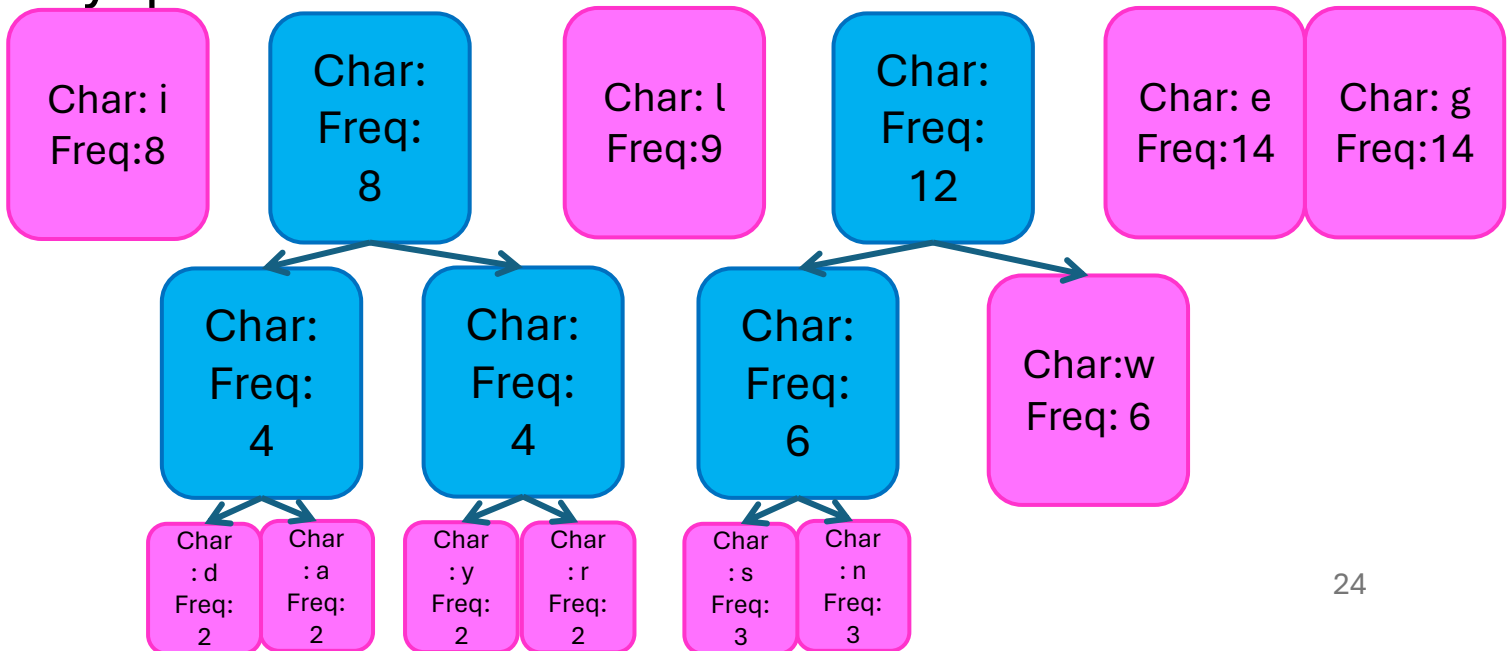
- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



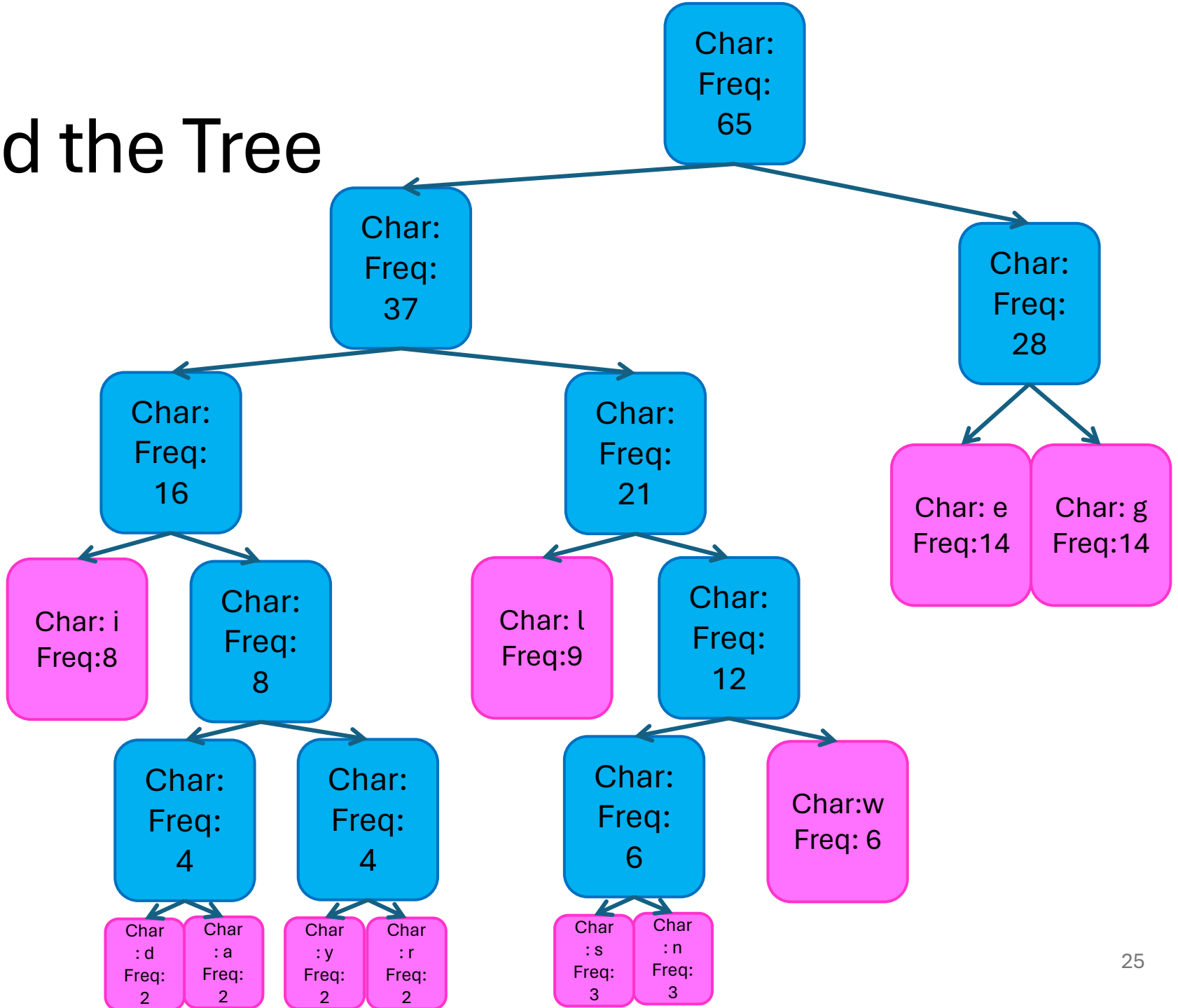
Step 3: Build the Tree

wiggle wiggle wiggle like a gypsy queen
wiggle wiggle wiggle all dressed in green

- While there is more than 1 node in the priority queue:
  - Remove the least-frequent pair
  - Make them children of a new node
  - Make new node's frequency their sum
  - Add new node to the priority queue



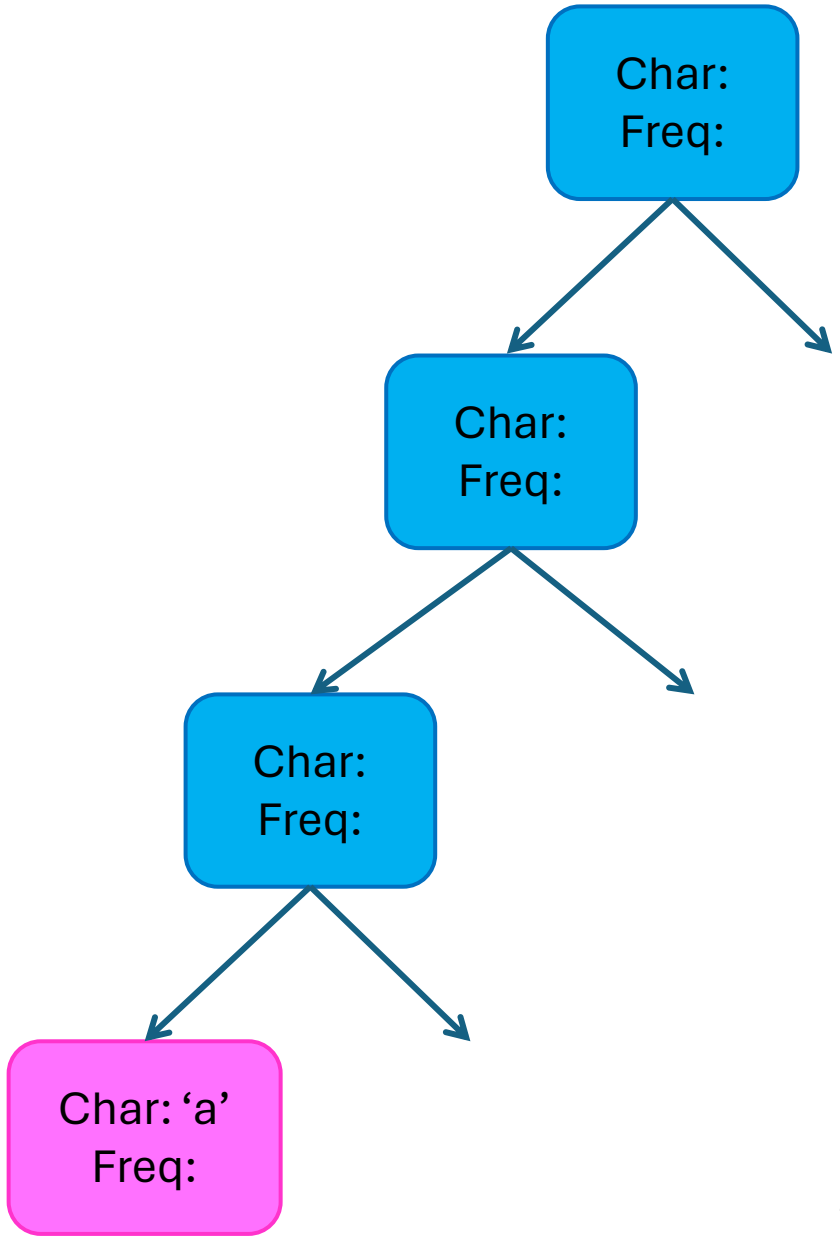
Step 3: Build the Tree



Decoding (Rebuild the Tree)

- From Encoding we have the .code file
- Use the .code file to build the tree
  - Use each path at a time to “branch”

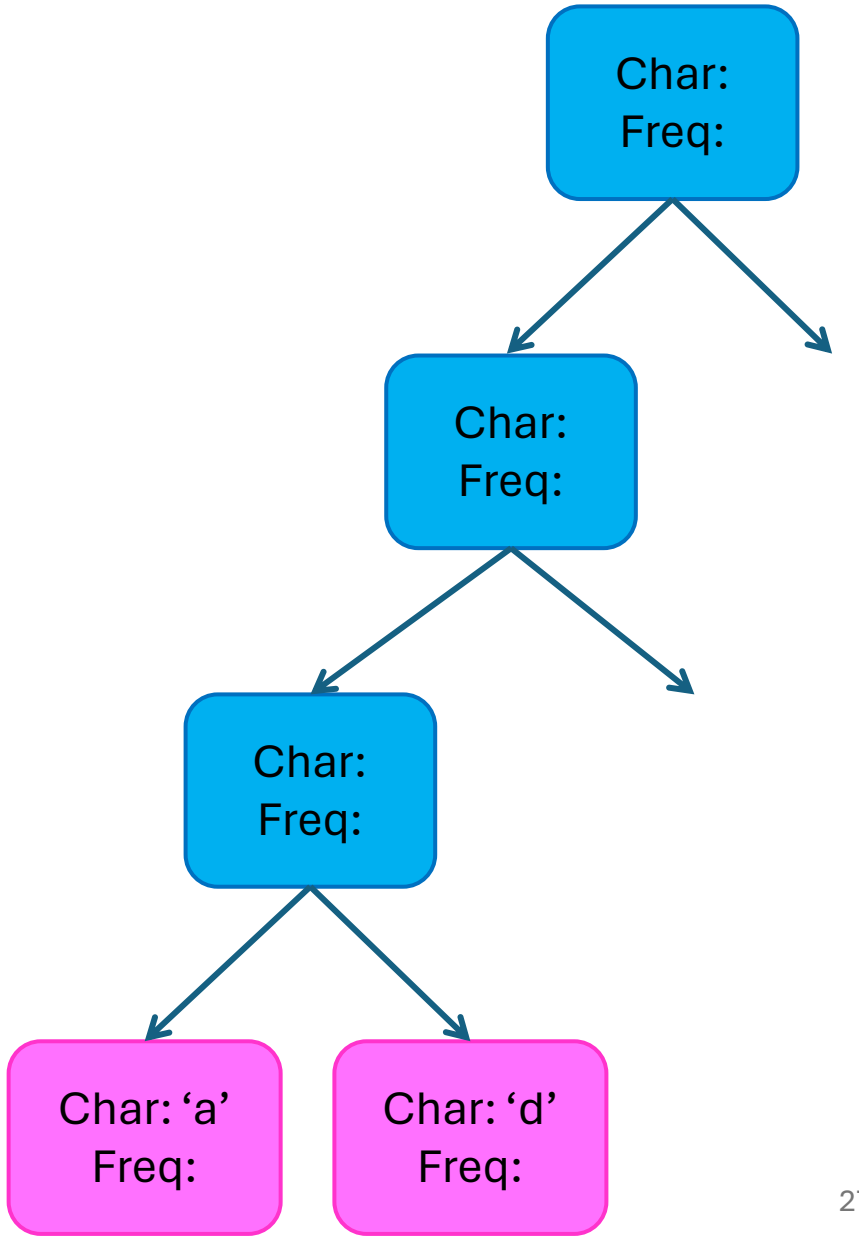
Ascii value	**97**
Path	**000**
Ascii value	100
Path	001
Ascii value	101
Path	01
Ascii value	103
Path	1



Decoding (Rebuild the Tree)

- From Encoding we have the .code file
- Use the .code file to build the tree
  - Use each path at a time to “branch”

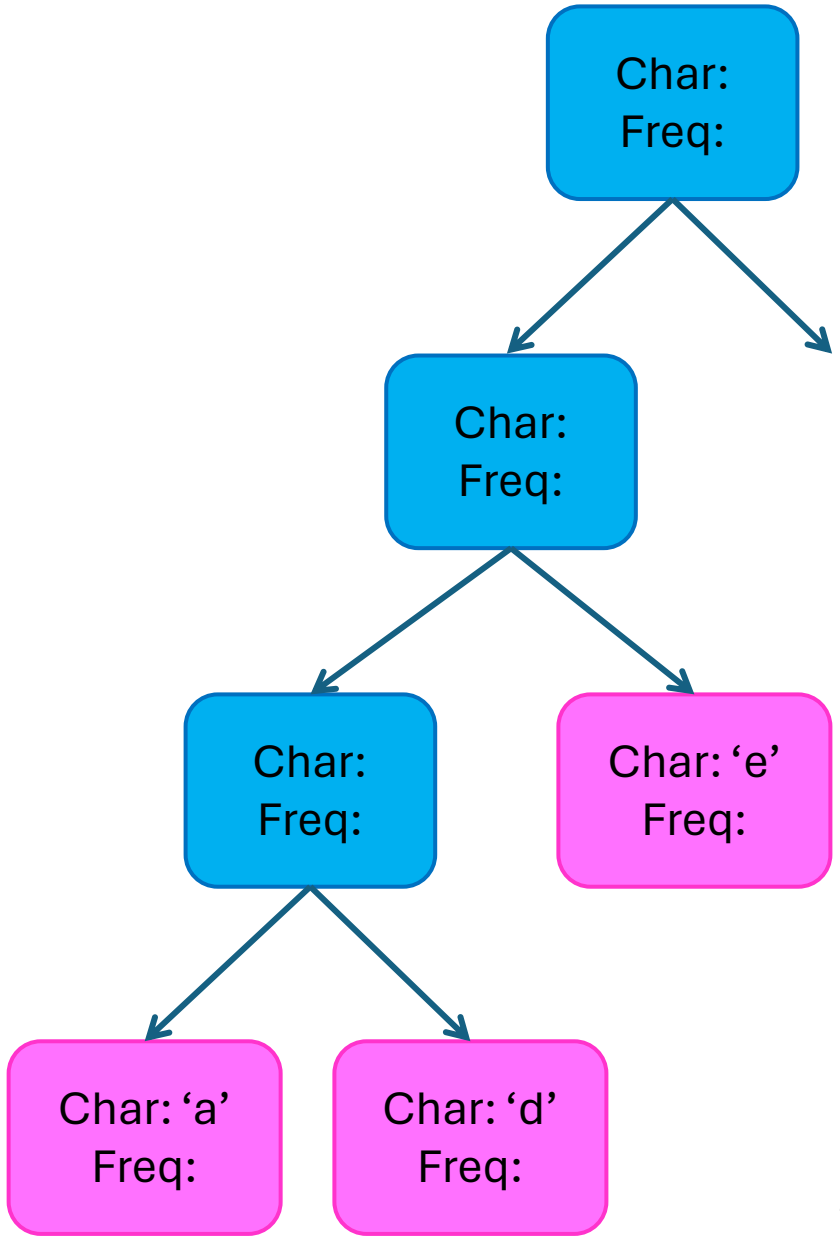
Ascii value	**97**
Path	**000**
Ascii value	**100**
Path	**001**
Ascii value	101
Path	01
Ascii value	103
Path	1



Decoding (Rebuild the Tree)

- From Encoding we have the .code file
- Use the .code file to build the tree
  - Use each path at a time to “branch”

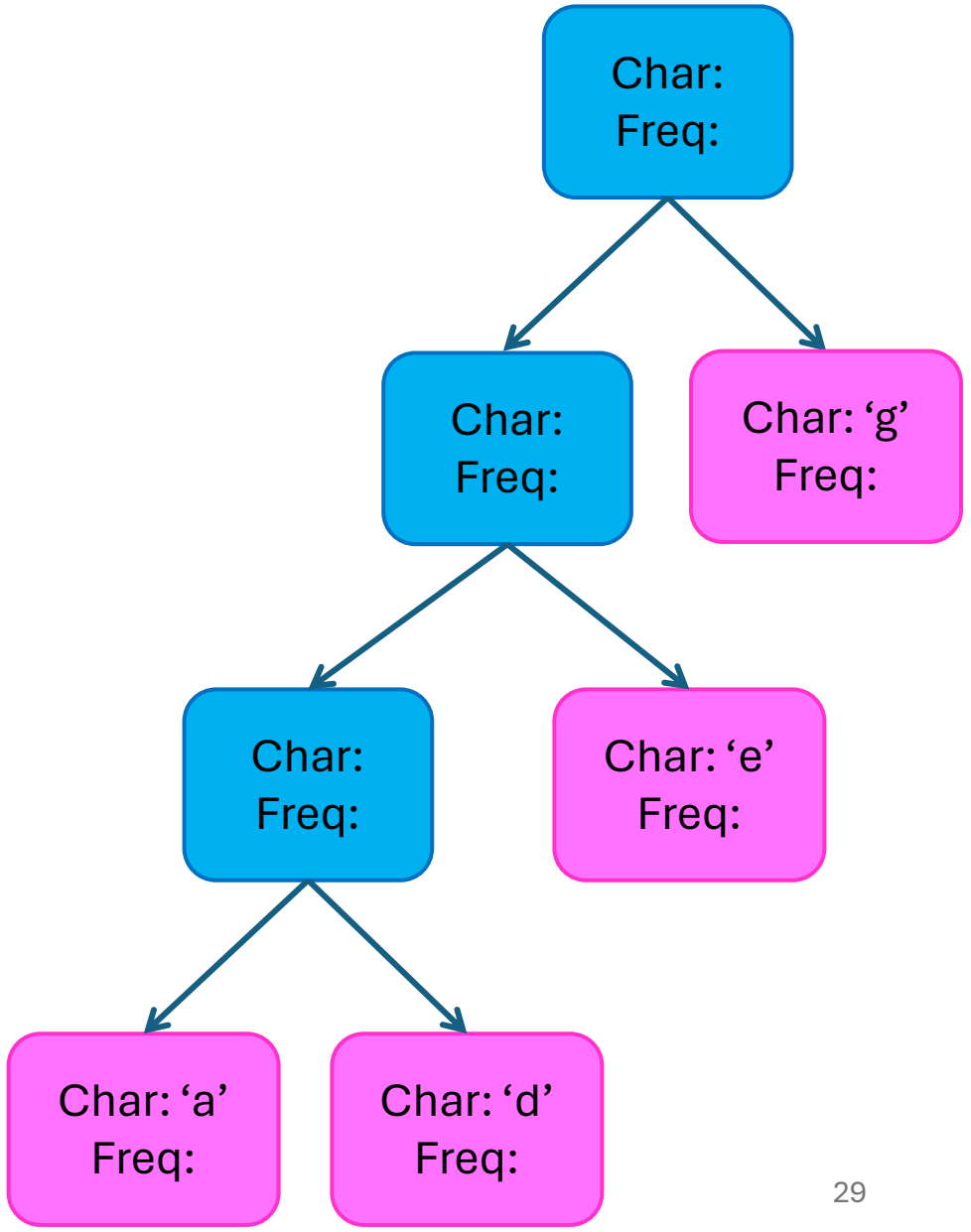
Ascii value	**97**
Path	**000**
Ascii value	**100**
Path	**001**
Ascii value	**101**
Path	**01**
Ascii value	103
Path	1



Decoding (Rebuild the Tree)

- From Encoding we have the .code file
- Use the .code file to build the tree
  - Use each path at a time to “branch”

Ascii value	**97**
Path	**000**
Ascii value	**100**
Path	**001**
Ascii value	**101**
Path	**01**
Ascii value	**103**
Path	**1**



BitInputStream Class

- Used to read 1 bit at a time
- Works a lot like Scanner

Method	Behavior
BitInputStream(String file)	Creates a stream of bits from file
hasNextBit()	Returns true if bits remain in the stream
nextBit()	Reads and returns the next bit in the stream

Summary

- Part A: Compression
  - public HuffmanCode(int[] counts)
    - Slides 14-15
    - Slides 18-25
  - public void save(PrintStream out)
    - Slides 16-17
- Part B: Decompression
  - public HuffmanCode(Scanner input)
    - Slides 26-29
  - public void translate(BitInputStream in, PrintStream out)
    - Slide 11